Purpose of testing

* Testing is identifying the errors/bugs and checking against various test cases so as to check whether the program/software will work according to users requirements or not.
* Myth: It is beleived by developers who were expert in particular domain, that they could write error code.
  reality: But even in the code in written by expert contains errors/bugs.
* Generally it is assumed that there would be 1-3 bugs/errors for every 100 lines of code

Productivity & quality in s/w

- If the cost of rework exceeds 2% in smaller projects and if cost of rework exceeds 80% in big companies then the product must be destroyed of original cost
- only if above condition is met, then quality of product increases

Goals of testing

A prevented bug is better than detected & corrected bug

- Testing
  - Test design
  - Testing
- prevention of bugs is not possible by test designer or testers but possible for designers.
- Secondary: discovery of bugs.
- Previous documentations help in preventing the similar mistakes to be made again

Phases of Tester's mental life

Phase 0: (until - 1956) - thinking - debugging
Phase 1: (1957 - 1978) - demonstration - s/w works oriented
Phase 2: (1979 - 1982) - destruction - s/w dosen't work
Phase 3: (1983 - 1988) - risk reduction - evaluation
Phase 4: (1989 - 2000) - Prevention -

* until 1956 thinking and debugging are considered as testing under one name only
* The goal of software (1957 - 1978) was, there may be any number of bugs but finally a working product is developed called demonstration {This failed because the probability of showing software works decreases as testing increases}
* In 1983 - 1988, even there are a few bugs we cannot debug them (as one bug correction leads to another). These softwares come under this phase (The purpose of these testing is to show the s/w doesn't work)
* An acceptable product is a product which is accepted by customer even though there are some bugs here and there.
* Phase - 4, mostly type checking errors are present, which are rectified.

Testing isn't everything

* even without testing, we can check the quality of software {other approaches than testing to create better software}

① Inspection method: which involves formal inspections. we check whether the code meets the rules and regulations are not. In formal inspection, we check & verify the documentation after each phase of development [desk checking & code reading]

② Design style: The main objective of the product should be reflected in the design, which can be understood by any stakeholder. The features in the design must be unambiguous [testability, openness & clarity prevents bugs]

③ Statastic Analysis design: (compilers) whether the interpretation task which is done manually by programmers is efficient or not. Compilers were not existing at that time, it took more effort and accuracy in interpreting the code line by line. But the manually interpeting is very less efficient [But now compiler have taken programmers job]

④ language: If the language supports various features and support it becomes easy to write error free code. [Programmers find new bugs when using new language]

And if the programmer is very good at a language, he can write efficient code

(5) Development methodology & Development environment: The process model we choose and the deployement methods used to deploy programs improve the chance to rectify more bugs and produce a efficient working product.

Dichotomies → which involves various approaches for testing

(1) Testing Vs debugging: until 1956 testing & debugging was considered as same, because the computers were not commercialized. But as the market grew for computers testing and debugging were considered as seperate. Test cases were choosen, in the parts where the input is taken and output is generated. Based on system we design which type of testing is needed (like unit testing etc). we use predefined testing procedures and generate the predicted outputs. Debugging starts when the results are generated which were different from the predicted results / expected results. Debugging involves some corrections which may rise another issues. testing is time bound and need to be completed in time, but whereas debugging donot follow a definite procedure and every rectification must be documented as to know why certain things are changed. Testings can be automated, but debugging cannot be automated as one needs to constantly involve in project to debug, but testing can be done with minimal knowledge by anyone.

(2) Functional vs Structural:

Functional testing mainly focuses on identifies the features are working according to the user requirements on not. Functional testing is similar to black box testing. Functional testing needs relatively large time. Structural testing mainly focusses on identifying whether the core implementation is working and also it focuses on implementation details and need small amount of time.

[Functional testing can detect all bugs → takes infinite time
Structural testing can detect real bugs → finite time]

(3) Designers vs Testers:

Test designer | Designers work is to develop the test cases. Testers are one who actually implement the test case and test the code.

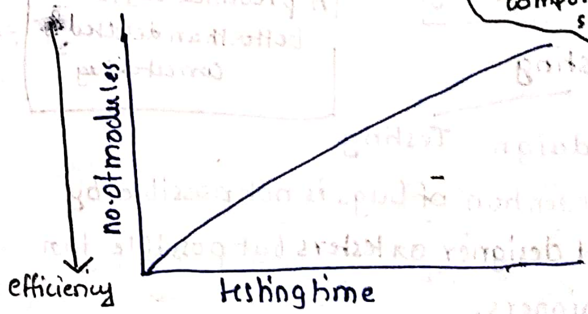
(During functional testing designers tester are different. In functional testing tester & programmer merge into one person)

(4) large vs small

large programs need relatively large amount of time for testing if it contains many modules. whereas small programs need less amount of time as they have lesser modules

(5) modularity vs efficiency

(A module is a discrete well defined, small component of a system)



(6) Buyers vs Builder

Builders are the developers, and buyers are customers. Operators are responsible for the maintainance of the product / software

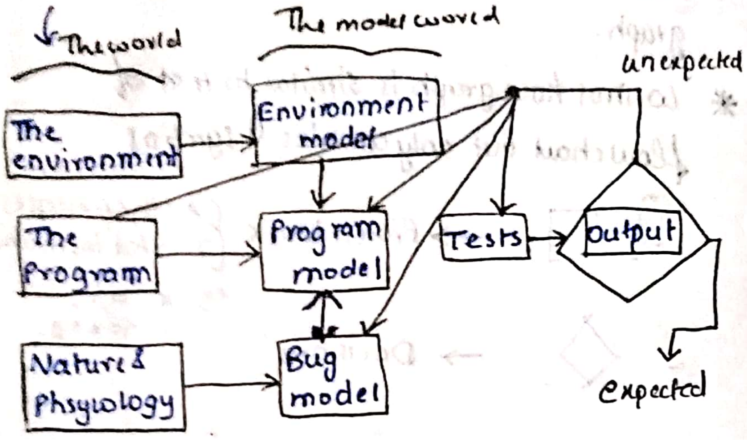
(But if the buyer & builder are same, there will be no accountability)

Roles / users in system → user, Operator, Tester, Builder, Buyer

Model for Testing

→ There will be 3 models for the testing.

These three models can be combinedly called as the model for testing
1. environment model
2. program model
3. bug model



The environment: Software & hardware requirements to [struck out] