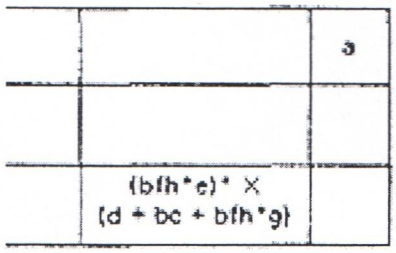
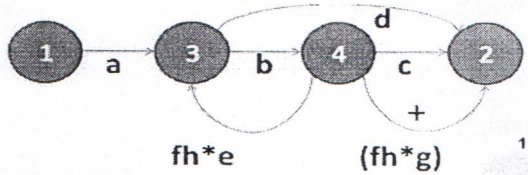


- There is only one node to remove now, node 3. This will result in a term in the (1,2) entry whose value is

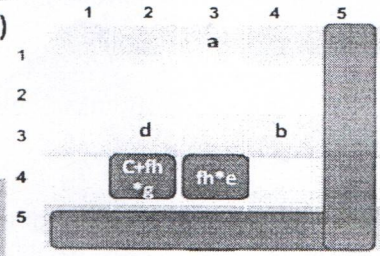
$$a(bfh^*e)^*(d + bc + bfh^*g)$$

Node Reduction Algorithm

*Eliminating node 5 ... the self loop is represented with a * and the outgoing link from the node is multiplied . . .*



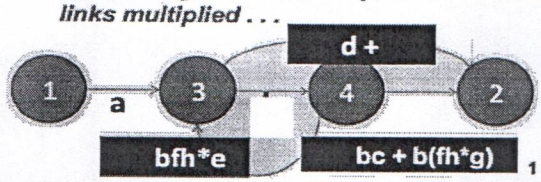
Now, we eliminate node 5-



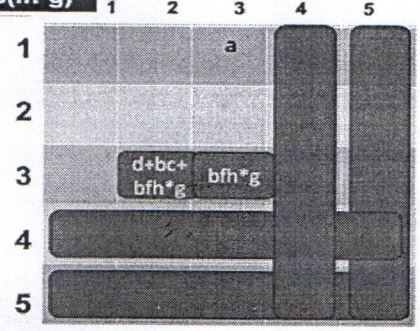
***TIP:** The out-link of the node removed will correspond to the row and the in-link will correspond to the column*

Node Reduction Algorithm

Eliminating node 4 ... the parallel link is added up and serial links multiplied . . .

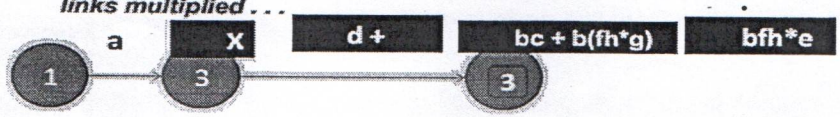


Now, we eliminate node 4-



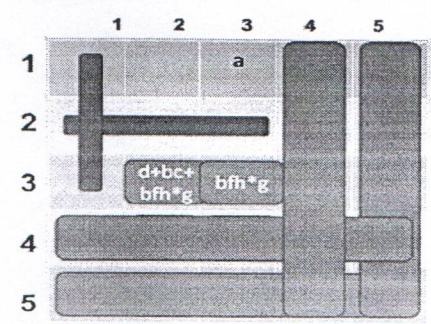
Node Reduction Algorithm

Eliminating node 4 ... the parallel link is added up and serial links multiplied . . .



Now, we should ideally eliminate node 3-
Since row 2 and column 2 are empty, these are dropped. Values of nodes 1 and 3 are retained.

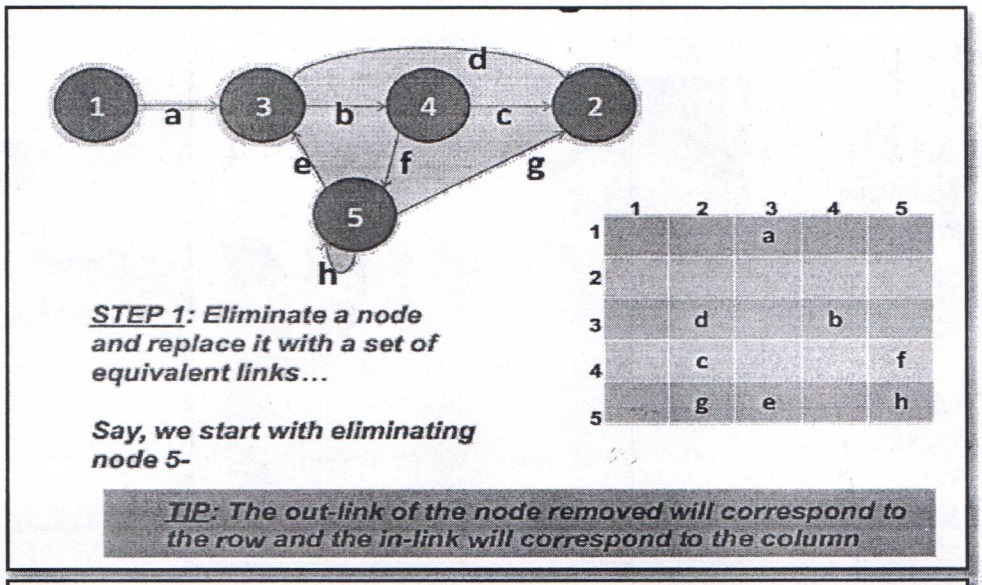
If row n and its corresponding column n is not empty, them we need to continue removing node 3 and retain values for nodes 1 & 2.

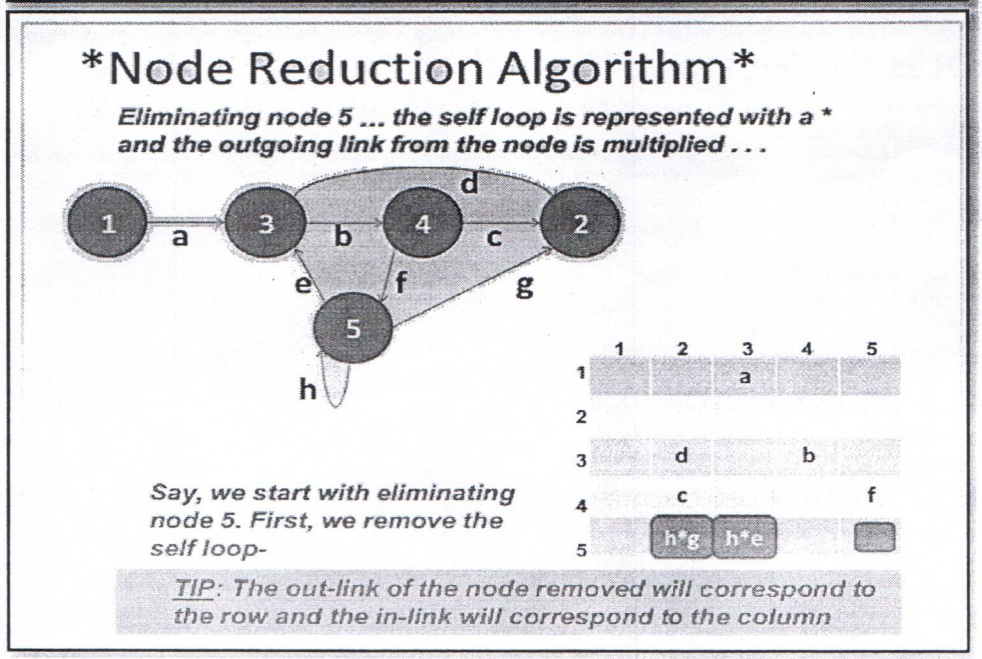


4. Node Reduction Algorithm (General):-

The matrix reduction method is more methodical than graphical method.

1) Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2) Combine the parallel terms and simplify as you can.
3) Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4) The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.





$$\sum_{i=1}^{\infty} A^i = A + A^2 + A^3 \cdot \cdot \cdot A^{\infty}$$

- Therefore, the original infinite sum can be replaced by

$$\sum_{i=1}^{\infty} A^i = A(A + I)^*$$

Some matrix properties:-

Node Reduction Algorithm – *some matrix properties:*

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

interchanging rows 3 & 4 ...

	1	2	*	*	5
1			a		
2					
*		c			f
*		d		b	
5		g	e		h

To interchange node names in a matrix, we must interchange both the corresponding rows and the corresponding columns...

interchanging columns 3 & 4 ...

	1	2	4	3	5
1				a	
2					
4		c			f
3		d	b		
5		g		e	h

- More generally, given two matrices A and B, with entries a_{ik} and b_{kj}, respectively, their product is a new matrix C, whose entries are c_{ij}, where:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} a_{11}a_{12}a_{13}a_{14} \\ a_{21}a_{22}a_{23}a_{24} \\ a_{31}a_{32}a_{33}a_{34} \\ a_{41}a_{42}a_{43}a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11}b_{12}b_{13}b_{14} \\ b_{21}b_{22}b_{23}b_{24} \\ b_{31}b_{32}b_{33}b_{34} \\ b_{41}b_{42}b_{43}b_{44} \end{bmatrix} = \begin{bmatrix} c_{11}c_{12}c_{13}c_{14} \\ c_{21}c_{22}c_{23}c_{24} \\ c_{31}c_{32}c_{33}c_{34} \\ c_{41}c_{42}c_{43}c_{44} \end{bmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42}$$
$$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43}$$
$$\vdots$$
$$c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42}$$
$$\vdots$$
$$c_{44} = a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}$$

- **After applying matrix powers and products for the matrix of Figure 12.1g yields**

		a		
	d		b	
	c			f
	g	e		h

A

	ad		ab	
	bc			bf
	fg	fe		fh
	ed + hg	he	eb	h^2

A^2

	abc			abf
	bfg	bfe		bfh
	fed + fhg	fhe	feb	fh^2
	ebc + hed + h^2g	h^2e	heb	ebf + h^3

A^3

3.2. The Set of All Paths

- **To use matrix operations to obtain the set of all paths between all nodes or, equivalently, a property (described by link weights) over the set of all paths from every node to every other node, using the appropriate arithmetic rules for such weights.**
- The set of all paths between all nodes is easily expressed in terms of matrix operations. It's given by the following infinite series of matrix powers:

The idea behind partition testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes.

6) Partial Ordering Relations

A partial ordering relation satisfies the **reflexive**, **transitive**, and **antisymmetric** properties.

Partial ordered graphs have several important properties: they are loop free, there is at least one maximum element, and there is at least one minimum element.

3. Powers of Matrix:-

Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry.

Let A be a matrix whose entries are a_{ij}. The set of all paths between any node i and any other node j (possibly i itself), via all possible intermediate nodes, is given by

$$a_{ij} + \sum_{k=1}^{n} a_{ik}a_{kj} + \sum_{k=1}^{n}\sum_{m=1}^{n} a_{ik}a_{km}a_{mj} + \sum_{k}^{n}\sum_{m}^{n}\sum_{l}^{n} a_{ik}a_{km}a_{ml}a_{lj} + \cdots + \sum_{k=1}^{n}\sum_{m=1}^{n}\sum_{l=1}^{n}\cdots\sum_{p=1}^{n} a_{ik}a_{km}a_{ml}\cdots a_{qp}a_{pj}$$

1. **Consider the relation between every node and its neighbor.**
2. Extend that relation by considering each neighbor as an intermediate node.
3. Extend further by considering each neighbor's neighbor as an intermediate node.
4. Continue until the longest possible nonrepeating path has been established.
5. Do this for every pair of nodes in the graph.

3.1. Matrix Powers and Products:-

Given a matrix whose entries are a_{ij}, the square of that matrix is obtained by replacing every entry with

$$a_{ij} = \sum_{k=1}^{n} a_{ik}a_{kj}$$

5. "Data object X is defined at program node *a* and used at program node *b*." The relation between nodes *a* and *b* is that there is a *du* chain between them.

- Relations are that there be an algorithm by which we can determine whether or not the relation exists between two nodes.

2.1. Properties of Relations:-

1) **Transitive Relations:-**

A relation is transitive if **aRb** and **bRc** implies **aRc**.

Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of.

2) **Reflexive Relations**

A relation R is reflexive if, for every a, **aRa**. A reflexive relation is equivalent to a self loop at every node.

Examples of reflexive relations include: equals, is acquainted with, is a relative of.

3) **Symmetric Relations**

A relation R is symmetric if for every a and b, **aRb** implies **bRa**.

A symmetric relation mean that if there is a link from a to b then there is also a link from b to a. A graph whose relations are not symmetric are called directed graph.

4) **Antisymmetric Relations**

A relation R is antisymmetric if for every a and b, if **aRb** and **bRa**, then **a=b**, or they are the same elements.

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time.

5) **Equivalence Relations**

An equivalence relation is a relation that satisfies the **reflexive**, **transitive**, and **symmetric** properties. Equality is the most familiar example of an equivalence relation.

If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation.

Connection matrix:- The connection matrix is obtained by replacing each entry with 1 if there is a link and 0 if there **isn't.**

- Each row of a matrix denotes the out links of the node corresponding to that row.
- Each column denotes the in links corresponding to that node.
- A **branch** is a node with more than one nonzero entry in its row.
- A **junction** is node with more than one nonzero entry in its column.
- A self loop is an entry along the diagonal.

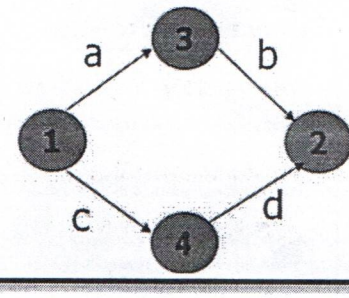
Cyclomatic Complexity:-

The Cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row.

Adding these values and then adding 1 to the sum yields the graph's Cyclomatic complexity.

		a	c
	b		
	d		

		1	1
	1		
	1		



2. RELATIONS:-

- A **relation** is a property that exists between two (usually) objects of interest.

1. "Node *a is connected* to node *b*" or *a*R*b* where "R" means "is connected to."
2. "*a* >= *b*" or *a*R*b* where "R" means "greater than or equal."
3. "*a is a subset* of *b*" where the relation is "is a subset of."
4. "It takes 20 microseconds of processing time to get from node *a* to node *b*." The relation is expressed by the number 20.

- A connection from node i to j does not imply a connection from node j to node i.
- If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.

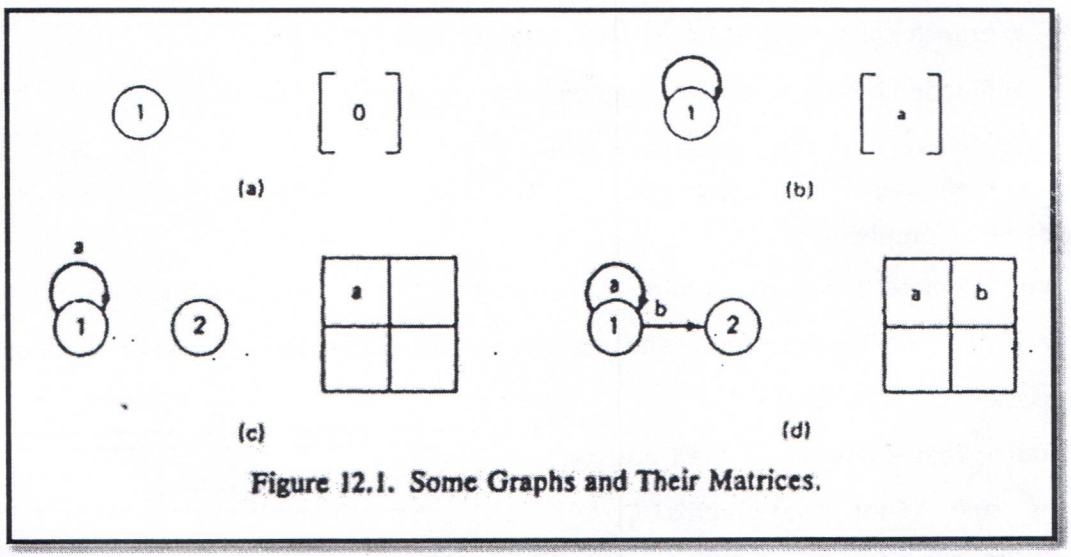
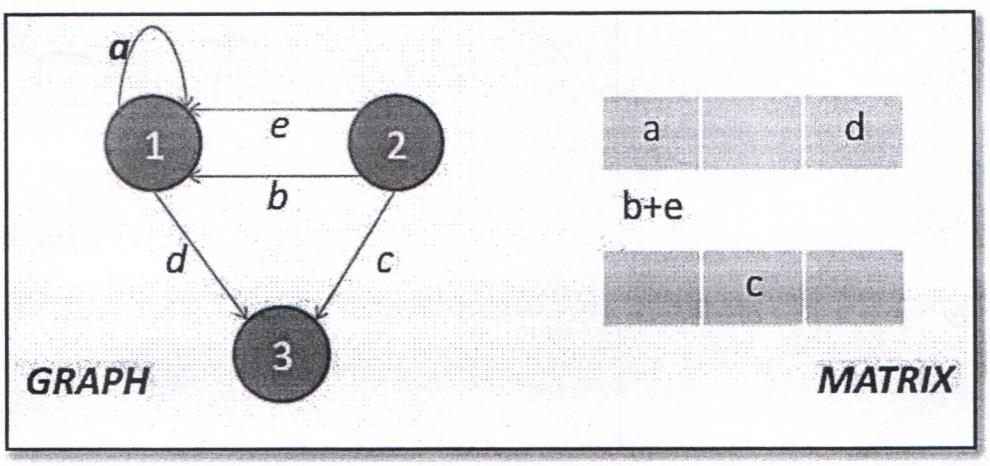


Figure 12.1. Some Graphs and Their Matrices.



1.2. A simple weight:-

A simplest weight we can use is to note that there is or **isn't a connection.** Let **"1"** mean that there **is a connection and "0"** mean that there **isn't.**

The arithmetic rules are:

1+1=1	1*1=1
1+0=1	1*0=0
0+0=0	0*0=0

A matrix defined with weights like this is called connection matrix.

GRAPH MATRICES AND APPLICATIONS

Introduction:-

- **Graph matrices** are introduced as another representation for graphs;
- One solution to this problem is to represent the graph as a matrix and to use matrix operations **equivalent to path tracing. These methods are more methodical and mechanical and don't dep**end on your ability to see a path they are more reliable.

The Basic Algorithms:-

The basic tool kit consists of:

- Matrix multiplication, which is used to get the path expression from every node to every other node.
- A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes.
- A collapsing process which gets the path expression from any node to any other node.

1. THE MATRIX OF A GRAPH:-

1.1. Basic Principles:-

A graph matrix is a square array with one row and one column for every node in the graph.

Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.

Some of the principles to be observed:-

- The size of the matrix equals the number of nodes.
- There is a place to put every possible direct connection or link between any and any other node.
- The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.

3. STATE TESTING:-

3.1. IMPACT OF BUGS:-

1. Wrong number of states.
2. Wrong transition for a given state-input combination.
3. Wrong output for a given transition.
4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
5. States or sets of states that are split to create inequivalent duplicates.
6. States or sets of states that have become dead.
7. States or sets of states that have become unreachable.

3.2. PRINICIPLES:-

- The starting point of state testing is:
  1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
  2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.
- A set of tests, then, consists of three sets of sequences:
  1. Input sequences.
  2. Corresponding transitions or next-state names.
  3. Output sequences.

3.3. LIMITATIONS AND EXTENSIONS:-

1. Simply identifying the **factors** that **contribute** to the **state**, calculating the total number of states, and comparing this number to the designer's notion catches some bugs.
2. Insisting on a justification for all supposedly **dead**, **unreachable**, and **impossible states** and **transitions** catches a few more bugs.
3. Insisting on an explicit specification of the **transition** and **output** for every combination of input and state catches many more bugs.
4. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.
5. In executing state tests, it is essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

- The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ.
- There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged.

2.2. TRANSITION BUGS:-

2.2.1. Unspecified and contradictory Transitions

Every input-state combination must have a specified transition.

If the transition is impossible, then there must be a mechanism that prevents the input from occurring in that state.

Exactly one transition must be specified for every combination of input and state.

A program can't have contradictions or ambiguities.

Ambiguities are impossible because the program will do something for every input.

Even the state does not change, by definition this is a transition to the same state.

2.2.2. Unreachable States

An unreachable state is like unreachable code. A state that no input sequence can reach.

An unreachable state is not impossible, just as unreachable code is not impossible

There may be transitions from unreachable state to other states; there usually because the state became unreachable as a result of incorrect transition.

There are two possibilities for unreachable states:

There is a bug; that is some transitions are missing.

2.2.3. Dead States

A dead state is a state that once entered cannot be left. This is not necessarily a bug but it is suspicious.

2.2.4. Output Errors

The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect.

Output actions must be verified independently of states and transitions. State Testing

In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base.

Find the number of states as follows:

1. Identify all the component factors of the state.
2. Identify all the allowable values for each factor.
3. The number of states is the product of the number of allowable values of all the factors.

2.1.2. EQUIVALENT STATES:-

- Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state.

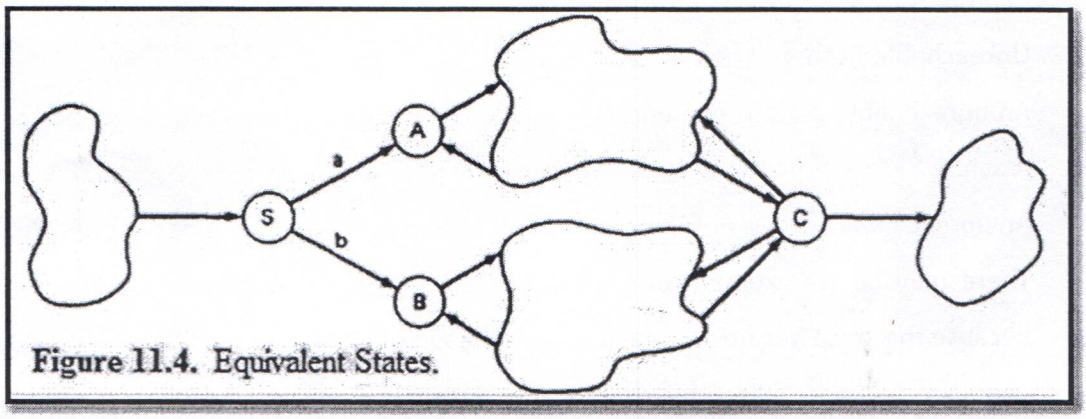


Figure 11.4. Equivalent States.

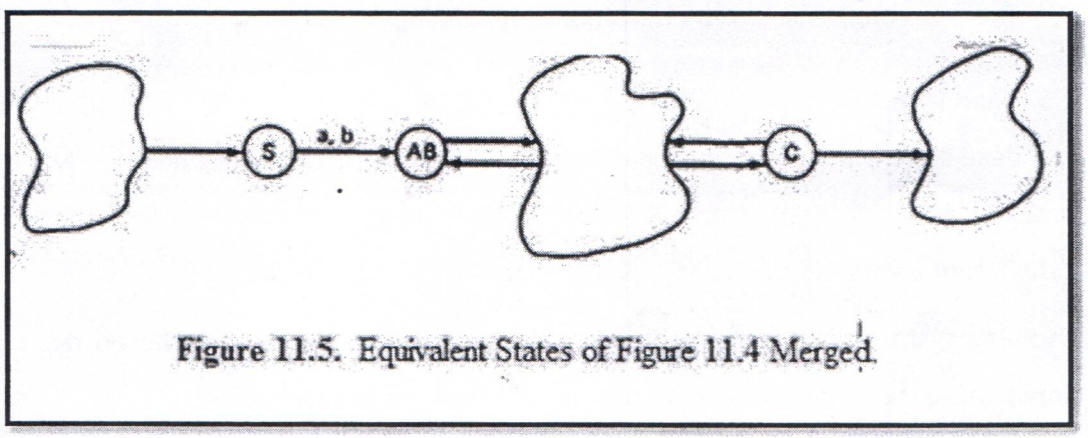


Figure 11.5. Equivalent States of Figure 11.4 Merged.

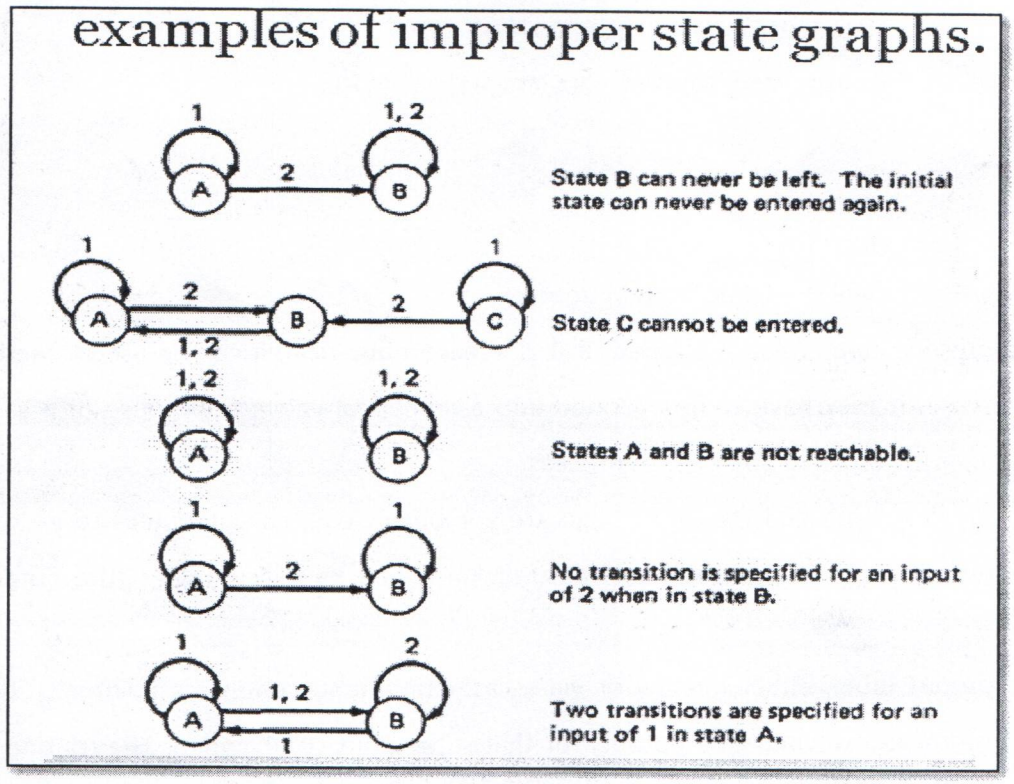
Recognizing Equivalent States

Equivalent states can be recognized by the following procedures:

2. GOOD STATE GRPAHS AND BAD STATE GRAPHS

Here are **some principles** for judging Whether state graphs are good or bad

1. The total number of states is equal to the product of the possibilities of factors that make up the state.
2. For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
3. For every transition there is one output action specified. The output could be trivial, but at least one output does something sensible.
4. For every state there is a sequence of inputs that will drive the system back to the same state.



2.1. State Bugs:-

2.1.1. Number of States:-

The number of states in a state graph is the number of states we choose to recognize or model.

- A table or case statement that specifies the output or output code, if any, associated with every state-input combination (**OUTPUT_TABLE).**
- A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport **(DEVICE_TABLE).**

The routine operates as follows, where # means concatenation:

BEGIN

- PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
- ACCEPT INPUT_VALUE
- INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
- POINTER := INPUT_CODE#PRESENT STATE
- NEW_STATE := TRANSITION_TABLE(POINTER)
- OUTPUT_CODE := OUTPUT_TABLE(POINTER)
- CALL OUTPUT_HANDLER(OUTPUT_CODE)
- DEVICE_TABLE(DEVICE_NAME) := NEW_STATE

END

a. The present state is fetched from memory.
b. The present input value is fetched. If it is already numerical, it can be used directly; otherwise, it may have to be encoded into a numerical value, say by use of a case statement, a table, or some other process.
c. The present state and the input code are combined (e.g., concatenated) to yield a pointer (row and column) of the transition table and its logical image (the output table).
d. The output table, either directly or via a case statement, contains a pointer to the routine to be executed (the output) for that state-input combination. The routine is invoked (possibly a trivial routine if no output is required).
e. The same pointer is used to fetch the new state value, which is then stored.

3. The box at the **intersection** of a **row** and a **column** specifies the **next state** (the transition) and the output.

The state table for the above figure is

	INPUT	
STATE	OKAY	ERROR
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7	...	...

1.5. Time Versus Sequence:-

Time	Sequence
1. State graphs don't represent time	1. State graphs represent sequence.
A transition might take microseconds or centuries; a system could be in one state for milliseconds and another for eons, or the other way around;	the state graph would be the same because it has no notion of time

1.6. Software Implementation:-

1.6.1. Implementation and operation:-

There are four tables involved:

- A table or process that encodes the input values into a compact list (INPUT_CODE_TABLE).
- A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).

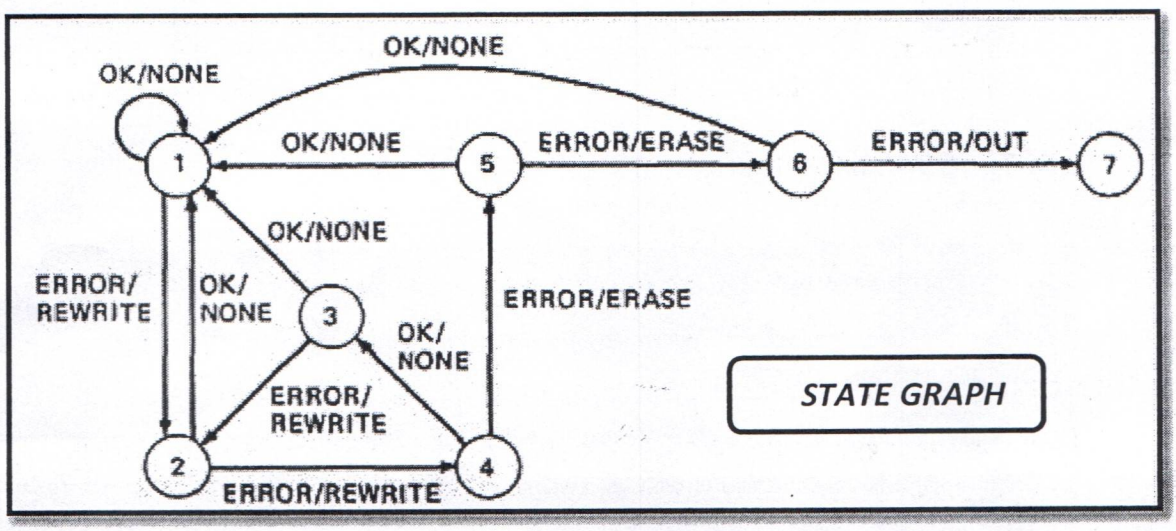
6. A **C** received in the **"ZCZ"** state completes the sequence and the system enters the **"ZCZC"** state.

7. A **Z** breaks the sequence and causes a transition back to the "Z" state; any other character causes a return to the **"NONE"** state.
8. The system stays in the **"ZCZC"** state no matter what is received.

1.3. OUTPUTS:-

An output* can be associated with any link.

Outputs are denoted by letters or words and are separated from inputs by a slash as follows: **"input/output."**



There are only **two** kinds of **inputs (OK, ERROR)** and **four** kinds of **outputs (REWRITE, ERASE, NONE, OUT-OF-SERVICE).**

1.4. STATE TABLES:-

It's more convenient to represent the state graph as a table that specifies the states, the inputs, the transitions and the outputs.

The following conventions are used

1. Each **row of the table** corresponds to a **state.**
2. Each **column corresponds** to an **input condition.**

States are represented by **Nodes**. States are **numbered** or may identified by **words** or whatever else is convenient.

1.2. INPUTS AND TRANSITIONS:-

- As a result of those inputs, the state changes, or is said to have made a **Transition**.
- Transitions are denoted by **links** that join the states.
- The input that causes the transition are marked on the link; that is, **the inputs are link weights.**
- There is **one out link** from every state for every input.
- If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: **"input1, input2, input3....."**.
- A finite-state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

The ZCZC detection example can have the following kinds of inputs:

1. Z
2. C
3. Any character other than Z or C, which we'll denote by A

The state graph of Figure 11.1 is interpreted as follows:

1. If the system is in the **"NONE"** state, any input other than a **Z** will keep it in that state.
2. If a **Z** is received, the system transitions to the **"Z"** state.
3. If the system is in the **"Z" state** and a **Z is received**, it will remain in the **"Z" state**.
4. If a **C** is received, it will go to the "**ZC**" state; if any other character is received, it will go back to the "**NONE**" state because the sequence has been broken.
5. A **Z** received in the "**ZC**" state progresses to the "**ZCZ**" state, but any other character breaks the sequence and causes a return to the "**NONE**" state.

UNIT V

STATES, STATE GRAPHS, AND TRANSITION TESTING

Introduction

- Th **STATE GRAPH** and its associated **STATE TABLE** are useful models for describing software behavoiur.
- The ***finite state machine*** is as fundamental to software engineering as Boolean algebra is to logic.
- State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.
- Finite state machines can also be implemented as table-driven software, in which case they are a powerful design option.

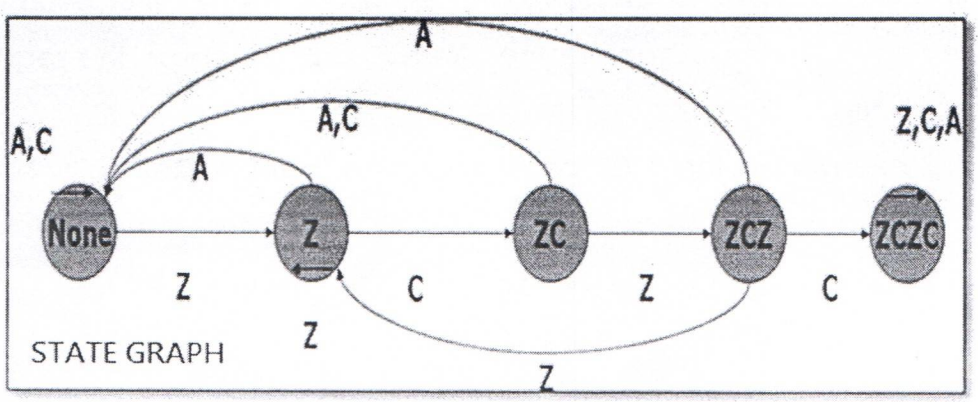
1. STATE GRAPHS:-

1.1. States:-

A state is **defined as: "A combination** of circumstances or attributes belonging for the time being to a person or **thing."**

For example, a program that detects the character sequence "ZCZC" can be in the following states.

1) Neither **ZCZC** nor any part of it has been detected.
2) Z has been detected.
3) ZC has been detected.
4) ZCZ has been detected.
5) ZCZC has been detected.



SPECIFICATIONS

1. General

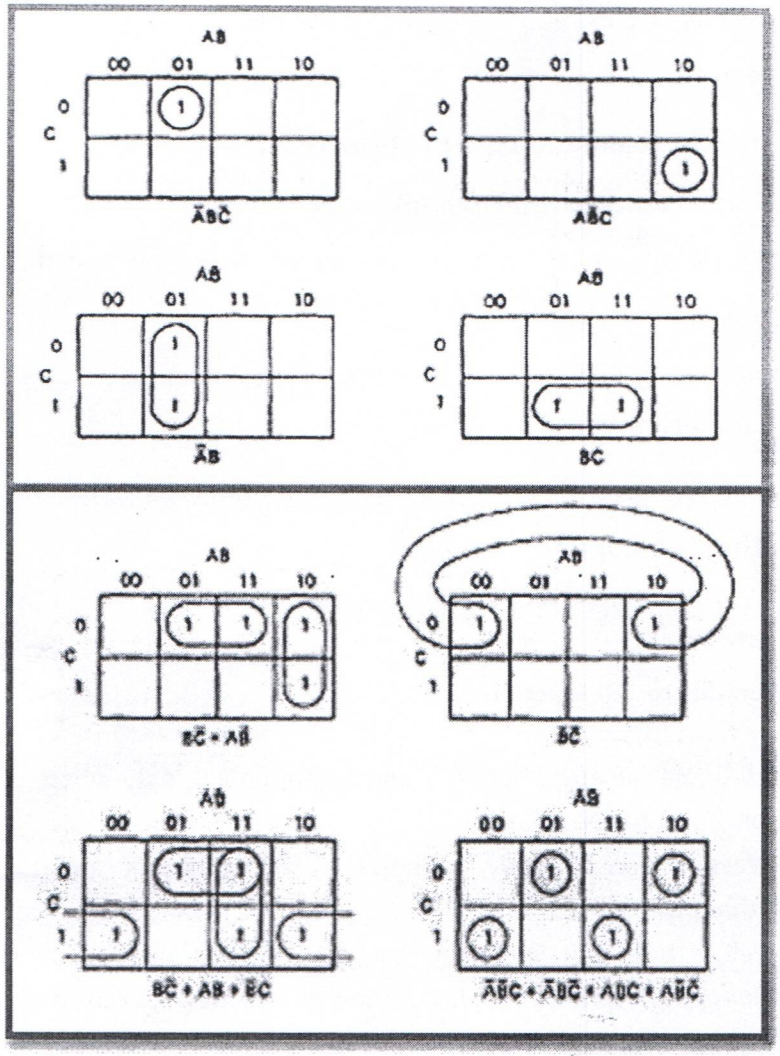
The procedure for specification validation is straightforward:

* Rewrite the specification using consistent terminology.
* Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.
* Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it might seem.
* Convert the rewritten specification into an equivalent set of Boolean expressions.

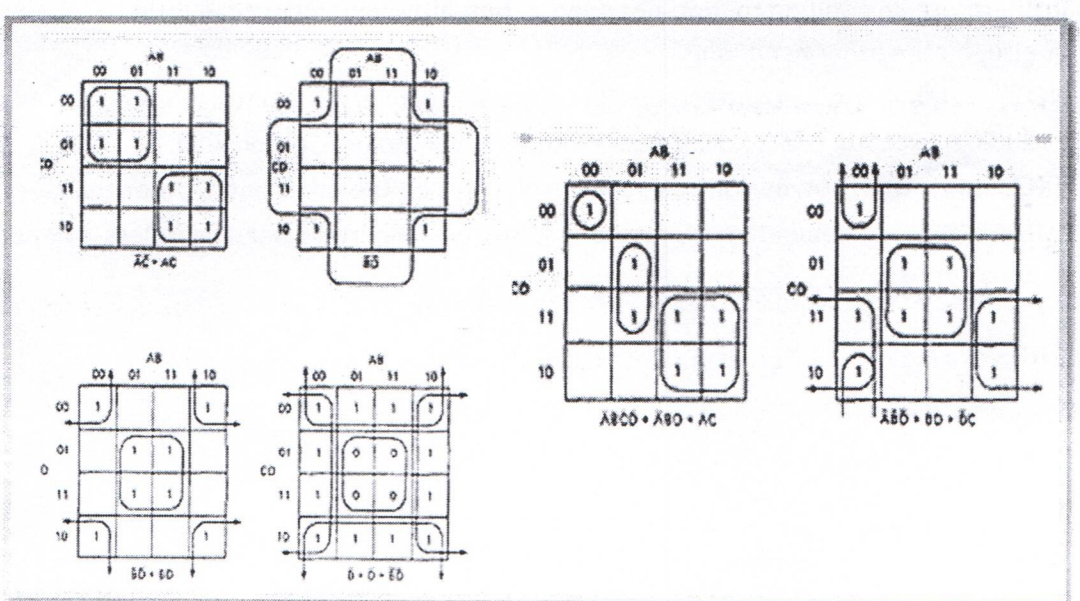
2. Finding and Translating the Logic:-

* The specifications into sentences of the following form:
    * **"IF predicate THEN action."**

* **If**—based on, based upon, because, but, if, if and when, only if, only when, provided that, when, when or if, whenever.
* **Then**—applies to, assign, consequently, do, implies that, infers that, initiate, means that, shall, should, then, will, would.
* **And**—all, and, as well as, both, but, in conjunction with, coincidental with, consisting of, comprising, either . . . or, furthermore, in addition to, including, jointly, moreover, mutually, plus, together with, total, with.
* **OR**—and, and if . . . then, and/or, alternatively, any of, anyone of, as well as, but, case, contrast, depending upon, each, either, either . . . or, except if, conversely, failing that, furthermore, in addition to, nor, not only . . . but, although, other than, otherwise, or, or else, on the other hand, plus.
* **NOT**—but, but not, by contrast, besides, contrary, conversely, contrast, except if, excluding, excepting, fail, failing, less, neither, never, no, not, other than.
* **EXCLUSIVE OR**—but, by contrast, conversely, nor, on the other hand, other than, or.
* **IMMATERIAL**—independent of, regardless, irrespective, irrelevant, regardless, but not if, whether or



Four Variables:-



The charts show all possible truth values that the variable A can have.

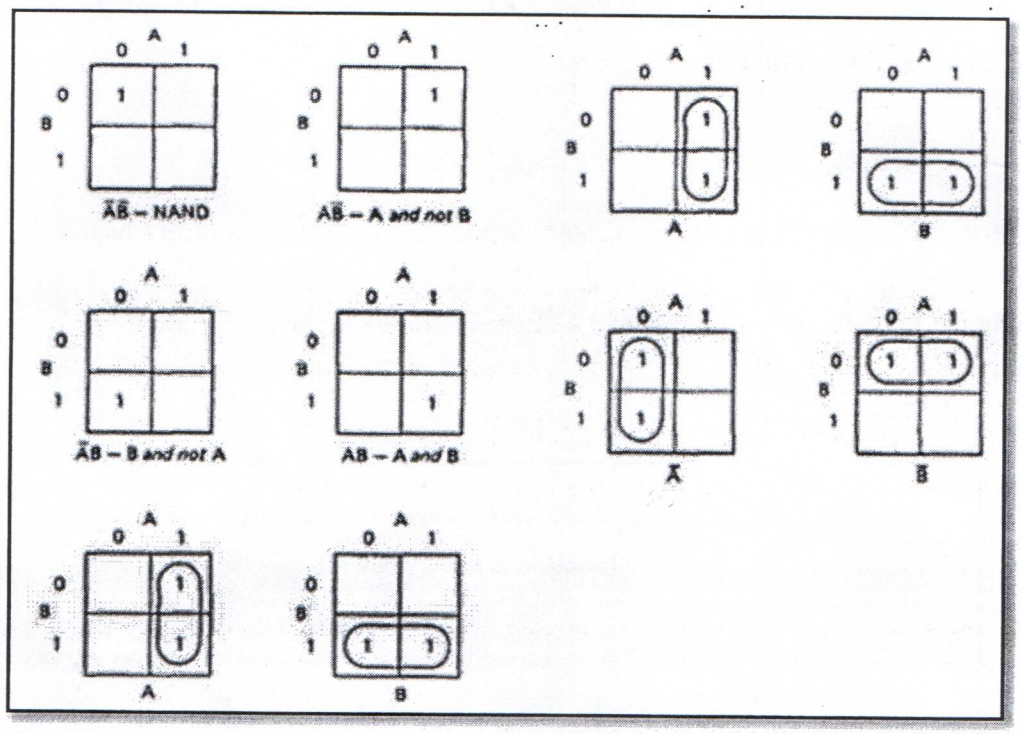
A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.

The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.

We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.

TWO VARIABLES:

Figure 6.7 shows eight of the sixteen possible functions of two variables.



THREE VARIABLES:

KV charts for three variables are shown below.

As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.

A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.

A few examples will illustrate the principles:

Test Case Design:-

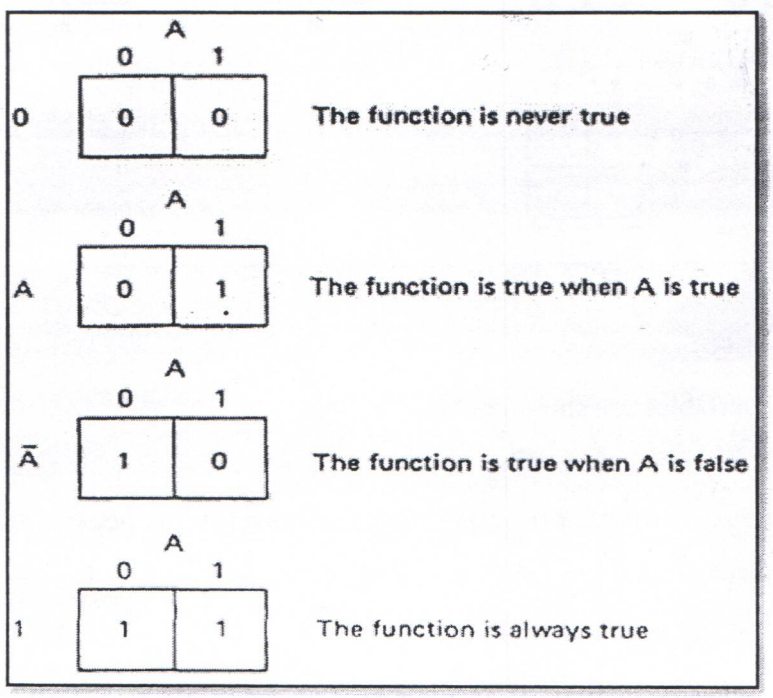
* ***Simplest***—Use any prime implicant in the expression to the point of interest as a basis for a path. The only values that must be taken are those that appear in the prime implicant.
* 2. ***Prime Implicant Cover***—Pick input values so that there is at least one path for each prime implicant at the node of interest.
* 3. *All Terms*—Test all expanded terms for that node—for example, five terms for node 6, four for node 8, and four for node 12. That is, at least one path for each term.
* 4. *Path Dependence*—Because in general, the truth value of a predicate is obtained by interpreting the predicate, its value may depend on the path taken to get there. Do every term by every path to that term.

3. KV CHARTS:

Karnaugh-Veitch chart reduces boolean algebraic manipulations to graphical trivia.

SINGLE VARIABLE:

Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.



$$
\begin{aligned}
N6 &= A + \overline{A}\overline{B}\overline{C} \\
&= A + \overline{B}\overline{C} && \text{: Use rule 19, with "B" = } \overline{B}\overline{C}. \\
N8 &= (N6)B + \overline{A}B \\
&= (A + \overline{B}\overline{C})B + \overline{A}B && \text{: Substitution.} \\
&= AB + \overline{B}\overline{C}B + \overline{A}B && \text{: Rule 16 (distributive law).} \\
&= AB + B\overline{B}C + \overline{A}B && \text{: Rule 9 (commutative multiplication).} \\
&= AB + 0C + \overline{A}B && \text{: Rule 10.} \\
&= AB + 0 + \overline{A}B && \text{: Rule 8.} \\
&= AB + \overline{A}B && \text{: Rule 3.} \\
&= (A + \overline{A})B && \text{: Rule 16 (distributive law).} \\
&= 1 \times B && \text{: Rule 5.} \\
&= B && \text{: Rules 7, 9.}
\end{aligned}
$$

$$
\begin{aligned}
N11 &= (N8)C + (N6)\overline{B}C \\
&= BC + (A + \overline{B}\overline{C})\overline{B}C && \text{: Substitution.} \\
&= BC + A\overline{B}C && \text{: Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B}A) && \text{: Rules 9, 16.} \\
&= C(B + A) && \text{: Rule 19.} \\
&= AC + BC && \text{: Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A}\overline{B}C \\
&= AC + BC + \overline{A}\overline{B}C \\
&= C(B + \overline{A}\overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\overline{C} \\
&= C + B\overline{C} + (A + \overline{B}\overline{C})\overline{B}\overline{C} \\
&= C + B\overline{C} + \overline{B}\overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}
$$

Paths and Domains:-

* Each predicate on the path is denoted by a capital letter (either overscored or not).
* Design, or specification such that there is one and only one product term for each domain: call these d_1, d_2, . . . d_i, . . . d_m. Consider any two of these product terms, d_i and d_j. For every i not equal to j, $d_i d_j$ must equal zero.

Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify the point.

$$N6 = A + \overline{A}\,\overline{B}\,\overline{C}$$
$$N8 = (N6)B + \overline{A}B = AB + \overline{A}\,\overline{B}\,\overline{C}B + \overline{A}B$$
$$N11 = (N8)C + (N6)\overline{B}C$$
$$N12 = N11 + \overline{A}\,\overline{B}C$$
$$N2 = N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C}$$

RULES OF BOOLEAN ALGEBRA:

- Boolean algebra has three operators: X (AND), + (OR) and $\overline{A}$ (NOT)
  - **X** : meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinary algebra.
  - **+** : meaning OR. "A + B" means "either A is true or B is true or both".
  - meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.

1. $A + A$	$= A$	If something is true, saying i
$\overline{A} + \overline{A}$	$= \overline{A}$	twice doesn't make it truer, ditto for falsehoods.
2. $A + 1$	$= 1$	If something is always true, ther "either A or true or both" mus also be universally true.
3. $A + 0$	$= A$	
4. $A + B$	$= B + A$	Commutative law.
5. $A + \overline{A}$	$= 1$	If either A is true or not-A is true then the statement is always true
6. AA	$= A$	
$\overline{A}\,\overline{A}$	$= \overline{A}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\overline{A}$	$= 0$	A statement can't be simulta neously true and false.
11. $\overline{\overline{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not neve going to get this nohow."?
12. $\overline{0}$	$= 1$	
13. $\overline{1}$	$= 0$	
14. $\overline{A + B}$	$= \overline{A}\,\overline{B}$	Called "De Morgan's theorem o law."
15. $\overline{AB}$	$= \overline{A} + \overline{B}$	
16. $A(B + C)$	$= AB + AC$	Distributive law.
17. $(AB)C$	$= A(BC)$	Multiplication is associative.
18. $(A + B) + C$	$= A + (B + C)$	So is addition.
19. $A + \overline{A}B$	$= A + \overline{B}$	Absorptive law.
20. $A + AB$	$= A$	

2. PATH EXPRESSIONS:

GENERAL:

- **Logic-based testing is structural testing when it's applied to structure** (e.g., control flowgraph of an implementation);
- **it's functional testing when it's applied to a specification.**
- In logic-based testing we focus on the truth values of control flow predicates.
- A **predicate** is implemented as a process whose outcome is a truth functional value.
- For our purpose, logic-based testing is restricted to binary predicates.

BOOLEAN ALGEBRA:

STEPS:

1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say $\overline{A}$)

2. The truth value of a path is the product of the individual labels.

Concatenation or products mean "AND". For example, the straight through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of.

3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

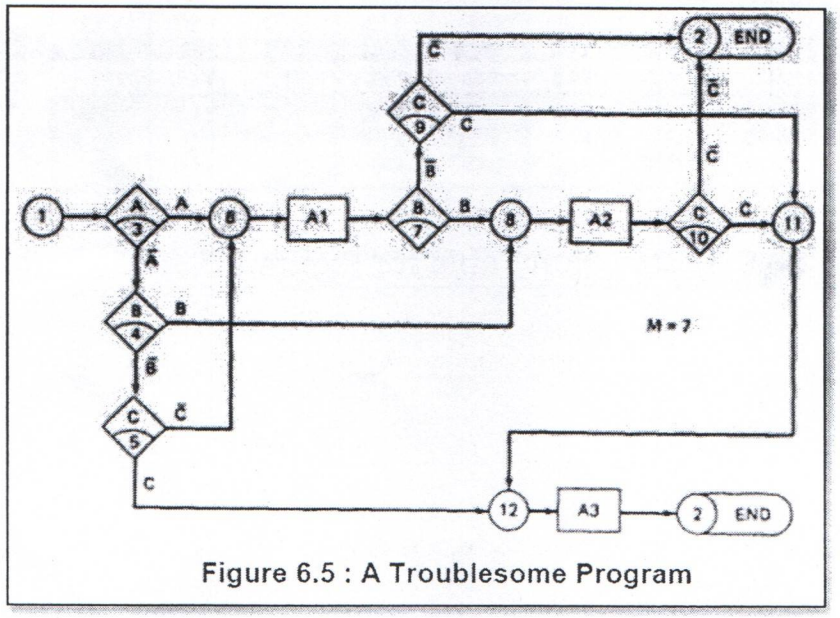


Figure 6.5 : A Troublesome Program

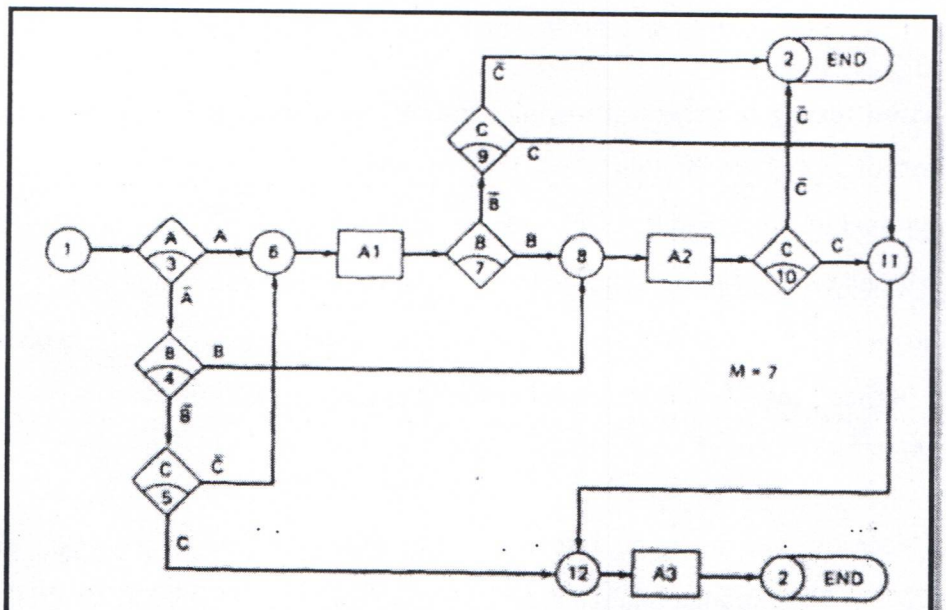


Figure 6.5 : A Troublesome Program

The programmer tried to force all three processes to be executed for the cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.

$\bar{A}\bar{B}\bar{C}$

o Table 6.3 shows the conversion of this flowgraph into a decision table after expansion.

	RULES							
	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3 : Decision Table for Figure 6.5

If the decision appears on a path, put in a **YES** or **NO** as appropriate. If the decision does not appear on the path, put in an **I,** Rule 1 does not contain decision C, therefore its entries are: YES, YES, I, YES.

The corresponding decision table is shown in Table 6.1

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A	YES	YES	YES	NO	NO	NO
CONDITION B	YES	NO	YES	I	I	I
CONDITION C	I	I	I	YES	NO	NO
CONDITION D	YES	I	NO	I	YES	NO
ACTION 1	YES	YES	NO	NO	NO	NO
ACTION 2	NO	NO	YES	YES	YES	NO
ACTION 3	NO	NO	NO	NO	NO	YES

Similarly, if we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

o Consider the following specification whose putative flowgraph is shown in Figure 6.5:

If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.

2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.

3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.

4. If none of the conditions is met, then do processes A1, A2, and A3.

5. When more than one process is done, process A1 must be done first, then A2, and then A3.

The only permissible cases are: (A1),(A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).

o Figure 6.5 shows a sample program with a bug.

Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule 2).

"Condition" is another word for predicate.

Decision-table uses "**condition**" and "**satisfied**" or "met". Let us use "**predicate**" and **TRUE / FALSE**.

Now the above translations become:

1. **Action 1** will be taken if predicates **1** and **2** are **true** and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).
2. **Action 2** will be taken if the predicates are all false, (rule 3).
3. **Action 3** will take place if predicate 1 is false and predicate 4 is true (rule4).

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

0. The specification is given as a decision table or can be easily converted into one.
1. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
2. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
3. Once a rule is satisfied and an action selected, no other rule need be examined.
4. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter

1.2. DECISION-TABLES AND STRUCTURE:

Decision tables can also be used to examine a program's structure.

Figure 6.4 shows a program segment that consists of a decision tree.

These decisions, in various combinations, can lead to actions 1, 2, or 3.

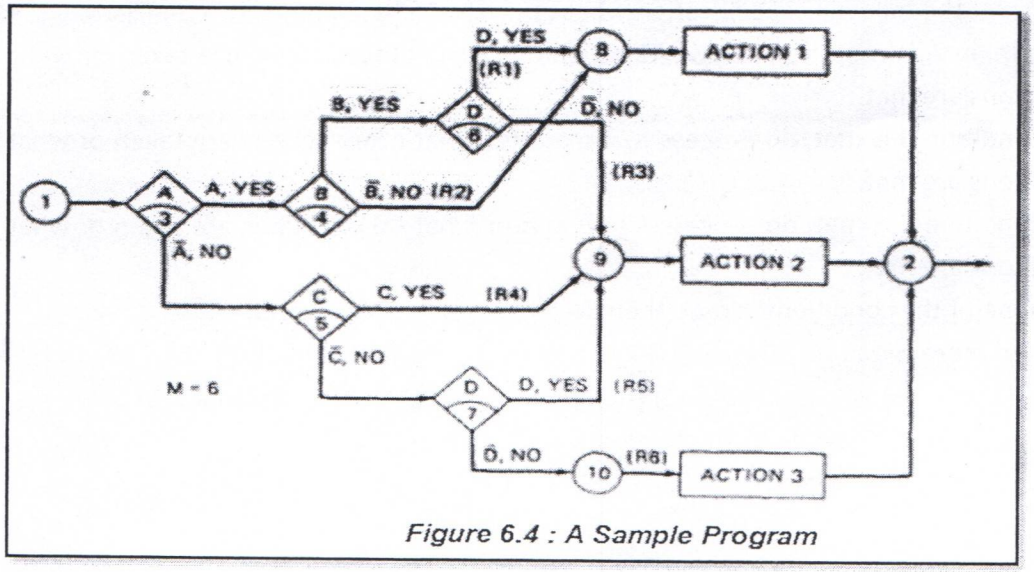
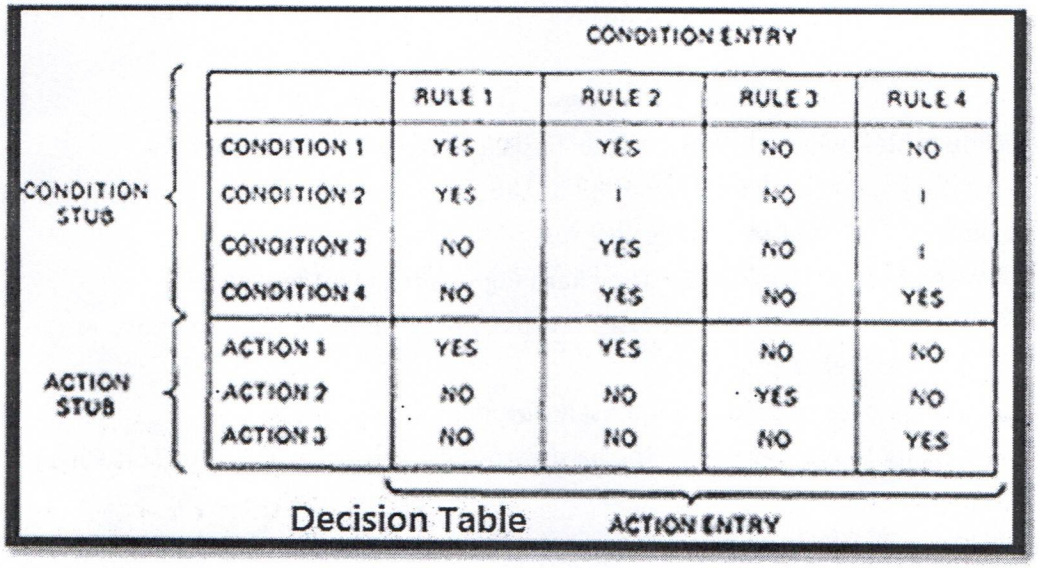


Figure 6.4 : A Sample Program

- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.
- The condition stub is a list of names of conditions.



A more general decision table can be as below:

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule. The action stub names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES", the action will take place; if "NO", the action will not take place.

The table in Figure 6.1 can be translated as follows:

LOGIC BASED TESTING

Introduction:-

* **Decision tables**, which provide a useful basis for program and test design.
* Consistency and completeness can be analyzed by using Boolean algebra, which can also be used as a basis for test design.
* Boolean algebra is trivialized by using **karnaugh-veitch charts**.
* "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
* Hardware logic *test* design, are intensely automated.
* Many test methods developed for hardware logic can be adapted to software logic testing.
* Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.

Knowledge-based system:-

* The **knowledge-based system** (also **expert system** or **"artificial intelligence"** system has become the programming construct of choice for many applications that were once considered very difficult (WATE86).
* Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database.
* The processing is done by a program called the **inference engine.**
* Decision tables are extensively used in business data processing;

1 DECISION TABLES:

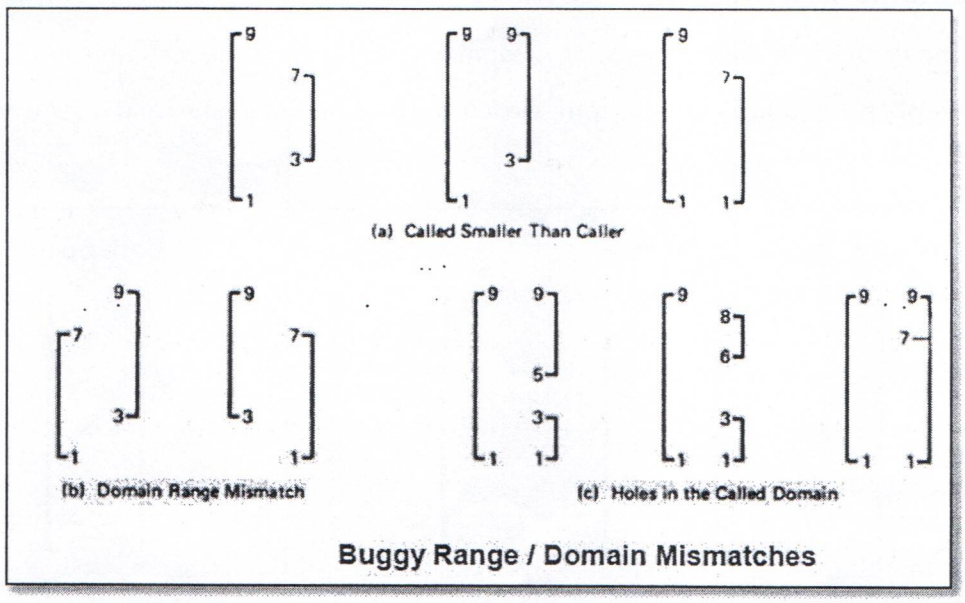
1.1. Definitions and Notation:-

Table 10.1 is a **limited-entry decision table**. It consists of four areas called

- The **condition stub**,
- The **condition entry**,
- The **action stub**, and
- The **action entry**.

The routine is used by many callers; some require values inside a range and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for the caller.

Figure (a) shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples are buggy.



Buggy Range / Domain Mismatches

In **Figure (b) the ranges and domains don't line up**; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes.

Figure(c) combines these notions to show various ways we can have **holes in the domain**: these are all probably buggy.

5.4. INTERFACE RANGE / DOMAIN COMPATIBILITY TESTING:

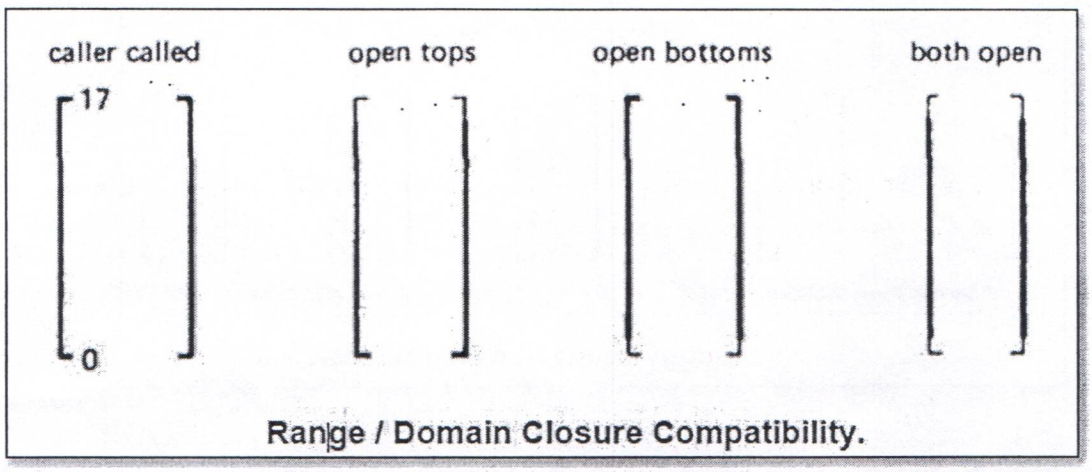
- For interface testing, bugs are more likely to **concern single variables** rather than peculiar combinations of two or more variables.
- **Test every input variable independently** of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable.
- There are **two boundaries to test** and it's a one-dimensional domain; therefore, it requires **one on** and **one off point per boundary** or a total of **two on points** and **two off points** for the **domain** - pick the off points appropriate to the closure (COOOOI).
- Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in component testing.

5.2. CLOSURE COMPATIBILITY:

Assume that the caller's range and the called domain spans the same numbers - for example, ***0 to 17.***

Figure below shows the **four ways** in which the caller's range closure and the called domain closure can agree. The ***thick line*** means ***closed*** and the ***thin line*** means ***open***.

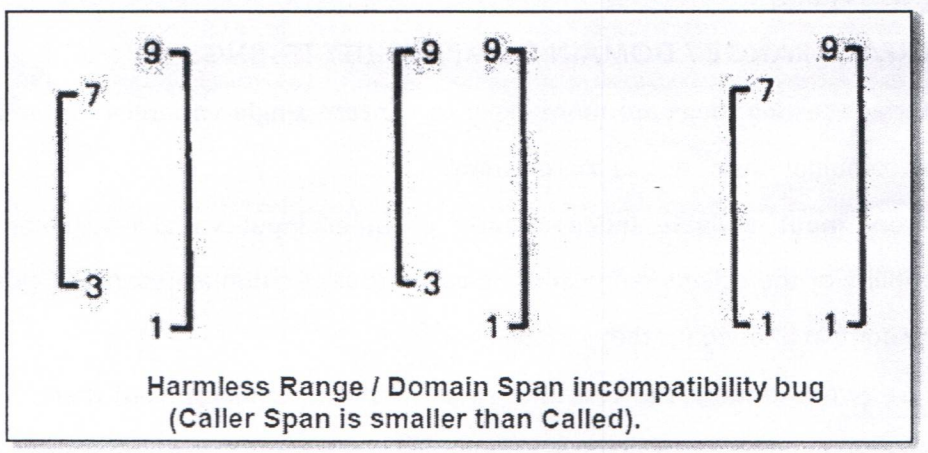
Figure below shows the **four cases** consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom.



Range / Domain Closure Compatibility.

5.3. SPAN COMPATIBILITY:

Figure below shows three possibly harmless span incompatibilities



Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

In all cases, the caller's range is a subset of the called's domain.

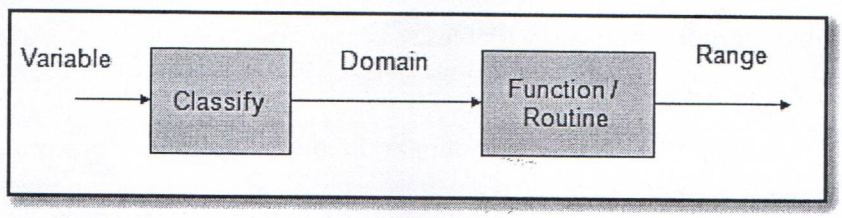
5. DOMAIN AND INTERFACE TESTING

INTRODUCTION:

- Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components.
- Interface between any two components is considered as a subroutine call.
- For a single variable, the domain span is the set of numbers between (and including) the smallest value and the largest value.
- For every input variable we want (at least): compatible domain spans and compatible closures (Compatible but need not be Equal).

5.1. DOMAINS AND RANGE:

- The ***set of output values*** produced by a function is called the **range** of the function, in contrast with the **domain**, which is the ***set of input values*** over which the function is defined.



- For most testing, our aim has been to specify input values and to predict and/or confirm output values that result from those inputs.
- Interface testing requires that we select the output values of the calling routine ***i.e.*** caller's range must be compatible with the called routine's domain.
- An interface test consists of exploring the correctness of the following mappings:

  - **caller domain --> caller range (caller unit test)**
  - **caller range --> called domain (integration test)**
  - **called domain --> called range (called unit test)**

Figure(d) has a tilted boundary, which creates wrong domain segments A' and B'. In this example the bug is caught by the left on point.

4) **Extra Boundary:** An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug.

The extra boundary in Figure (e) is caught by two on points, and depending on which way the extra boundary goes, possibly by the off point also.

5) **Missing Boundary:** A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and will cause many test failures although there is only one bug.

A missing boundary, shown in Figure (f), is caught by the two on points because the processing for A and B is the same - either A or B processing.

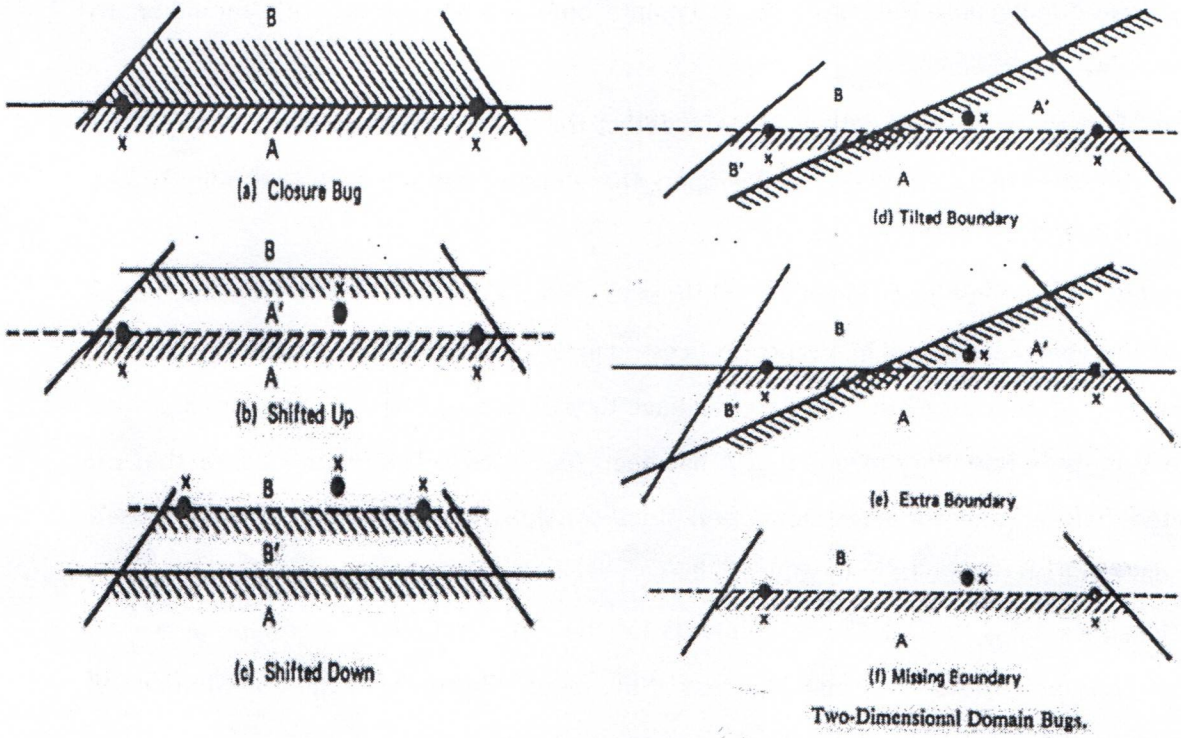
4.5. PROCEDURE FOR TESTING: The procedure is conceptually is straight forward. It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.

1. Identify input variables
2. Identify variables which appear in domain-defining predicates, such as control-flow predicates
3. Interpret all domain predicates in terms of input variables:
    - Transform non-linear to linear
    - Find data flow path
4. Predicate expression with p # predicates.

   Find # domains: $< 2^p$
5. Solve inequalities for extreme points
6. Use extreme points to solve for nearby on points

4.4. TESTING TWO DIMENSIONAL DOMAINS:

- Figure below shows possible domain boundary bugs for a two-dimensional domain.
- A and B are adjacent domains and the boundary is closed with respect to A, which means that it is open with respect to B.



Two-Dimensional Domain Bugs.

For Closed Boundaries:

1) **Closure Bug:** Figure (a) shows a faulty closure, such as might be caused by using a wrong operator (for example, x >= k when x > k was intended, or vice versa). **The two on points detect this bug because those values will get B rather than A processing.**

2) **Shifted Boundary:** In Figure (b) the bug is a shift up**, which converts part of domain B into A processing, denoted by A'.** This result is caused by an incorrect constant in a predicate, such as x + y >= 17 when x + y >= 7 was intended. The off point (closed off outside) catches this bug.

3) **Tilted Boundary:** A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, 3x + 7y > 17 when 7x + 3y > 17 was intended.

In **Figure (a)** we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts > to >= or < to <= (Figure 4.13b).

One test (marked x) on the boundary point detects this bug because processing for that point will go to domain A rather than B.

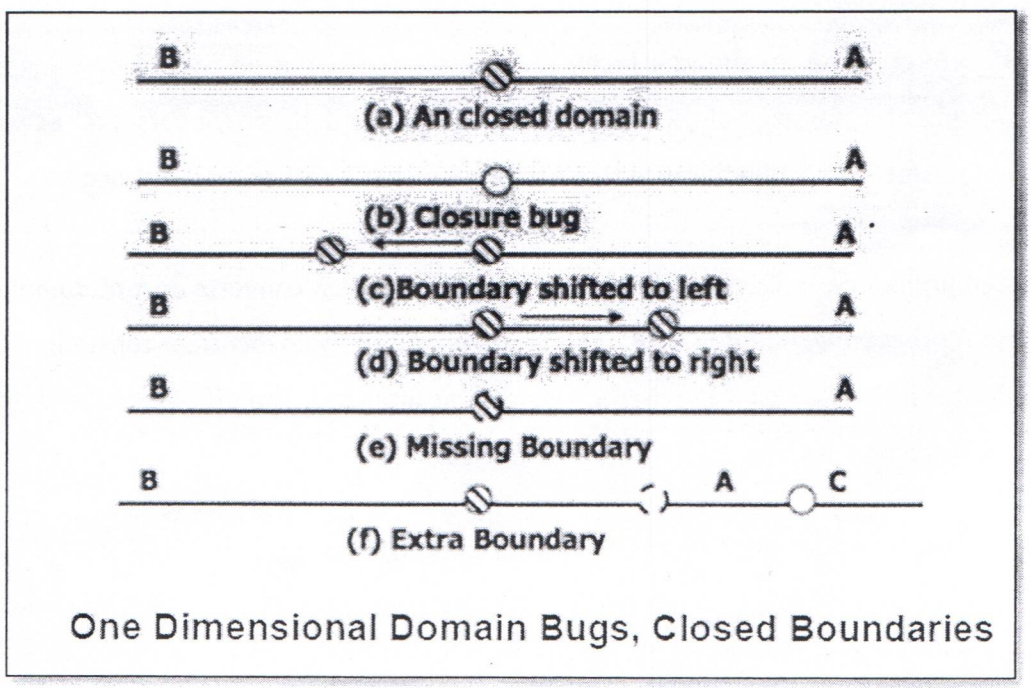
In **Figure(c)** we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing.

Figure (d) shows a shift the other way. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition; the off point is in A (Open Off Inside).

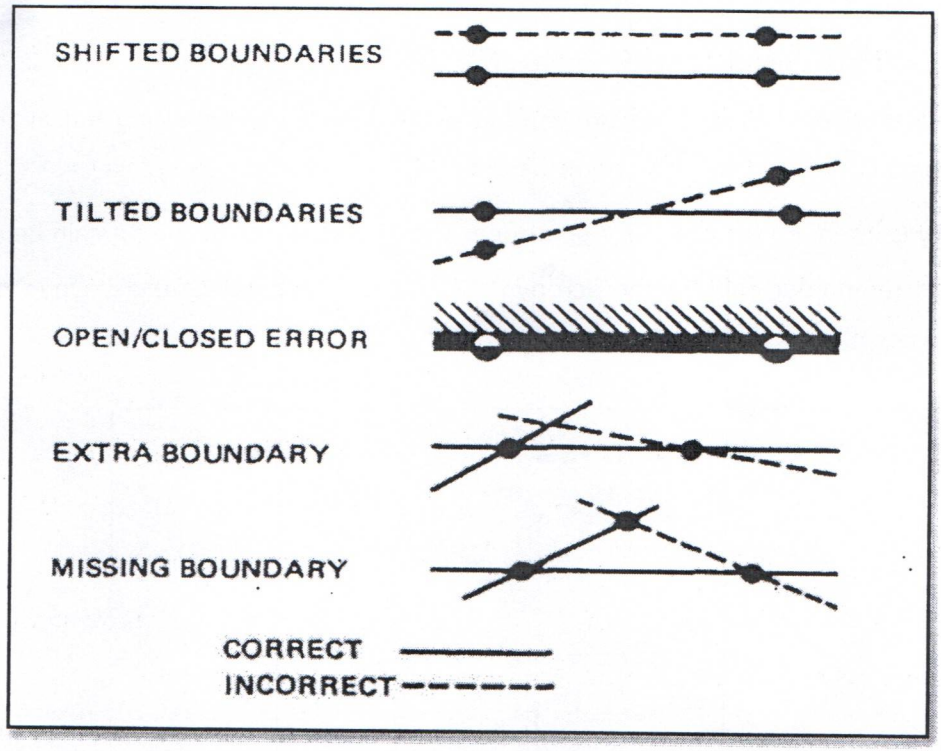
Figure (e) The same open off point also suffices to detect a missing boundary because what should have been processed in A is now processed in B.

Figure (f) To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would do it.

For closed domains look at **Figure below** as for the **open boundary**, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside the domain.

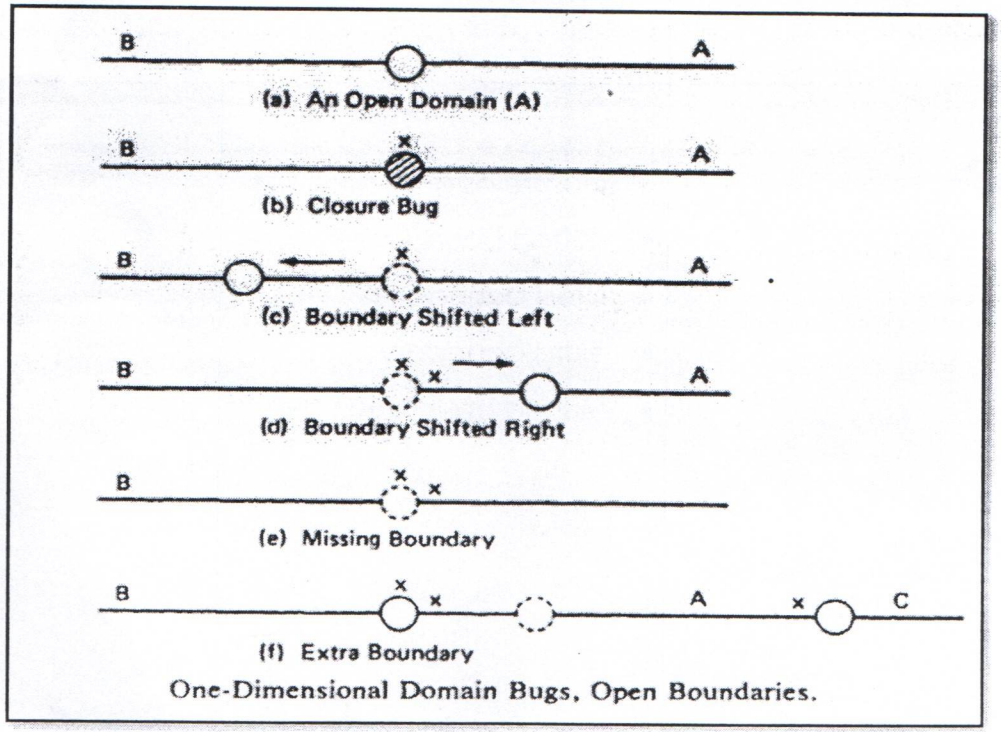


One Dimensional Domain Bugs, Closed Boundaries



4.3. TESTING ONE DIMENSIONAL DOMAINS:

- Below Figure shows possible domain bugs for a **one-dimensional open domain boundary.**
- The closure can be wrong or the boundary can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

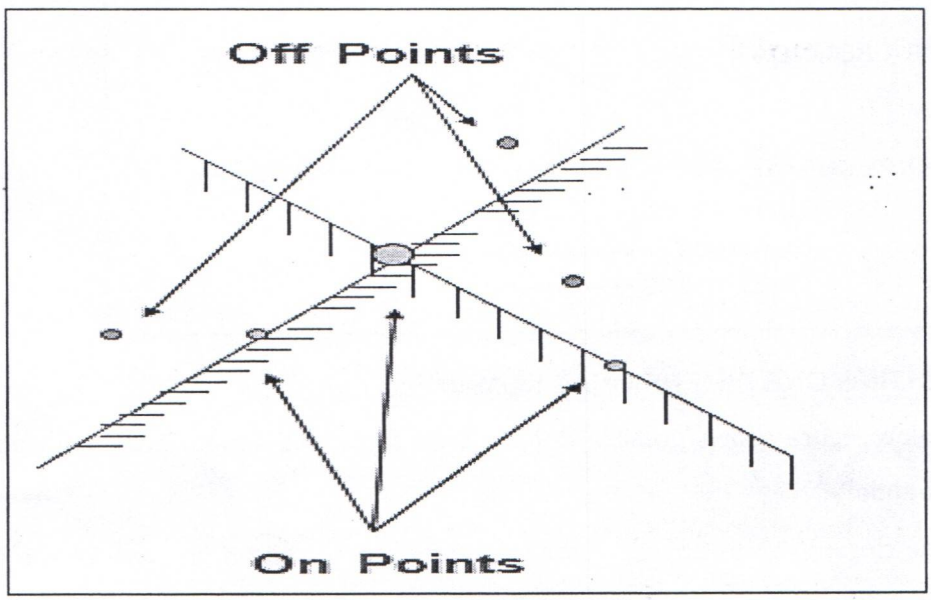


One-Dimensional Domain Bugs, Open Boundaries.

An **on point** is a point on the boundary.

If the domain **boundary is closed,** an **off point** is a point near the boundary but in the **adjacent domain**.

If the **boundary is open**, an **off point** is a point near the boundary but in the domain being tested; You can remember this by the acronym COOOOI: Closed off Outside, Open off Inside.



Below Figure shows ***generic domain bugs***:

- closure bug
- shifted boundaries
- tilted boundaries
- extra boundary
- Missing boundary.

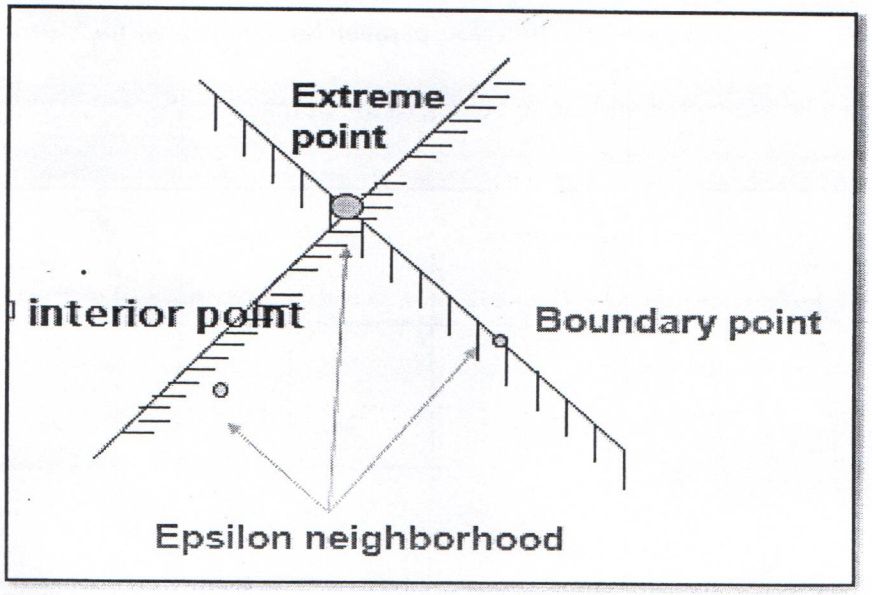
4. DOMAIN TESTING:

4.1. DOMAIN TESTING STRATEGY: The domain-testing strategy is simple, although possibly tedious (slow).

- ***Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.***
- Define test strategy for each possible bug related to boundary.
- Test points for a domain useful to test its adjacent domain.
- Run the tests and by post test analysis (the tedious part) determine if any boundaries are faulty and if so, how.
- Run enough tests to verify every boundary of every domain.

4.2. DOMAIN BUGS AND HOW TO TEST FOR THEM:

- An **interior point** is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.
- A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in the domain.
- An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain.



Programmers introduce bugs and testers misdesign test cases by: **smoothing out concavities (Figure 4.8a), filling in holes (Figure 4.8b),** and **joining disconnected pieces (Figure 4.8c).**

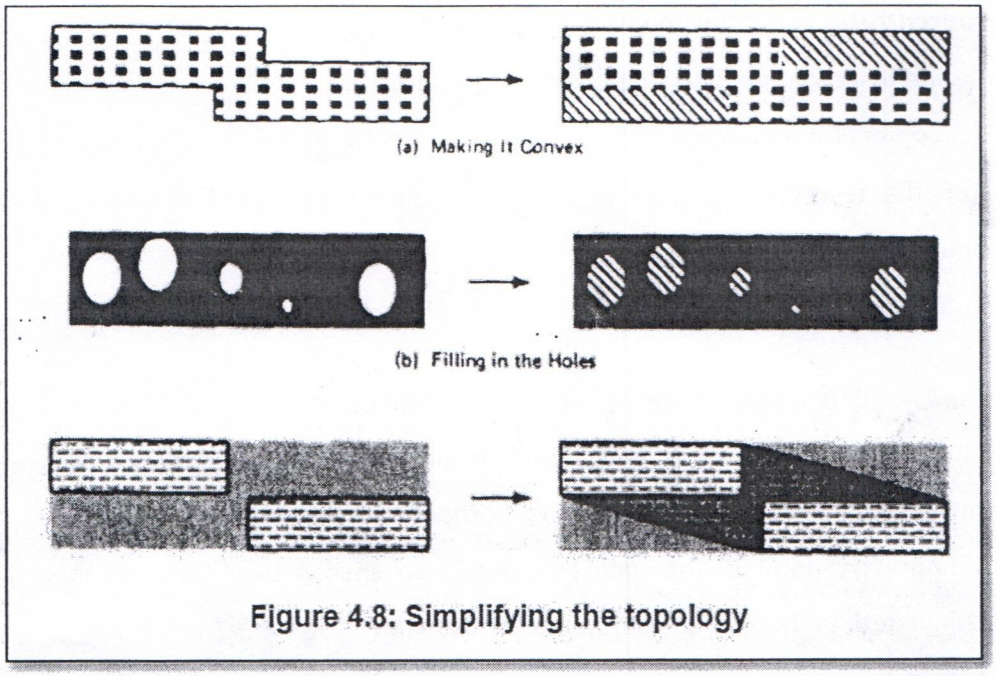
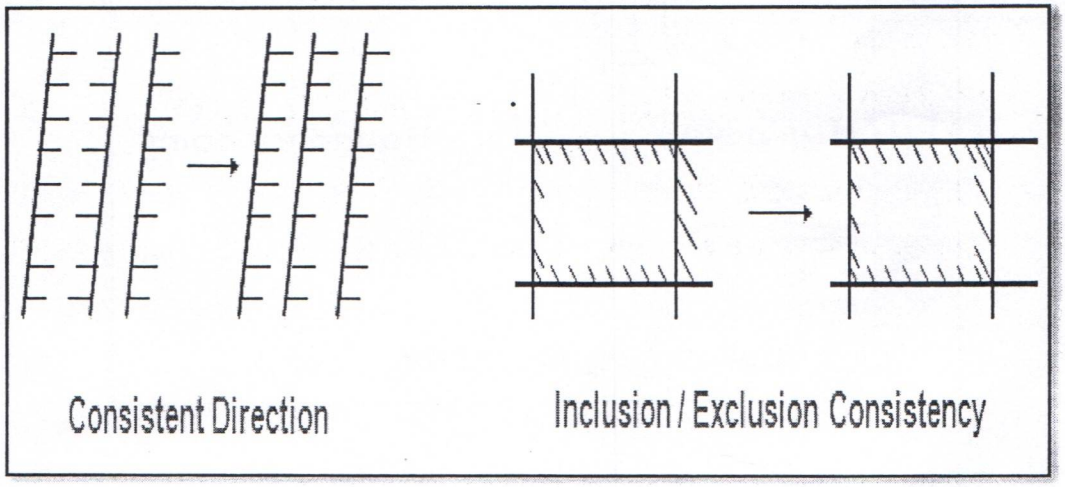


Figure 4.8: Simplifying the topology

3.3. RECTIFYING BOUNDARY CLOSURES:

- Make closures in one direction for parallel boundaries with closures in both directions
- Force a Bounding Hyperplane to belong to the Domain.

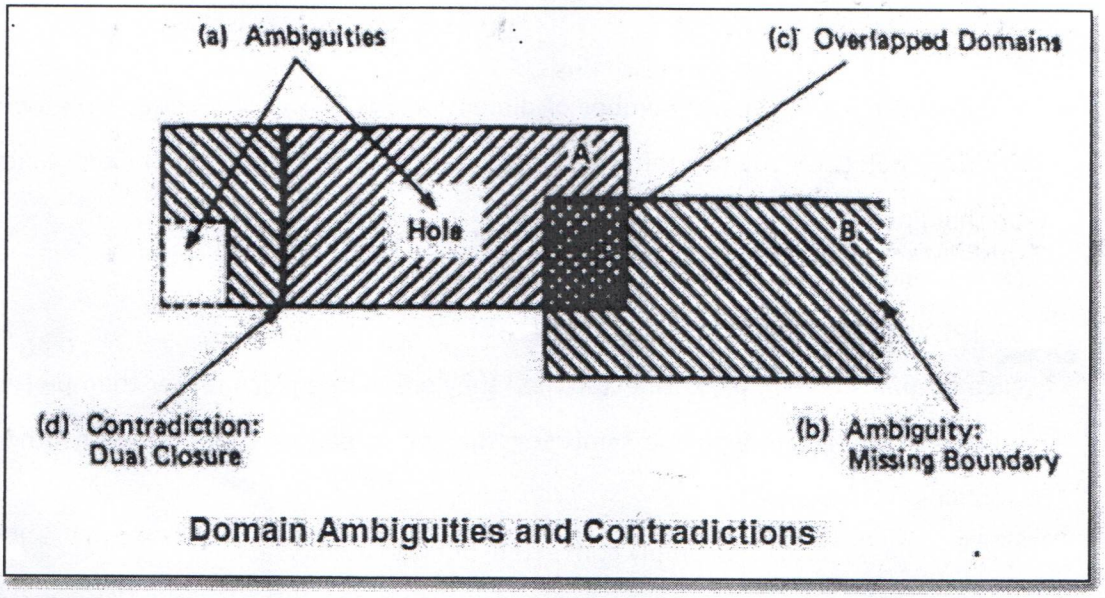


3. UGLY DOMAINS:

- Some domains are born ugly and some are uglified by bad specifications.
- Every simplification of ugly domains by programmers can be either good or bad.
- If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.

3.1. AMBIGUITIES AND CONTRADICTIONS:

- **Domain ambiguities are holes in the input space.**
- The holes may lie within the domains or in cracks between domains



Domain Ambiguities and Contradictions

- Two kinds of contradictions are possible: **overlapped domain specifications** and **overlapped closure specifications**
- **Fig(c)** shows overlapped domains and **Fig (d)** shows dual closure assignment.

3.2. SIMPLIFYING THE TOPOLOGY:

The programmer's and tester's reaction to complex domains is the same - simplify

There are **three** generic cases:

- Concavities
- Holes
- Disconnected pieces.

2.6. CLOSURE CONSISTENCY:

- Figure shows another desirable domain property: boundary closures are consistent and systematic.
- The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies - e.g., the boundary lines belong to the domains on the right.
- Consistent closure means that there is a simple pattern to the closures - for example, using the same relational operator for all boundaries of a set of parallel boundaries.

2.7. CONVEX:

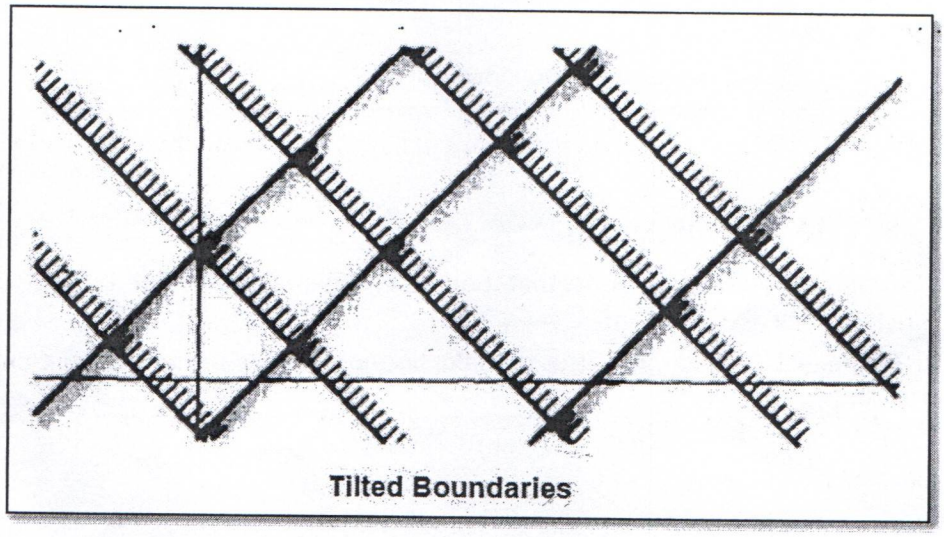
- A geometric figure (in any number of dimensions) is convex if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within the figure.
- Nice domains are convex; dirty domains aren't.

2.8. SIMPLY CONNECTED:

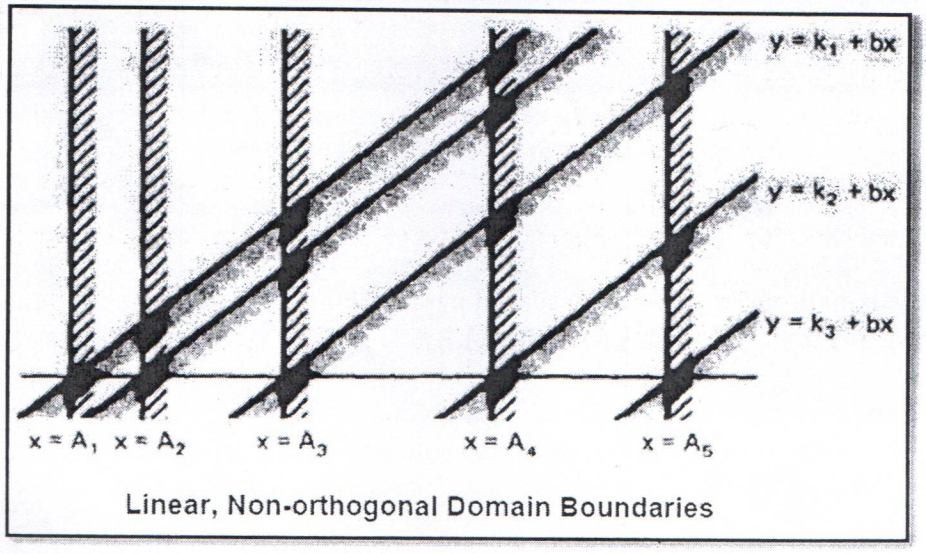
- **Nice domains are simply connected. i.e.** they are in one piece rather than pieces all over the place intersperse (=mix together or spread or combine) with other domains
- if a domain is convex it is simply connected, but not vice versa.
- Consider domain boundaries defined by a compound predicate of the (Boolean) form ABC. Say that the input space is divided into two domains, one defined by ABC and, therefore, the other defined by its negation.
- For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than 17.
- Simple connectivity, especially for default cases, may be impossible.

2.5. ORTHOGONAL BOUNDARIES:

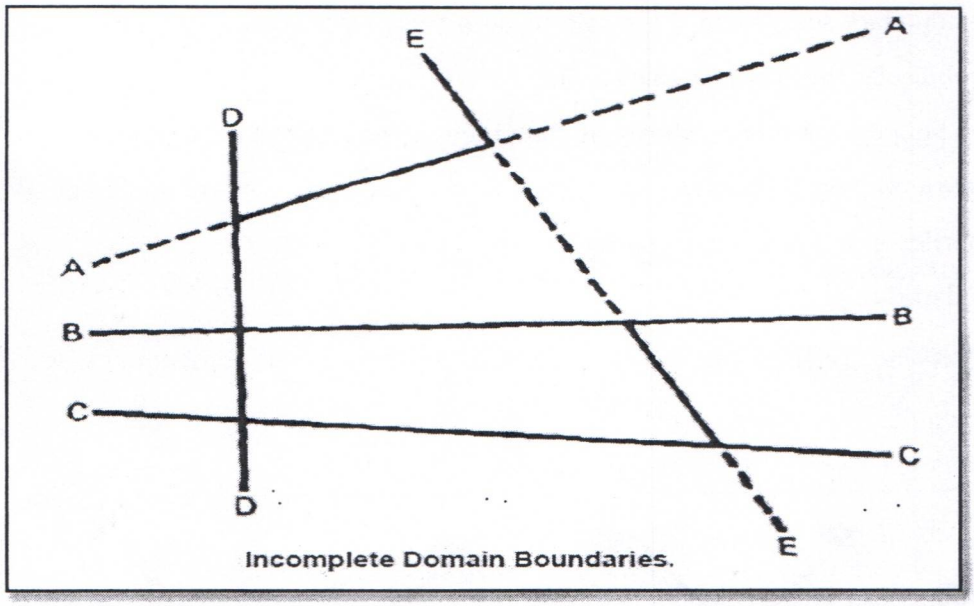
- Two boundary sets U and V are said to be orthogonal if every inequality in V is perpendicular to every inequality in U.
- If two boundary sets are orthogonal, then they can be tested independently
- In Figure we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6 + 4 = 10 (O(n)). If we tilt the boundaries to get Figure 4.5,
- we must now test the intersections. We've gone from a linear number of cases to a quadratic: from O(n) to $O(n^2)$.



Tilted Boundaries



Linear, Non-orthogonal Domain Boundaries



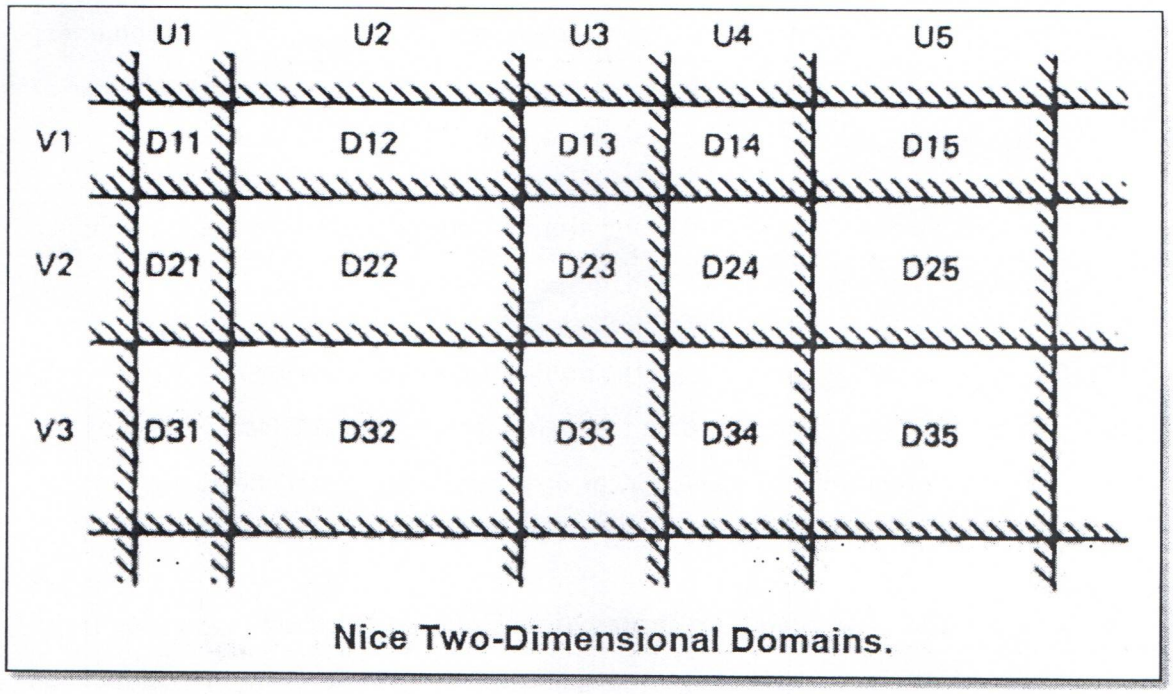
Incomplete Domain Boundaries.

2.4. SYSTEMATIC BOUNDARIES:

- Systematic boundary means that boundary inequalities related by a simple function such as a constant.
- In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant.

$$f_1(X) >= k_1 \text{ or } f_1(X) >= g(1,c)$$
$$f_1(X) >= k_2 \quad f_2(X) >= g(2,c)$$
$$\ldots\ldots\ldots\ldots \quad \ldots\ldots\ldots\ldots$$
$$f_i(X) >= k_i \quad f_i(X) >= g(i,c)$$

- Where **fi** is an arbitrary linear function, **X** is the input vector, **ki** and **c** are constants, and **g(i,c)** is a decent function over i and c that yields a constant, such as k + ic.
- The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines (such as the spokes of a wheel, if equally spaced in angles, systematic).
- If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.



Nice Two-Dimensional Domains.

2.2. LINEAR AND NON LINEAR BOUNDARIES:

- **Nice domain boundaries are defined by linear inequalities or equations.**
- The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general n+ 1 point to determine an n-dimensional hyper plane.

2.3. COMPLETE BOUNDARIES:

- **Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.**
- The Figure shows some incomplete boundaries. Boundaries **A** and **E** have gaps.
- The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds.
- If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

5) **Over specified Domains:** The domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.

6) **Boundary Errors:** Errors caused in and around the boundary of a domain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.

7) **Closure Reversal:** A common bug. The predicate is defined in terms of >=. The programmer chooses to implement the logical complement and incorrectly uses <= for the new predicate; i.e., x >= 0 is incorrectly negated as x <= 0, thereby shifting Boundary values to adjacent domains.

8) **Faulty Logic:** Compound predicates (especially) are subject to faulty logic transformations and improper simplification. Simplification of compound predicates.

2. NICE AND UGLY DOMAINS:

2.1. NICE DOMAINS:

Where do these domains come from?

- Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction.
- Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent.
- Conversely, specified domains can be incomplete and/or inconsistent.
- Incomplete means that there are input vectors for which no path is specified and inconsistent means that there are at least two contradictory specifications over the same segment of the input space.

Some important properties of nice domains are: **Linear, Complete, Systematic, And Orthogonal, Consistently closed, Convex and simply connected.**

The bug frequency is lesser for nice domain than for ugly domains.

1.5. DOMAIN DIMENSIONALITY:

- **Every input variable adds one dimension to the domain.**
- **One variable** defines domains on a **number line.**
- **Two variables** define **planar domains.**
- **Three variables** define **solid domains.**
- Every new predicate slices through previously defined domains and cuts them in half.
- Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space.
- Thus, planes are cut by ***lines*** and ***points***, ***volumes*** by ***planes, lines*** and ***points*** and n-***spaces*** by hyperplanes.

1.6. BUG ASSUMPTION:

- Processing is OK. Domain definition may be wrong.

  ⇒ **Boundaries are wrong.**

  ⇒ **Predicates are wrong.**

- Once input vector is set on the right path, it's correctly processed.

Many different bugs can result in domain errors. Some of them are:

Domain Errors:

1) **Double Zero Representation:** distinct positive and negative zero, boundary errors for negative zero are common.

2) **Floating point zero check:** A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself or multiplied by zero.

3) **Contradictory domains:** A contradictory domain specification means that at least two supposedly distinct domains overlap.

4) **Ambiguous domains:** Ambiguous domains means that union of the domains is incomplete. That is there are missing domains or holes in the specified domains.

1.4. A DOMAIN CLOSURE:

- A domain boundary is ***closed*** with respect to a domain if the points on the boundary belong to the domain.
- If the boundary points belong to **some other domain**, the boundary is said to be *open*.

Figure 4.2 shows three situations for a one-dimensional domain - i.e., a domain defined over one input variable; call it x

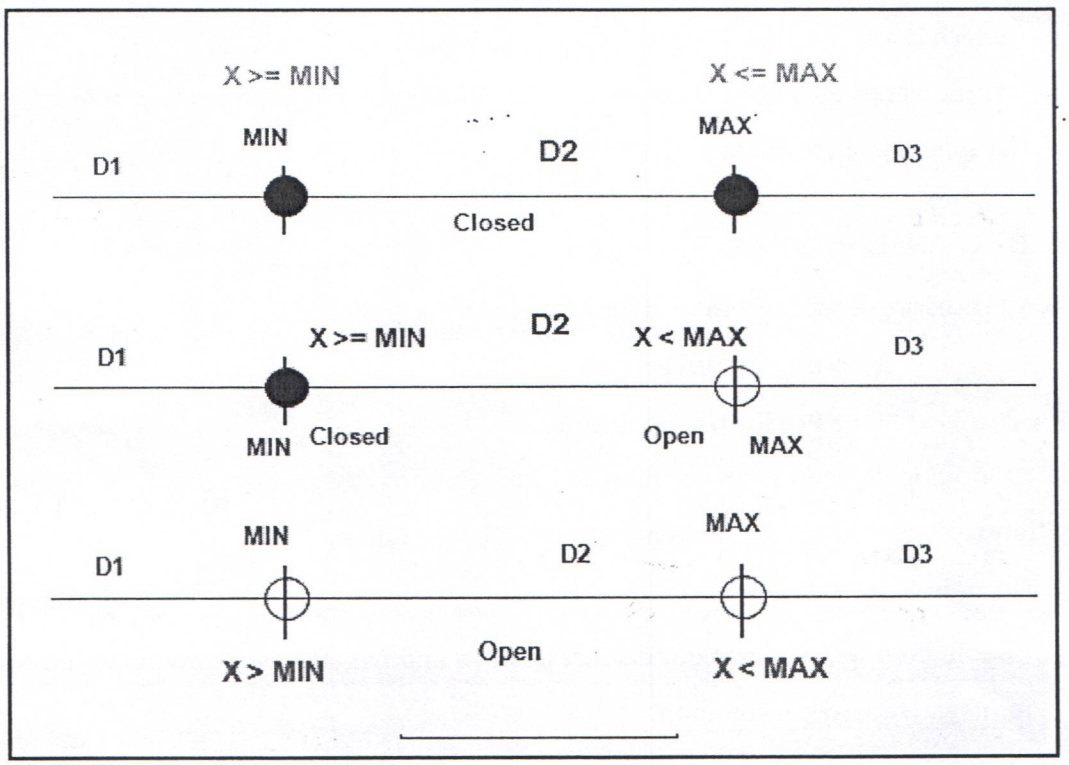


Figure 4.2: Open and Closed Domains

→ Three domains **D1, D2, D3**

a) **D2** boundaries are closed both at the minimum and maximum values. I.e. both points belong to **D2.**

b) **D2** is closed on the minimum side and open on the maximum side. i.e. **D1** is open at the minimum of **D2** and **D3** is closed at **D2** maximum.

c) **D2** open on both sides i.e. those boundaries are closed for **D1** and **D2**

➢ We can infer that for each case there must be atleast one path to process that case.

1.2. A DOMAIN IS A SET:

- **An input domain is a set.**
- Domain testing does not work well with arbitrary discrete sets of data objects.
- Domain for a loop-free program corresponds to a set of numbers defined over the input vector.

1.3. DOMAINS, PATHS AND PREDICATES:

- In domain testing, predicates are assumed to be interpreted in terms of input vector variables.
- If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine - that is, based on the implementation control flow graph.
- Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flow graph for the routine;
- For every domain, ***there is at least one path*** through the routine.
- There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or more domains.
- Domains are defined by their ***boundaries***. Domain boundaries are also where most domain bugs occur.
- For every boundary there is ***at least one predicate*** that specifies what numbers belong to the domain and what numbers don't.
- A domain may have one or more boundaries - no matter how many variables define it.
- Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.

Ex: - In the statement IF **x>0** THEN ALPHA ELSE BETA we know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s).

DOMAIN TESTING

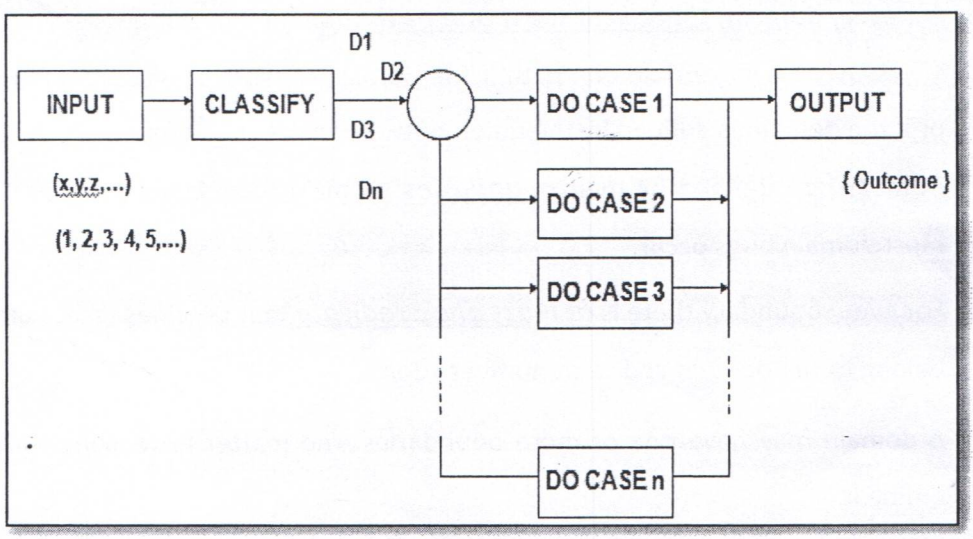
1. DOMAINS AND PATHS:

INTRODUCTION:

Domain: *A domain is a set of possible values of an independent variable or the variables of a function.*

- Domain testing attempts to determine whether the ***classification*** is or is ***not correct***.
- Domain testing can be based on ***specifications*** or ***equivalent implementation information.***
- If domain testing is based on specifications, it is a ***functional test technique***.
- If domain testing is based implementation details, it is a ***structural test technique***.

1.1. THE MODEL: The following figure is a schematic representation of domain testing.



- In domain testing, we focus on the classification aspect of the routine rather than on the calculations.
- Structural knowledge is not needed for this model - only a consistent, complete specification of input values for each case.

Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

SLICING AND DICING:

SLICING:-

- *A static program **slice is a part of a program** defined wrt a variable **'v'** and a statement **'s'**; It is the set of all statements that could affect the value of **'v'** at stmt **'s'**.*

```
Stmt1    var v
stmt2
Stmt3    var v
Stmt4    var v
Stmt s   var v
```

If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i

DICING:-

- *A program **dice is a part of a slice** in which all statements which are known to be correct have been removed.*
- A dice is obtained from a slice by incorporating information obtained through **testing** or **experiment** (e.g., debugging).

Debugging

- Select a slice.
- Narrow it to a dice.
- Refine the dice till its one faulty stmt.

Dynamic slicing:-

- Is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

→ ***Slicing methods bring together testing, maintenance & debugging.***

Weaker than ACU + p and APU + c

6) **All Predicate Uses (APU), All Computational Uses (ACU) Strategies :**

- **APU :**

  Include definition-free path for every definition of every variable from the definition to predicate use.

- **ACU :**

  Include for every definition of every variable include at least one definition-free path from the definition to every computational use.

It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

7) **ORDERING THE STRATEGIES:**

Figure 3.12 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's head.

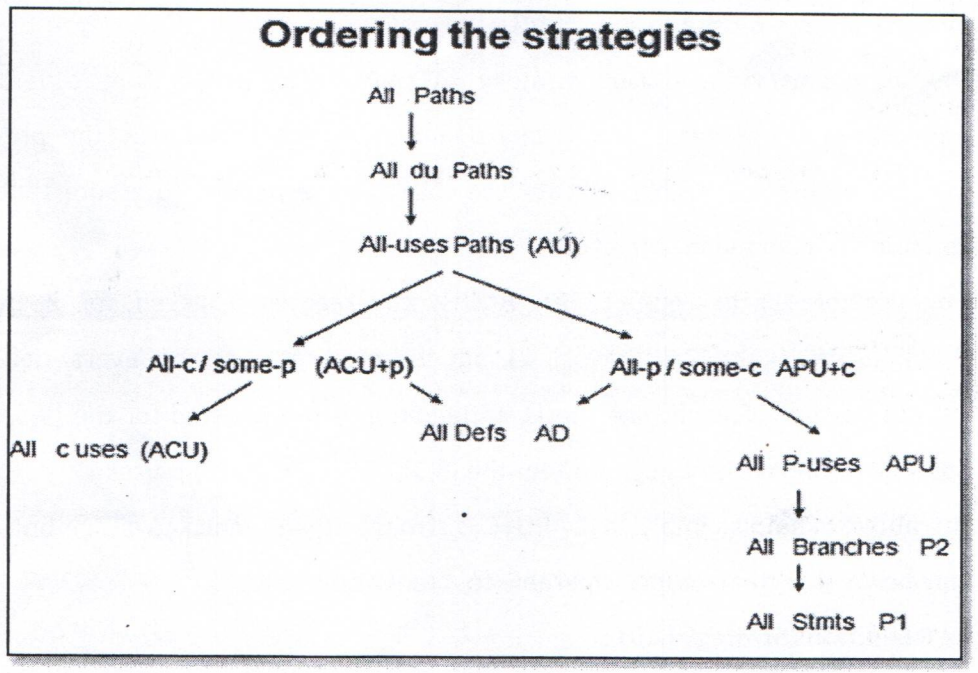


Figure 3.12: Relative Strength of Structural Test Strategies.

The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.

3) **All p-uses/some c-uses strategy (APU+C) :**

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 variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

For variable Z: In Figure 3.10, for APU+C we can select paths that all take the upper 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 for variable Z.

For variable V:In Figure 3.11, 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.

4) **All c-uses/some p-uses strategy (ACU+P) :**

The all **c-uses**/some **p-uses** strategy (ACU+P) is to first ensure coverage by 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.

For variable Z: In Figure 3.10, 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+P may be weaker than, or incomparable to, branch coverage.

5) **All Definitions Strategy (AD) :**

The all definitions strategy asks only every definition of every variable be covered by ***atleast one use of that variable****, be that use a computational use or a predicate use.*

For variable Z: Path (1, 3, 4, 5, 6, 7, 8. . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.

4. **Du-path**:

A path ($n_1 - n_2 - \ldots - n_j - n_k$) is a du-path w.r.t. variable x if node n_1 has a global definition of x and either

i. node n_k has a global c-use of x and ($n_1 - n_2 - \ldots - n_j - n_k$) is a def-clear simple path w.r.t. x, or

ii. Edge (n_j, n_k) has a p-use of x and ($n_1 - n_2 - \ldots - n_j - n_k$) is a def-clear, loop-free path w.r.t. x.

STRATEGIES:

- The structural test strategies are based on the ***program's control flowgraph***. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set.

Various types of data flow testing strategies in decreasing order of their effectiveness are:

1) **All - du Paths (ADUP):**

The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

For variable X and Y: In Figure 3.9, because variables **X** and **Y** are used only on link **(1, 3),** any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy.

2) **All Uses Strategy (AU):**

The all uses 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.

At least one path segment from every definition to every use that can be reached from that definition.

For variable V: In Figure 3.11, 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 to use both.

6. DATA FLOW TESTING STRATEGIES:

INTRODUCTION:

- ***Data Flow Testing Strategies are structural strategies.***
- In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
- Data flow strategies require ***data-flow link weights (d, k, u, c, p).***
- Data Flow Testing Strategies are based on ***selecting test path segments*** (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.

→ For example, all subpaths that contain a d (or u, k, du, dk).

- A strategy X is stronger than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for **weaker**.

TERMINOLOGY:

1. **Definition-Clear Path Segment:-**

   A path **(i, n_1... n_m, j)** is called a *definition-clear path* with respect to x from node **i** to node **j** if it contains no definitions of variable x in nodes (n_1... n_m, j).

   - All paths in Figure 3.9 are definition clear because variables X and Y are defined only on the first link (1, 3) and not thereafter.
   - In Figure 3.10, 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).

2. **Loop-Free Path Segment:-**

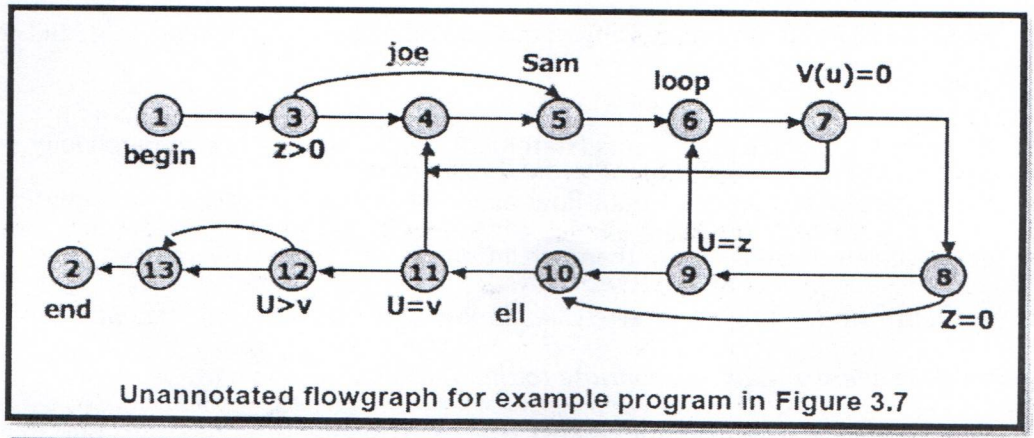
   A loop-free path is a path in which all nodes are distinct (visited only once).

   For Example, path (4,5,6,7,8,10) in **Figure 3.10** is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

3. **Simple path segment:-**

   A simple path is a path in which all nodes, except possibly the first and the last, are distinct.

   For example, in Figure 3.10, (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.



Unannotated flowgraph for example program in Figure 3.7

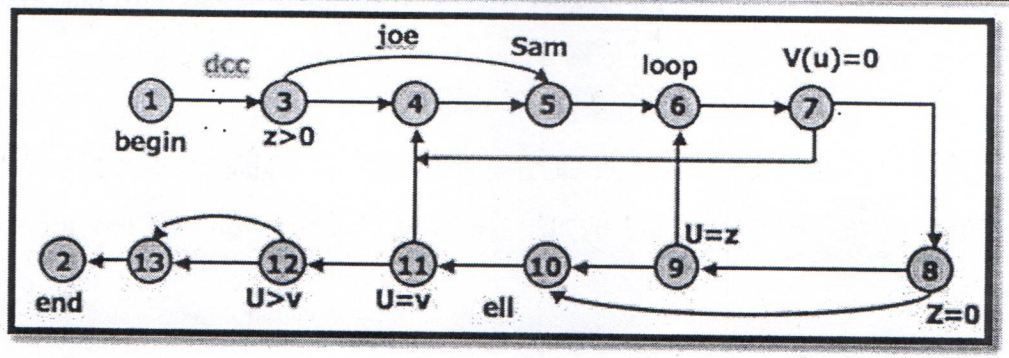


Figure 3.9: Control flow graph annotated for X and Y data flows.

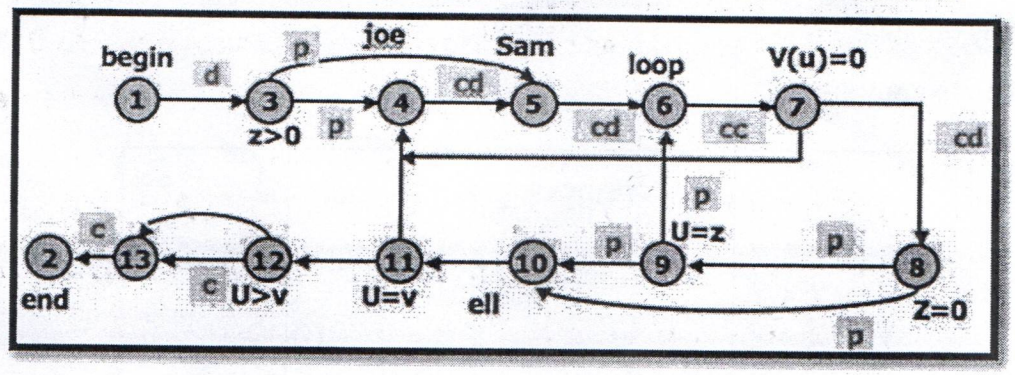


Figure 3.10: Control flow graph annotated for Z data flow.

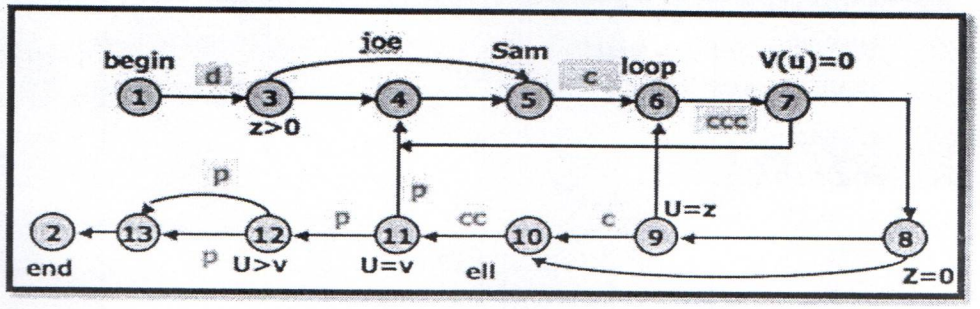


Figure 3.11: Control flow graph annotated for V data flow.

3. Similarly, ***entry nodes are dummy nodes placed at entry statements*** (e.g., BEGIN) for the same reason.
4. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter.
5. ***Predicate nodes*** (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the ***p - use(s) on every outlink, appropriate to that outlink***.
6. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
7. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
8. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Consider an example:-

CODE* (PDL)

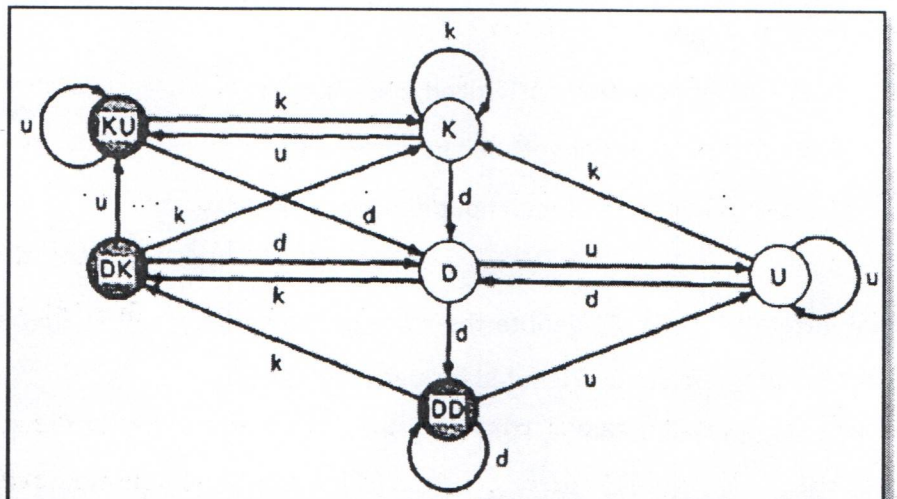
```
      INPUT X, Y                    V(U-1):=V(U+1) + U(V-1)
      Z := X + Y                ELL:V(U+U(V)) := U + V
      V := X - Y                    IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM             IF U > V THEN U := Z
JOE:  Z := Z - 1                    Z := U
SAM:  Z := Z + V                    END
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z - 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
```

Program Example (PDL)

* A contrived horror

If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

b) **Forgiving Data - Flow Anomaly Flow Graph:** Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible.



Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states

5.2. DATA FLOW MODEL:

- The data flow model is based on the program's control flow graph
- Here we annotate each link with symbols (d, k, u, c, p) or sequences of symbols (dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.
- The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

1. To every statement there is a node, whose name is unique. ***Every node has at least one outlink and at least one inlink*** except for exit nodes and entry nodes.
2. ***Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements*** (e.g., END, RETURN), to complete the graph.

4. **k-** :- not anomalous. The last thing done on this path was to kill the variable.
5. **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
6. **u-** :- not anomalous. The variable was used but not killed on this path.

DATA FLOW ANOMALY STATE GRAPH:

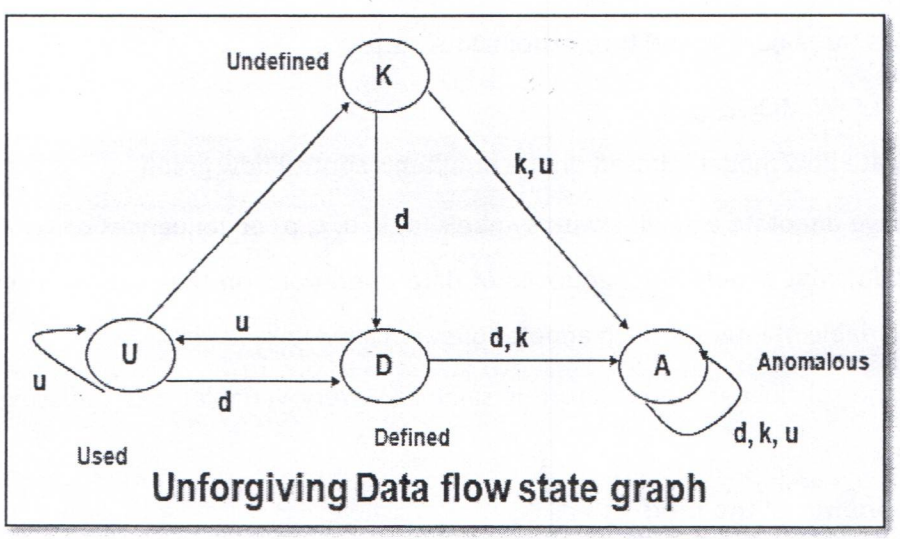
- Data flow anomaly model prescribes that an object can be in one of **four** distinct states:

0. **K: -** undefined, previously killed, doesnt exist
1. **D: -** defined but not yet used for anything
2. **U: -** has been used for computation or in predicate
3. **A: -** anomalous

These capital letters **(K, D, U, A)** denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

There are **two types** of Data Flow Anomaly Flow Graphs:-

a) **Unforgiving Data - Flow Anomaly Flow Graph:** Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.



The variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., opening, closing, and using files and that 'killing'), **the object's state becomes anomalous (state A)** and, once it is anomalous, **no action** can return the variable to a working state.

DATA FLOW ANOMALIES:

- *An anomaly is denoted by a two-character sequence of actions.*
- What is an anomaly is depend on the application.
- For example, **ku** means that the object is **killed** and then **used**, where as **dd** means that the object is **defined twice** without an intervening usage.
- There are ***nine*** possible **two-letter combinations** for d, k and u. some are **bugs**, some are **suspicious**, and some are **okay**.

→ **A Two letter sequence of Actions (d, k, u)**

1. **dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
2. **dk** :- probably a bug. Why define the object without using it?
3. **du** :- the normal case. The object is defined and then used.
4. **kd** :- normal situation. An object is killed and then redefined.
5. **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
6. **ku** :- a bug. the object does not exist.
7. **ud** :- usually not a bug because the language permits reassignment at almost any time.
8. **uk** :- normal situation.
9. **uu** :- normal situation.

→ In addition to the two letter situations, there are **six** single letter situations.

- **Leading dash (-) no action from START to this point from this point till the EXIT.**
- **(-)A trailing dash to mean that nothing happens after the point of interest to the exit.**

They possible anomalies are:

1. **-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
2. **-d** :- okay. This is just the first definition along this path.
3. **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.

5.1. DATA FLOW GRAPHS:

- The data flow graph is a **graph** consisting of **nodes** and **directed links**.
- A spec. for relations among the data objects.

Data Object State and Usage:

Data Objects can be **created**, **killed** and **used**.

They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.

The following symbols denote these possibilities: **Program Actions (d, k, u):**

1. **Defined: d** - defined, created, initialized etc
2. **Killed or undefined: k** - killed, undefined, released etc
3. **Usage: u** - used for something (**c** - used in Calculations, **p** - used in a predicate)

1. Defined (d):

- An object is defined explicitly when it appears in a data declaration.
- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.
- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

2. Killed or Undefined (k):

- An object is killed or undefined when it is released or otherwise made unavailable.
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately.

3. Usage (u):

- A variable is used for **computation (c)** when it appears on the right hand side of an assignment statement.
- A file record is **read** or **written**.
- It is used in a **Predicate (p)** when it appears directly in a predicate.

Multi-instruction, Multi-data machines (MIMD) Architecture:

- These machines can fetch several instructions and objects in parallel.
- They can also do arithmetic and logical operations simultaneously on different data objects.
- The decision of how to sequence them depends on the compiler.

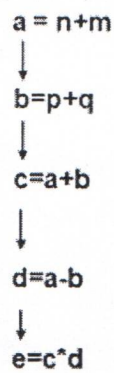
Program Control flow with Von Neumann's paradigm

Given m, n, p, q, find e.

e = (m+n+p+q) * (m+n-p-q)

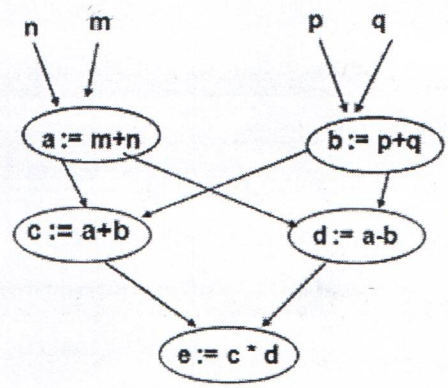
```
a := m + n
b := p + q
c := a + b
d := a - b

e := c * d
```



Program Flow using Data Flow Machines paradigm

```
BEGIN
 PAR DO
      READ m, n, n, p, q
END PAR
PAR DO
      a := m+n
      b := p+q
END PAR
PAR DO
      c := a+b
      d := a-b
END PAR
PAR DO
      e := c * d
END PAR
END
```



Data Flow Graph

DATA FLOW TESTING

Definition:

DFT is a family of test strategies based on selecting paths through the program's control flow in order to explore the sequence of events related to the status of data objects.

Data Flow Testing (DFT) uses Control Flow Graph (CFG) to explore dataflow anomalies.

Example:

Pick enough paths to assure that every data item has been initialized prior to its use, or that all objects have been used for something.

5. DATA FLOW MACHINES:

There are **two** types of data flow machines with different architectures.

(1) Von Neumann machines

(2) Multi-instruction, multi-data machines (MIMD).

Von Neumann Machine Architecture:

- Most computers today are von-Neumann machines.
- This architecture features interchangeable storage of instructions and data in the same memory units.
- The Von Neumann machine Architecture ***executes one instruction at a time*** in the following, micro instruction sequence:

1. Fetch instruction from memory
2. Interpret instruction
3. Fetch operands
4. Process or Execute
5. Store result
6. Increment program counter
7. GOTO 1

- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

PATH SENSITIZATION:

- Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
- The remaining small percentage is often very difficult.
- **Sensitization is the act of defining the transaction.** If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

PATH INSTRUMENTATION:

- Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
- ***The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.***
- In some systems, such traces are provided by the operating systems or a running log.

Design and Maintain Test Database:-

- Test database is a set of data created for testing new or revised applications. Test data must contain a sample of every category of valid data as well as many invalid conditions as possible.
- Test Data shall be availed before a test is executed. Design and maintenance of required test databases is not only time consuming but also, shall be ensured at the beginning of the transaction flow testing to verify functionality of a system.

4. TRANSACTION FLOW TESTING TECHNIQUES:

GET THE TRANSACTIONS FLOWS:

- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

INSPECTIONS, REVIEWS AND WALKTHROUGHS:

- Transaction flows are natural agenda for system reviews or inspections.
- In conducting the walkthroughs, you should:
  - Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
  - Discuss paths through flows in functional rather than technical terms.
  - Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
- Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
- Select additional flow paths for loops, extreme values, and domain boundaries.
- Design more test cases to validate all births and deaths.
- Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert (=will make use of) the maximum beneficial effect on the project.

PATH SELECTION:

- Select a set of covering paths (c1+c2) using the analogous (=similar or equivalent) criteria you used for structural path testing.
- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.

3.2. **Mergers:** Transaction flow junction points are potentially as troublesome as transaction flow splits.

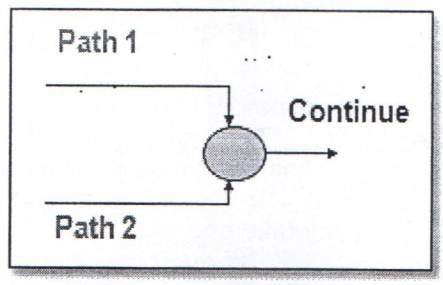
There are **three** types of junctions:

(1) Ordinary Junction

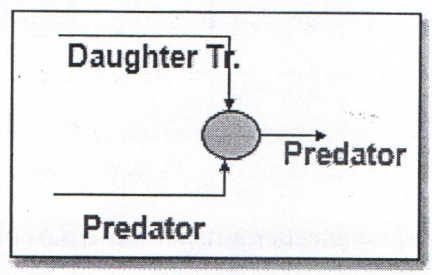
(2) Absorption

(3) Conjugation

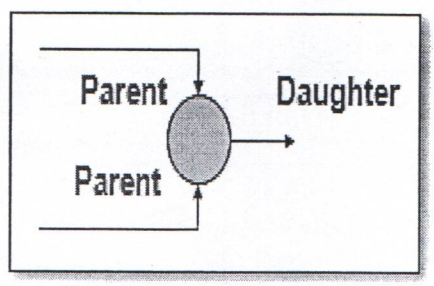
1) **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other.



2) **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity.



3) **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter.



Uses of Transaction-flow:-

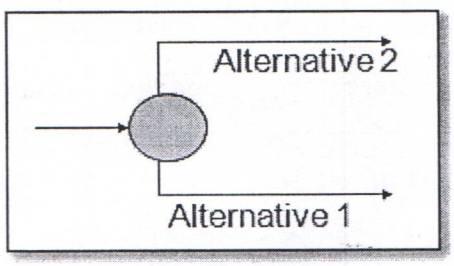
- Specifying requirements of big, online and complicated systems.
- Airline reservation systems, air-traffic control systems.
- Loops are less as compared to CFG. Loops are used for user input error processing

3. COMPLICATIONS:

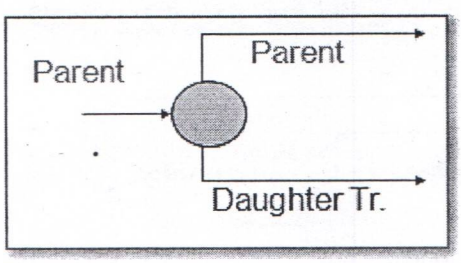
- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

3.1. Births: There are **three** different possible interpretations of the decision symbol, or nodes with two or more out links. It can be a **Decision**, **Biosis** or **Mitosis**.

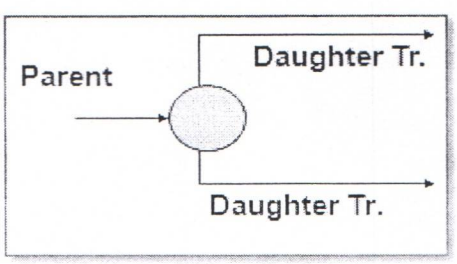
1. **Decision:** Here the transaction will take one alternative or the other alternative but not both.



2. **Biosis:** Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains its identity.



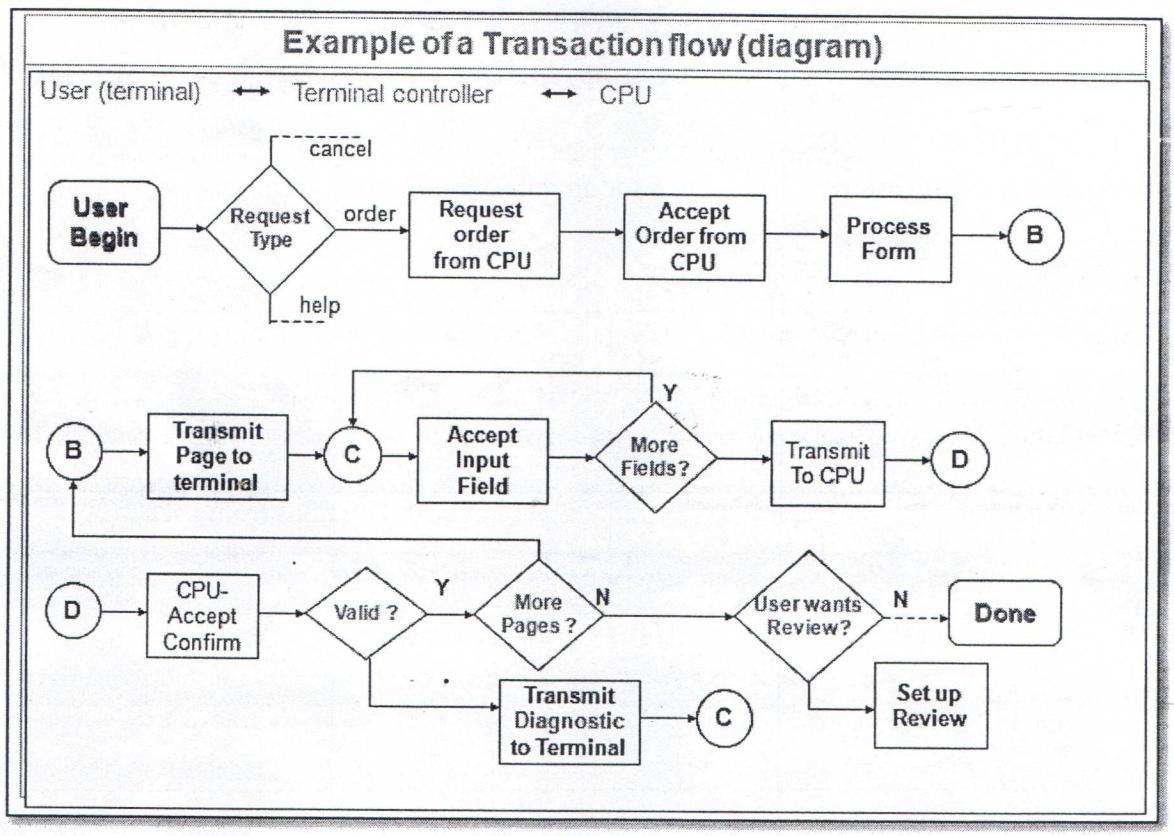
3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created.



2. TRANSACTION FLOW GRAPHS:

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- **Transaction flows** and **transaction flow testing** are to the independent system tester what control flows are path testing is to the programmer.
- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flow graph is a model of the structure of the system's behavior (functionality).

An example of a Transaction Flow is as follows:



Transaction-flow Graph : a scenario between users & computer

Transaction-flow : an internal sequence of events in processing a transaction

Introduction:-

Transaction-flow:-

- Transaction-flow represents a system's processing. Functional testing methods are applied for testing T-F.

Transaction-flow Graph:-

- TFG represents a behavioral (functional) model of the program (system) used for functional testing by an independent system tester.

1. TRANSACTION FLOWS:

- **A transaction is a unit of work seen from a system user's point of view.**
- 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.
- Transaction begins with Birth-that is they are created as a result of some external act.

Example of a transaction: A transaction for an online information retrieval system might consist of the following steps or tasks:

1. Accept input (tentative birth)	**2. Validate input (birth)**
3. Transmit acknowledgement 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 clean up (death)

Note:-

- Users View of a transaction : Single step
- Systems view : Sequence of many operations

reproduced. Automate test execution.

- Be creatively stupid when conducting tests. Every deviation from the predicted outcome or path must be explained. Every deviation must lead to either a test change, a code change, or a conceptual change.
- A test that reveals a bug has succeeded, not failed.

We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

CREC, Dept Of MCA Page 64

Two Link Marker Method:

- The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and on at the end.
- The two link markers now specify the path name and confirm both the beginning and end of the link.

Figure 2.14: Double Link Marker Instrumentation.

Link Counter: A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

APPLICATION OF PATH TESTING:

- Path testing based on structure is a powerful unit-testing tool. With suitable interpretation, it can be used for system functional tests.
- The objective of path testing is to execute enough tests to assure that, as a minimum, C1 + C2 have been achieved.
- Select paths as deviations from the normal paths, starting with the simplest, most familiar, most direct paths from the entry to the exit. Add paths as needed to achieve coverage.
- Add paths to cover extreme cases for loops and combinations of loops: no looping, once, twice, one less than the maximum, the maximum. Attempt forbidden cases.
- Find path-sensitizing input-data sets for each selected path. If a path is unachievable, choose another path that will also achieve coverage. But first ask yourself why seemingly sensible cases lead to unachievable paths.
- Use instrumentation and tools to verify the path and to monitor coverage.
- Incorporate the notion of coverage (especially C2) into all reviews and inspections.
- Make the ability to achieve C2 a major review agenda item.

CREC, Dept Of MCA Page 65

- Design test cases and path from the design flowgraph or PDL specification but sensitize paths from the code as part of desk checking. Do covering test case designs either prior to coding or concurrently with coding.
- Document all tests and expected test results as copiously as you would document code. Put test suites under the same degree of configuration control used for the software it tests. Treat each path like a subroutine. Predict and document the outcome for the stated inputs and the path trace (or name by links). Also document any significant environmental factors and preconditions.
- Your tests must be reproducible so that they can serve a diagnostic purpose if they reveal a bug. An undocumented test cannot be

• If coverage has not been achieved extend the cases to those that involve dependent predicates.
• Last, use correlated, dependent predicates.

PATH INSTRUMENTATION:
PATH INSTRUMENTATION:

• Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
• **Co-incidental Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason.

PATH INSTRUMENTATION:
PATH INSTRUMENTATION:

o Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
o **Co-incidental Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason.

Figure 2.11: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.

The types of instrumentation methods include:

Interpretive Trace Program:

o An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.
o If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
o The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.

Traversal Marker or Link Marker:

A simple and effective form of instrumentation is called a traversal marker or link marker.

o Name every link by a lower case letter.
o Instrument the links so that the link's name is recorded when the link is executed.
o The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

Figure 2.12: Single Link Marker Instrumentation

o **Why Single Link Markers aren't enough:** Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.

Figure 2.13: Why Single Link Markers aren't enough.

Correct	Buggy
X = A ..... if X-1 > 0 then ...	X = A ..... if X+A-2 > 0 then ...

The assignment (x=a) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

PATH SENSITIZING:

REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.

Extract the programs control flowgraph and select a set of tentative covering paths.

- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as

(A+BC) (D+E) (FGH) (IJ) (K) (l) (L).

- Multiply out the expression to achieve a sum of products form:

ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHI JKL

- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you cant find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION.**

HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.

TESTING BLINDNESS:

• Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.

There are three types of Testing Blindness:

Assignment Blindness:

Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate.

For Example:

Correct	Buggy
X = 7 ..... if Y > 0 then ...	X = 7 ..... if X+Y > 0 then ...

If the test case sets Y=1 the desired path is taken in either case, but there is still a bug.

Equality Blindness:

Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.

For Example:

Correct	Buggy
if Y = 2 then ..... if X+Y > 3 then ...	if Y = 2 then ..... if X > 1 then ...

The first predicate if y=2 forces the rest of the path, so that for any positive value of x. the path taken at the second predicate will be the same for the correct and buggy version.

Self Blindness:

Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.

For Example:

Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

PATH PREDICATE EXPRESSIONS:

A path predicate expression is a set of Boolean expressions, all of which must be satisfied to achieve the selected path.

X1+3X2+17>=0

X3=17

X4-X1>=14X2

Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.

Sometimes a predicate can have an **OR** in it.

Example:

A: X5 > 0	**E: X6 < 0**
B: X1 + 3X2 + 17 >= 0	**B: X1 + 3X2 + 17 >= 0**
C: X3 = 17	**C: X3 = 17**
D: X4 - X1 >= 14X2	**D: X4 - X1 >= 14X2**

Boolean algebra notation to denote the Boolean expression:

ABCD+EBCD= (A+E) BCD

PREDICATE COVERAGE:

- **Compound Predicate:** Predicates of the form A OR B, A AND B and more complicated boolean expressions are called as compound predicates.
- Sometimes even a simple predicate becomes compound after interpretation.
  Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17 . Or. X<17.
- ***Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.***

HEM: DO SOMETHING

.....

HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

INDEPENDENCE OF VARIABLES AND PREDICATES:

The path predicates take on truth values based on the values of input variables, either directly or indirectly.

If a variable's value does not change as a result of processing, **that variable is independent of the processing.**

If the variable's value can change as a result of the processing, **the variable is process dependent.**

A predicate ***whose truth value can change*** as a result of the processing is said to be **process dependent.**

A predicate **whose truth value does not change** as a result of the processing is **process independent.**

Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

CORRELATION OF VARIABLES AND PREDICATES:

A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.

Two variables are correlated if every combination of their values cannot be independently specified.

Variables whose values can be specified independently without restriction are called uncorrelated.

For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.

Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multiway branches as an equivalent set of if..then..else statements.

For example a three way case statement can be written as: If case=1 DO A1

ELSE (IF Case=2 DO A2 ELSE DO A3 ENDIF)ENDIF.

INPUTS:

In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.

For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types.

The input for a particular test is mapped as a one dimensional array called as an Input Vector.

PREDICATE INTERPRETATION:

The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called predicate interpretation.

The simplest predicate depends only on input variables.

For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.

Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignment statement y=x2+7. Although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.

Some times the interpretation may depend on the path; for example,

```
INPUT X
ON X GOTO A, B, C, ...

A: Z := 7 @ GOTO HEM
B: Z := -7 @ GOTO HEM
C: Z := 0 @ GOTO HEM
.....
```

The above path leads to the following Table

PATHS	DECISIONS				PROCESS–LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			√	√	√	√	√								
abhkgde	NO	YES		NO	√	√		√	√		√	√			√		
abhlibcde	NO, YES	YES		YES	√	√	√	√	√			√	√			√	
abcdfjgde	YES	NO, YES	YES		√	√	√	√	√	√	√			√			
abcdfmibcde	YES	NO, YES	NO		√	√	√	√	√	√			√				√

After you have traced a covering path set on the master sheet and filled in the table for every path, check the following:

- Does every decision have a **YES** and a **NO** in its column? **(C2)**
- Has every case of all case statements been marked? **(C2)**
- Is every three - way branch (less, equal, greater) covered? **(C2)**
- Is every link (process) covered at least once? **(C1)**

7.4. PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS:

PREDICATE:-*The logical function evaluated at a decision is called Predicate.*

The direction taken at a decision depends on the value of decision variable.

Some examples are: A>0, x+y>=90.....

PATH PREDICATE: *A predicate associated with a path is called a Path Predicate.*

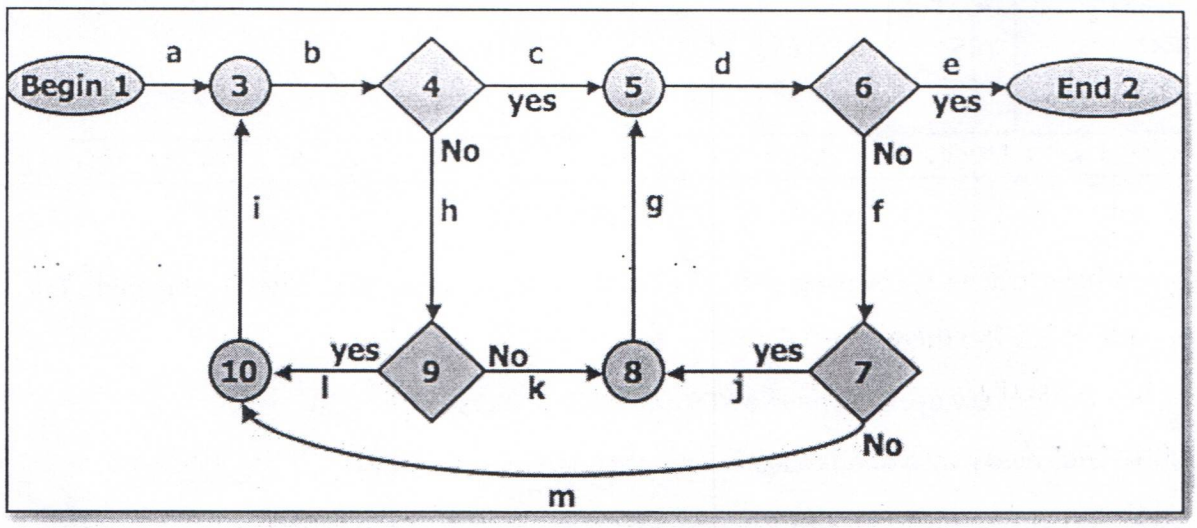
For example, "x is greater than zero", "x+y>=90", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

MULTIWAY BRANCHES:

The path taken through a multiway branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms.

Which paths to be tested? You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

Path Selection Example:



Practical Suggestions in Path Testing:

- Draw the control flow graph on a single sheet of paper.
- Make several copies - as many as you will need for coverage (C1+C2) and several more.
- Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
- Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
- As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.

PATH TESTING CRITERIA:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

There **THREE** different kinds of path testing criteria. They are

1. ***Path Testing (P_{inf})***
2. ***Statement Testing (P_1)***
3. ***Branch Testing (P_2)***

Path Testing (Pinf):

- Execute **all possible control flow paths through the program**: typically, this is restricted to all possible entry/exit paths through the program.
- If we achieve this prescription, we are said to have achieved 100% path coverage. This is the **strongest criterion in the path testing strategy family**:
- It is generally impossible to achieve.

Statement Testing (P1):

- **Execute all statements in the program at least once under some test.**
- If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
- An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by **C1.**
- This is the **weakest criterion in the family**: testing less than this for new software is unconscionable (unprincipled or cannot be accepted) and should be criminalized.

Branch Testing (P2):

- Execute enough tests to assure that **every branch alternative has been exercised at least once under some test.**
- If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
- An alternative characterization is to say that we have achieved 100% link coverage.
- For structured software, branch testing and therefore branch coverage strictly includes statement coverage. We denote branch coverage by **C2**.

7.3. PATH TESTING - PATHS, NODES AND LINKS:

Path: *a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.*

A path may go through several junctions, processes, or decisions, one or more times.

A path consists of segments.

- The **segment is a link** - a single process that lies between two nodes.
- A path segment is succession of consecutive links that belongs to some path.
- The **length of path** measured by the number of links in it and not by the number of the instructions or statements executed along that path.
- The **name of a path** is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

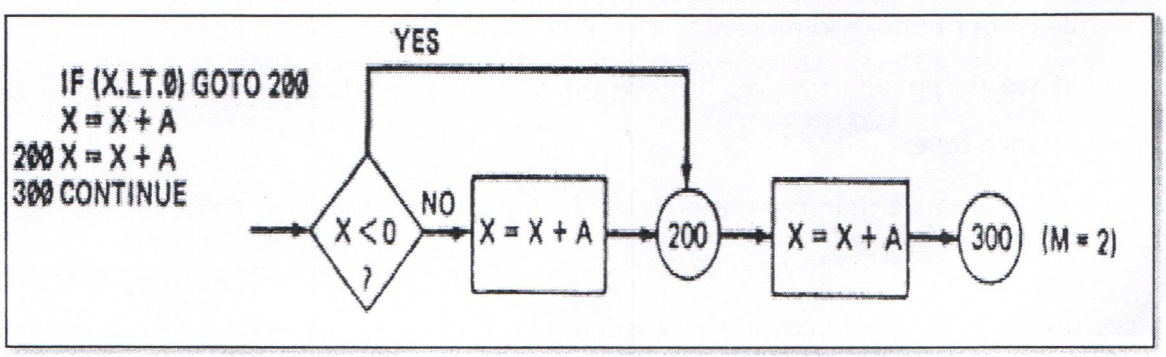
There are many paths between the entry and exit of a typical routine.

Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

- Exercise every path from entry to exit
- Exercise every statement or instruction at least once
- Exercise every branch and case statement, in each direction at least once.

EXAMPLE: **Here is the correct version.**



We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like

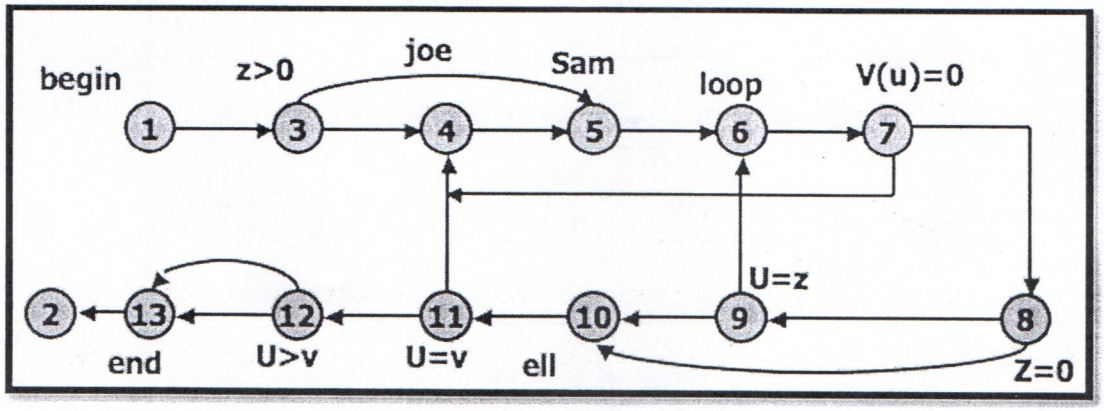


Figure 2.5: Simplified Flowgraph Notation

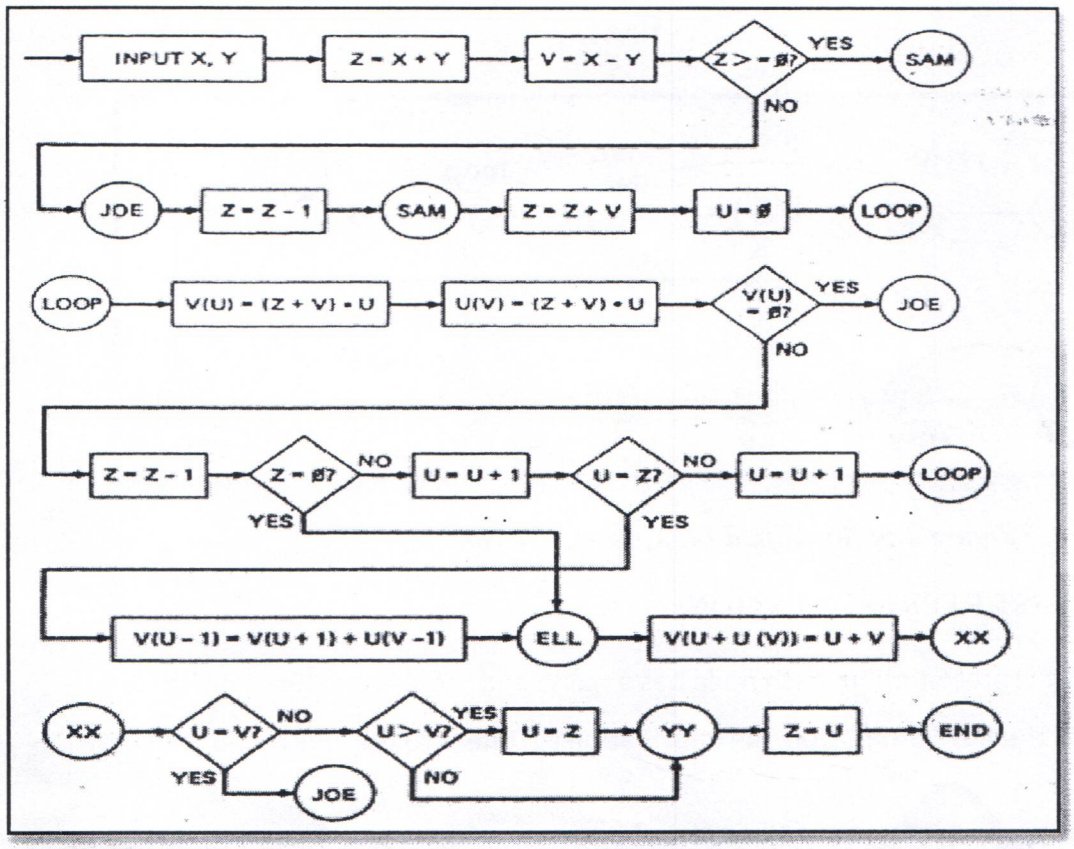
LINKED LIST REPRESENTATION:

In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. Only the information pertinent to the control flow is shown.

Linked List representation of Flow Graph:

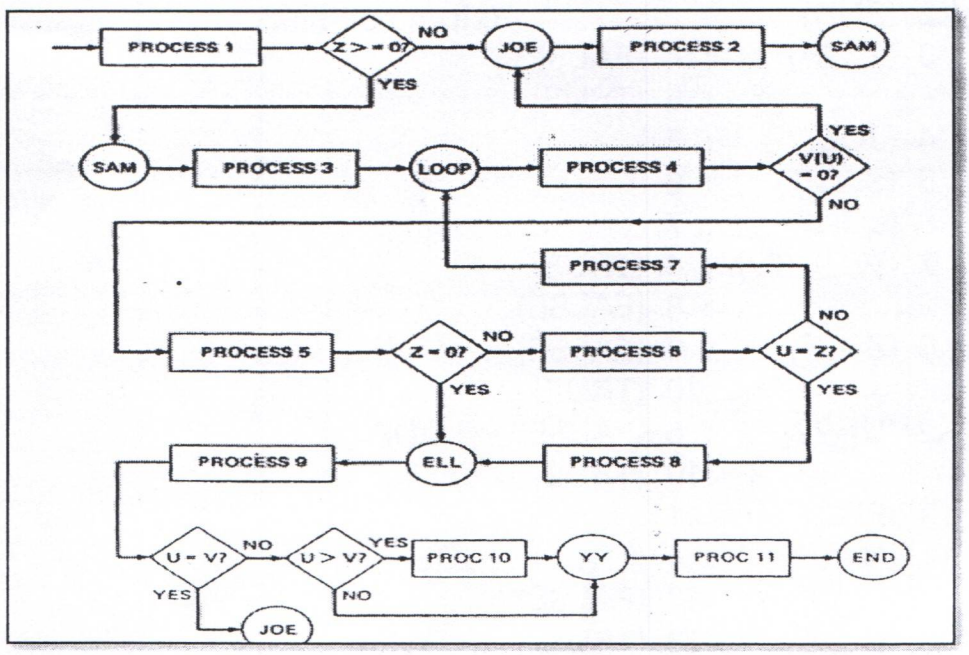
```
1  (BEGIN)     : 3
2  (END)       :                Exit, no outlink
3  (Z>ø?)      : 4 (FALSE)
               : 5 (TRUE)
4  (JOE)       : 5
5  (SAM)       : 6
6  (LOOP)      : 7
7  (V(U)=ø?)   : 4 (TRUE)
               : 8 (FALSE)
8  (Z=ø?)      : 9 (FALSE)
               :10 (TRUE)
9  (U=Z?)      : 6 (FALSE) = LOOP
               :10 (TRUE) = ELL
10 (ELL)       :11
11 (U=V?)      : 4 (TRUE) = JOE
               :12 (FALSE)
12 (U>V?)      :13 (TRUE)
               :13 (FALSE)
13             : 2 (END)
```

The first step in translating the program to a flowchart is shown in Figure 2.3.



One-to-one flowchart for example program

In **Figure 2.4** we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy.



Control Flowgraph

CONTROL FLOW GRAPHS Vs FLOWCHARTS:

- A program's flow chart resembles a **control flow graph**.
- In flow graphs, we **don't show the details of** what is in a process block.
- In flow charts **every part of the process** block is drawn.
- The flowchart focuses **on process steps**, where as the flow graph focuses on **control flow of the program.**
- The act of drawing a control flow graph is a useful tool that can help us clarify the **control flow** and **data flow issues**.

NOTATIONAL EVOULTION:

The control flow graph is simplified representation of the program's structure.

The notation changes made in creation of control flow graphs:

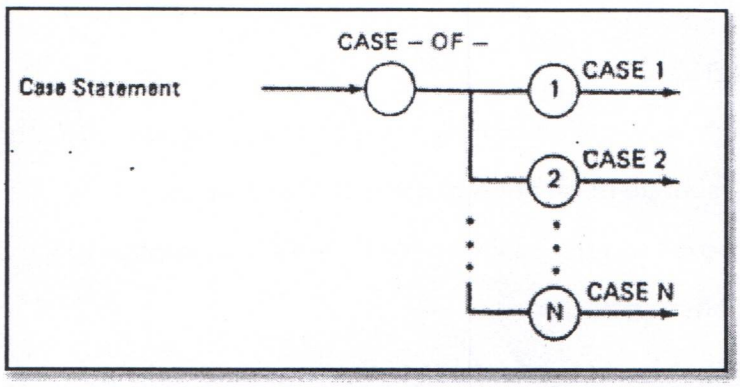
- The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- We don't need to know the specifics of the decisions, just the fact that there is a branch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- an example written in a FORTRAN like programming language called **Programming Design Language (PDL).**

CODE* (PDL)

```
      INPUT X, Y                      V(U-1):=V(U+1) + U(V-1)
      Z := X + Y                ELL:V(U+U(V)) := U + V
      V := X - Y                      IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM               IF U > V THEN U := Z
JOE:  Z := Z - 1                      Z := U
SAM:  Z := Z + V                      END
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z - 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
```

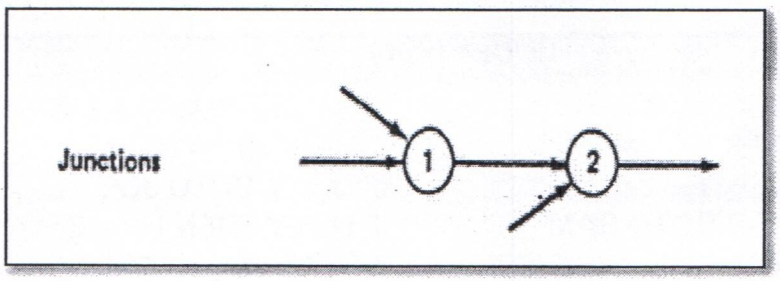
3. Case Statements:

- A case statement is a multi-way branch or decisions.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
- From the point of view of test design, there are no differences between Decisions and Case Statements. It is denoted by symbol



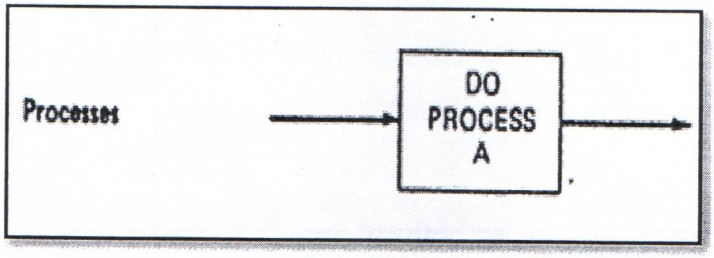
4. Junctions:

- A junction is a point in the program where the control flow can merge.
- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO. It is denoted by symbol



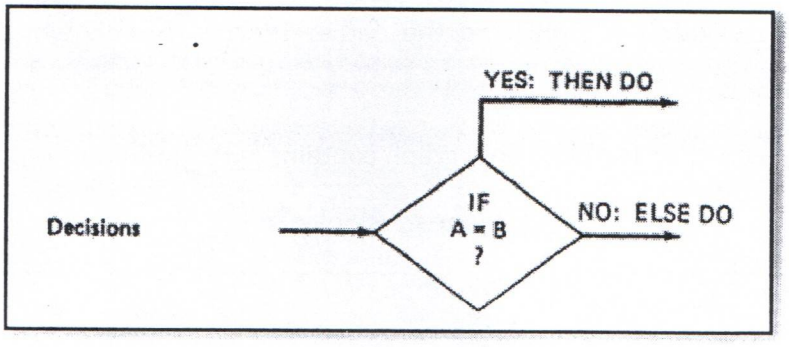
1. Process Block:

- *A process block is a sequence of program statements uninterrupted by either decisions or junctions.*
- It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof is executed.
- Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has **one entry** and **one exit**. It can consist of a **single statement** or **instruction**, a **sequence** of **statements** or **instructions**, a **single entry/exit** subroutine, a macro or function call, or a sequence of these.
- It is denoted by symbol



2. Decisions:

- A decision is a program point at which the control flow can diverge.
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- Most of the decisions are two-way but some are three way branches in control flow. It is denoted by symbol



FLOWGRAPHS AND PATH TESTING

7. BASICS OF PATH TESTING:

7.1. PATH TESTING:

- *Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.*
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- **Path testing techniques are the oldest of all structural test techniques.**
- Path testing is most applicable to new software for unit testing. **It is a structural technique.**
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

7.2. CONTROL FLOW GRAPHS:

- ***The control flow graph is a graphical representation of a program's control structure.*** It uses the elements named **process blocks, decisions,** and **junctions.**
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.

Flow Graph Elements: A flow graph contains **FOUR** different types of elements.

(1) Process Block

(2) Decisions

(3) Junctions

(4) Case Statements

Remedies:

1. Test Debugging:

Testing & Debugging tests, test scripts etc. Simpler when tests have localized affect.

2. Test Quality Assurance:

To monitor quality in independent testing and test design.

3. Test Execution Automation:

Test execution bugs are eliminated by test execution automation tools & not using Manual testing.

4. Test Design Automation:

Test design is automated like automation of software development. For a given Productivity rate, It reduces bug count.

****

Resource Management Remedies: A design remedy that prevents bugs is always preferable to a test method that discovers them.

- The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

8. **Integration Bugs:**
   - Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
   - These bugs results from inconsistencies or incompatibilities between components.
   - The communication methods include data structures, call sequences, registers, semaphores, and communication links and protocols results in integration bugs.
9. **System Bugs:**
   - System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.
   - There can be no meaningful system testing until there has been thorough component and integration testing.
   - All kinds of tests at all levels as well as integration tests - are useful.

6. TEST AND TEST DESIGN BUGS:

- Bugs in Testing (scripts or process) are not software bugs.
- It's difficult & takes time to identify if a bug is from the software or from the test script/procedure.

Bugs could be due to:

- Tests require code that uses complicated scenarios & databases, to be executed.
- Though an independent functional testing provides an un-biased point of view, this lack of bias may lead to an **incorrect interpretation of the specs**.

4. **Operating System Bugs:**
   - Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does.
   - Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls.
   - This approach may not eliminate the bugs but at least will localize them and make testing easier.
5. **Software Architecture:**
   - Software architecture bugs are the kind that called - interactive.
   - Routines can pass unit and integration testing without revealing such bugs.
   - Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
   - Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.
6. **Control and Sequence Bugs (Systems Level):**
   - These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.
   - The remedy for these bugs is highly structured sequence control.
   - Specialize, internal, sequence control mechanisms are helpful.
7. **Resource Management Problems:**
   - Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.
   - External mass storage units such as discs, are subdivided into memory resource pools.
   - Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc

5.5. INTERFACE, INTEGRATION, AND SYSTEM BUGS:

There are **9** various categories of bugs in Interface, Integration, and System Bugs are:

1. **External Interfaces:**
   - The external interfaces are the means used to communicate with the world.
   - Means to communicate with the world: drivers, sensors, input terminals, communication lines.
   - Primary design criterion should be - **robustness.**
   - **Bugs**: invalid timing, sequence assumptions related to external signals, misunderstanding external formats and no robust coding.
   - **Domain testing**, **syntax testing** & **state testing** are suited to testing external interfaces.
2. **Internal Interfaces:**
   - Must adapt to the external interface.
   - Have bugs similar to external interface
   - Bugs from improper
   - Protocol design, input-output formats, protection against corrupted data, subroutine call sequence, call-parameters.

Remedies (prevention & correction):

- Test methods of domain testing & syntax testing.
- Good integration testing is to test all internal interfaces with external world.

3. **Hardware Architecture:**
   - Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
   - Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
   - The remedy for hardware architecture and interface problems is twofold:

     (1) Good Programming and Testing

     (2) Centralization of hardware interface software in programs written by hardware interface specialists.

3. **Information, parameter, and control:** Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

   Examples: name, hash code, function using these. A variable in different contexts.

   - **Information:** dynamic, local to a single transaction or task.
   - **Parameter**: parameters passed to a call.
   - **Control**: data used in a control structure for a decision.

4. **Content, Structure and Attributes:**

   - **Contents:** are pure bit pattern & bugs are due to misinterpretation or corruption of it.
   - **Structure**: Size, shape & alignment of data object in memory. A structure may have substructures.
   - **Attributes**: Semantics associated with the contents (e.g. integer, string, and subroutine).

Preventive Measures (prevention & correction)

- Good source lang. documentation & coding style (incl. data dictionary).
- Data structures be globally administered. Local data migrates to global.
- Strongly typed languages prevent mixed manipulation of data.

5.4. CODING BUGS:

- **Coding errors** create other kinds of bugs.
- **Syntax errors** are removed when compiler checks syntax.
- **Coding errors**

  Typographical, misunderstanding of operators or statements or could be just arbitrary.
- **Documentation Bugs**
  - Erroneous comments could lead to incorrect maintenance.
  - Testing techniques cannot eliminate documentation bugs.
- **Solution**:

  Inspections, QA, automated data dictionaries & specification systems.

Remedies (prevention & correction)

→ Programming tools, explicit declaration & type checking in source language, preprocessors.

→ **Data flow** test methods help design of tests and debugging.

5. **Data-Flow Bugs and Anomalies:**

- Run into an un-initialized variable.
- Not storing modified data.
- Re-initialization without an intermediate use.
- Detected mainly by execution (testing).

Remedies (prevention & correction)

- **Data flow** testing methods & **matrix** based testing methods.

5.3. DATA BUGS:

Depend on the types of data or the representation of data. There are 4 sub categories.

1. **Generic Data Bugs**
2. **Dynamic Data Vs Static Data**
3. **Information, Parameter, and Control Bugs**
4. **Contents, Structure & Attributes related Bugs**

1. **Generic Data Bugs:-**

- Due to data object specs., formats, # of objects & their initial values.
- Common as much as in code, especially as the code migrates to data.
- Data bug introduces an operative statement bug & is harder to find.

2. **Dynamic Data Vs Static data:**

Dynamic Data	Static data
Transitory. Difficult to catch.	Fixed in form & content.
Due to an error in a shared storage object initialization.	Appear in source code or data base, directly or indirectly
Due to unclean / leftover garbage in a shared resource.	Software to produce object code creates a static data table – bugs possible
Prevention **Data validation, unit testing**	**Prevention** Compile time processing Source language features

1. **Control and Sequence Bugs:**

- Control and sequence bugs include paths left out, unreachable code.
- Improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing.

Prevention and Control:

→ Theoretical treatment and,

→ **Unit, structural, path, & functional testing.**

2. **Logic Bugs:**

- Misunderstanding of the semantics of the control structures & logic operators
- Improper layout of cases, including impossible & ignoring necessary cases,
- Using a look-alike operator, improper simplification, confusing Ex-OR with inclusive OR.
- Deeply nested conditional statements & using many logical operations in 1 stmt.

Prevention and Control:

→ **Logic** testing, careful checks, **functional** testing

3. **Processing Bugs:**

- Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
- Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of greater-than-or-equal etc.

Prevention

→ Caught in **Unit Testing** & have only localized effect

→ **Domain testing** methods

4. **Initialization Bugs:**

- Forgetting to initialize work space, registers, or data areas.
- Wrong initial value of a loop control parameter.
- Accepting a parameter without a validation check.
- Initialize to wrong data type or format.

- Removing the features might complicate the software, consume more resources, and foster more bugs.

3. Feature Interaction Bugs:
   - Providing correct, clear, implementable and testable feature specifications is not enough.
   - The problem is unpredictable interactions between feature groups or even between individual features. For **example,** your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.

Remedies:

- Use high level formal specification languages to eliminate human-to-human communication
- It's only a short term support & not a long term solution.
- ***Short term Support:*** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.
- ***Long term Support:*** Even with a great specification language, problem is not eliminated, but is shifted to a higher level. Simple ambiguities & contradictions may only be removed, leaving tougher bugs.

Testing Techniques:

Functional test techniques - **transaction flow testing, syntax testing, domain testing, logic testing, and state testing** - can eliminate requirements & specifications bugs.

5.2. STRUCTURAL BUGS:

We look at the 5 types, their causes and remedies.

1. **Control & Sequence bugs**
2. **Logic Bugs**
3. **Processing bugs**
4. Initialization bugs
5. Data flow bugs & anomalies

TAXONOMY OF BUGS:

5. **TAXONOMY**: - To study the consequences, nightmares, probability, importance, impact and the methods of prevention and correction.

- There is no universally correct way categorize bugs. The taxonomy is not rigid.
- A given bug can be put into one or another category depending on its history and the programmer's state of mind.
- There are **SIX** main categories are:
    1. Requirements, Features and Functionality Bugs
    2. Structural Bugs
    3. Data Bugs
    4. Coding Bugs
    5. Interface, Integration and System Bugs
    6. Test and Test Design Bugs.

5.1. REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS:

3 types of bugs: **Requirement & Specs, Feature, & feature interaction bugs.**

1. **Requirements and Specifications Bugs:**
    - Requirements and specifications developed from them can be **incomplete ambiguous, or self-contradictory.**
    - The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted.
    - The range is from a few percentages to more than 50%, depending on the application and environment.
2. **Feature Bugs:**
    - Specification problems usually create corresponding feature problems.
    - A feature can be wrong, missing, or superfluous (serving no useful purpose).
    - A missing feature or case is easier to detect and correct.
    - A wrong feature could have deep design implications.

6. **Very Serious:** System does another transaction instead of requested e.g. Credit another account, convert withdrawals to deposits.
7. **Extreme:** The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.
8. **Intolerable:** Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
9. **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.
10. **Infectious:** Corrupts other systems, even when it may not fail.

4.3. FLEXIBLE SEVERITY RATHER THAN ABSOLUTES:

- Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component.
- Many organizations have designed and used satisfactory, quantitative, quality metrics.
- Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.
- The factors involved in bug severity are:
    1. **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.
    2. **Context and Application Dependency:** Severity depends on the context and the application in which it is used.
    3. **Creating Culture Dependency:** What's important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games software vendors.
    4. **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.
    5. **The software development phase:** Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has been around.

4. THE TAXONOMY OF BUGS

4.1. IMPORTANCE OF BUGS: The importance of bugs depends on **frequency, correction cost, installation cost,** and **consequences.**

1. **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
2. **Correction Cost:** What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: **(1)** the cost of discovery **(2)** the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.
3. **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.
4. **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

Importance= ($) = Frequency* (Correction cost + Installation cost + Consequential cost)

4.2. CONSEQUENCES OF BUGS:

The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:

1. **Mild:** Aesthetic bug such as misspelled output or mal-aligned print-out.
2. **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
3. **Annoying:** The system's behaviour because of the bug is dehumanizing. ***E.g.*** Names are truncated or arbitrarily modified.
4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM won't give you money. My credit card is declared invalid.
5. **Serious:** Losing track of transactions & transaction events. Hence accountability is lost.

Software bug Software bug is the error flow Identified in a program or system which causes unexpected out come.

3.4. TESTS:

Tests are formal procedures, Inputs must be prepared, Outcomes should predicted, tests should be documented, commands need to be executed, and results are to be observed. ***All these errors are subjected to error***

There are three distinct kinds of testing on a typical software system. They are:

1. **UNIT/COMPONENT TESTING.**
2. **INTEGRATION TESTING**
3. **SYSTEM TESTING .**

1. Unit / Component Testing:

- A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc.
- **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure does not match the intended design structure.
- A **Component** is an integrated aggregate of one or more units.
- **Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.

2. Integration Testing:-

- **Integration** is the process by which components are aggregated to create larger components.
- **Integration Testing** is testing done to show that even though the components were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.

3. System Testing:

- A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components.
- It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

3.2. PROGRAM Model:

- Most programs are too complicated to understand in detail.
- The concept of the program is to be simplified in order to test it.
- If simple model of the program does not explain the unexpected behaviour, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

3.3. BUGS Model:

- Categorize the bugs as initialization, call sequence, wrong variable etc..
- An incorrect spec. may lead us to mistake for a program bug.

There are 9 Hypotheses regarding Bugs.

1. **Benign Bug Hypothesis:** The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)
2. **Bug Locality Hypothesis:** The belief that a bug discovered with in a component effects only that component's behaviour.
3. **Control Bug Dominance:** The belief those errors in the control structures (if, switch etc) of programs dominate the bugs.
4. **Code / Data Separation:** The belief that bugs respect the separation of code and data.
5. **Lingua Salvator Est:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
6. **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
7. **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
8. **Sadism Suffices:** The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques.
9. **Angelic Testers:** The beliefs that testers are better at test design than programmers are at code design.

3. MODEL FOR TESTING:

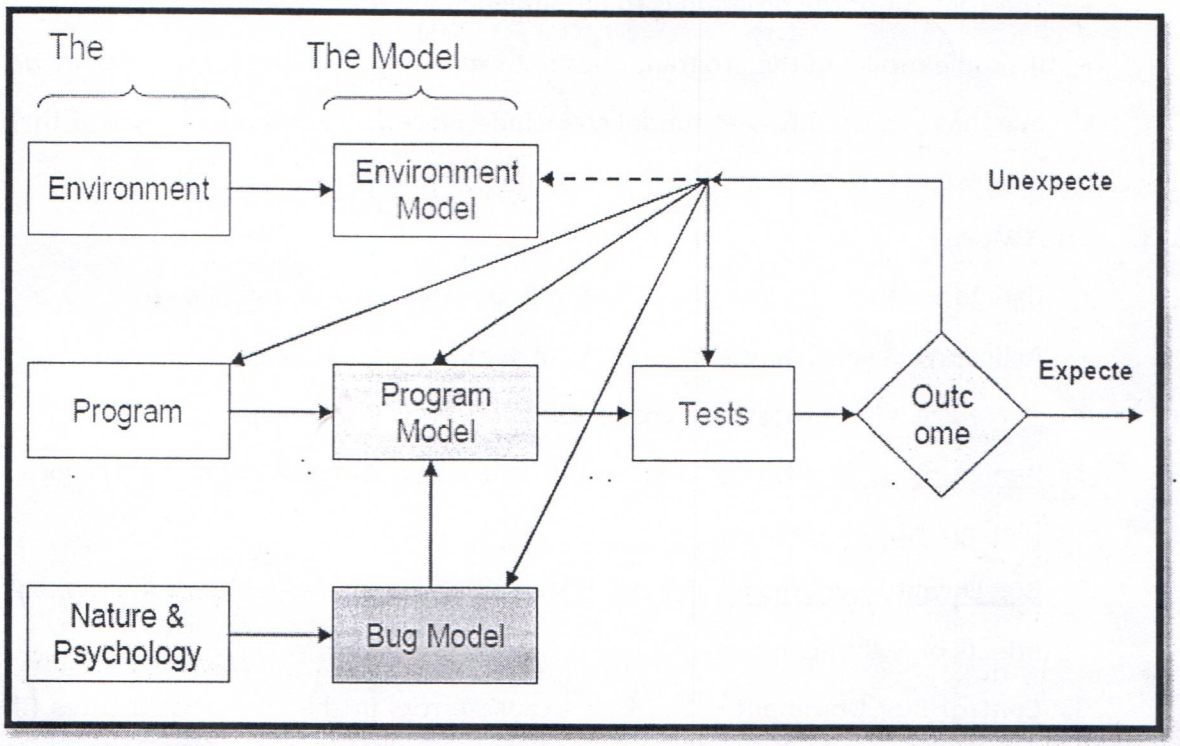


Figure: A Model for Testing

Above figure is a model of testing process. It includes three models:

1. **Environment model**
2. **Program model**
3. **Bug model**

3.1. ENVIRONMENT Model:

- A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.
- The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.
- Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

2.5. Programming in SMALL Vs programming in BIG

SMALL	BIG
More efficiently done by informal, intuitive means and lack of formality – if it's done by 1 or 2 persons for small & intelligent user population.	A large # of programmers & large # of components.
Done for e.g., for oneself, for one's office or for the institute.	Program size implies non-linear effects (on complexity, bugs, effort, rework quality).
Complete test coverage is easily done.	Acceptance level could be: Test coverage of 100% for unit tests and for overall tests ≥ 80%.

2.6. Buyer Vs Builder

- **Builder**: designs for & is accountable to the **Buyer**.
- **Buyer:** Pays for the **system** and hopes to get profits from the services to the **User**.
- **User**: is the ultimate beneficiary of the **system**. User's interests are guarded by the **Tester**.
- **Tester**: works towards the destruction of the software. The tester tests the software in the interests of the user & the operator.
- **Operator**: The operator lives with the mistakes of the builder, murky specs of Buyer, oversights of Tester and the complaints of User.

2.3. Designer Vs Tester

Completely separated in black box testing. Unit testing may be done by either. During functional testing, the designer and the tester are probably different persons

Designer	Tester
Test designer is the person who designs the test	Tester is the person who actually tests the code.
Tests designed by designers are more oriented towards structural testing and are limited to its limitations.	With knowledge about internal test design, the tester can eliminate useless tests, optimize & do an efficient test design
Likely to be biased	Tests designed by independent testers are biasfree.
Tries to do the job in simplest & cleanest way, trying to reduce the complexity.	Tester needs to suspicious, uncompromising, hostile and obsessed with destroying program.

2.4. Modularity (Design) Vs Efficiency

Modularity	Efficiency
Smaller the component easier to understand.	Implies more number of components & hence more no of interfaces increase complexity & reduce efficiency (=> more bugs likely)
Small components/modules are repeatable independently with less rework (to check if a bug is fixed).	Higher efficiency at module level, when a bug occurs with small components.
Microscopic test cases need individual Set ups with data, systems & the software. Hence can have bugs.	More no of test cases implies higher possibility of bugs in test cases. Implies more rework and hence less efficiency with microscopic test cases
Easier to design large modules & smaller interfaces at a higher level.	Less complex & efficient.

TESTING	DEBUGGING
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intuitive leaps, experimentation and freedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

2.2. Functional Vs Structural Testing

Tests can de designed from a functional or a structural point of view

- **Functional Testing:** Treats a program as a **black box**. Outputs are verified for conformance to specifications from user's point of view.
- **Structural Testing:** Looks at the implementation details: **programming style, control method, source language, database** & **coding details.**

.Both are useful, have limitations and target different kind of bugs. Functional tests can detect all bugs in principle, but would take infinite amount of time. Structural tests are inherently finite, but cannot detect all bugs.

2. **Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job.
3. **Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.
4. **Development Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

2. DICHOTOMIES: [differences]

It is the division of important terms related to testing into two especially mutually exclusive or contradictory groups or entities. There is dichotomy between theory and practice.

The Six of them are:

1. ***Testing & Debugging***
2. ***Functional Vs Structural Testing***
3. ***Designer Vs Tester***
4. ***Modularity (Design) Vs Efficiency***
5. ***Programming in SMALL Vs programming in BIG***
6. ***Buyer Vs Builder***

2.1. Testing Versus Debugging:

- The Purpose of testing is to show that **a program has bugs.**
- The purpose of Debugging is to find the **error or misconception** that led to the program's failure and to design and implement the program changes that correct the error.
- **Debugging** usually follows **testing**, but they differ as to goals, methods and most important psychology.

1.3. Phases in a tester's life:-

There are **5 Phases** in Tester life:-

Phase 0: says no difference between debugging & testing

- Today, it's a barrier to good testing & quality software.

Phase 1: says Testing is to show that the software works

- A failed test shows software does not work, even if many tests pass.
- Objective not achievable.

Phase 2: says Software does not work

- One failed test proves that.
- Tests are to be redesigned to test corrected software.
- But we do not know when to stop testing.

Phase 3: says Test for Risk Reduction

- We apply principles of statistical quality control.
- Our perception of the software quality changes – when a test passes/fails.
- Consequently, perception of product Risk reduces.
- Release the product when the Risk is under a predetermined limit.

Phase 4: A state of mind regarding "What testing can do & cannot do. What makes software testable".

- Applying this knowledge reduces amount of testing.
- Testable software reduces effort
- Testable software has less bugs than the code hard to test

1.4. Test Design:

We know that the software code must be designed and tested. Tests should be properly designed and tested before applying it to the actual code.

There are approaches other than testing to create better software.

Methods other than testing include:

0. **Inspection Methods:** Methods like walkthroughs, desk checking, formal inspections and code reading appear to be as effective as testing but the bugs caught do not completely overlap.
1. **Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.

Software Testing:-

Software testing is a process of executing a program or application with the intent of finding the software bugs.

Testing is verification against given specifications. For **software**, it is verification of functionality of a software product by executing the software, for conformance to the given specifications.

Software Testing Methodology:-

Software Testing Methodology is defined as strategies and testing types used to certify that the Application under Test meets client expectations.

Test Methodologies include functional and non-functional testing to validate the AUT.

1. PURPOSE OF TESTING:

1.1. Productivity and Quality in software:

→ In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.

→ If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.

→ Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing.

→ The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.

→ For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.

1.2. Goals for Testing:- Primary goal of Testing: Bug Prevention

- **Testing** and **Test Design** are parts of quality assurance should also focus on bug prevention.
- Test design and tests should provide clear diagnosis so that bugs can be easily corrected. If a bug is prevented, the corresponding rework is saved.
- *"A prevented bug is better than a detected and corrected bug".*

K. Hari

Elective -2: SOFTWARE TESTING METHODOLOGIES

UNIT-I

Introduction: Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of Bugs.

Flow Graphs and Path testing: Basics concepts of path testing, predicates, path predicates and Achievable paths, path sensitizing, path instrumentation, application of path testing.

UNIT-II

Transaction Flow Testing: Transaction flow, transaction flow testing techniques.

Dataflow testing: Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.

UNIT-III

Domain Testing: domains and paths, Nice & ugly domains, domain testing domains and interfaces testing, domain and interface testing, domains and testability.

UNIT-IV

Paths, Path products and Regular Expressions: Path products & path expression, reduction procedure, Applications, regular expressions & flow anomaly detection.

Logic Based Testing: Overview, decision tables, path expressions kv charts, specifications.

UNIT-V

State, State Graphs and Transition testing: State graphs, good & bad state graphs state testing, Testability tips.

Graph Matrices and Application: Motivational overview, matrix of graph, relations, power of a matrix, Node reduction algorithm, building tools. (Student should be given an exposure to a tool like J Meter or Win runner.)

Reference Books

1. Software Testing techniques –Baris Beizerm Dreamtech, Second edition.

2. Software Testing Tools – Dr. K.V.K.K. Prasad, Dreamtech.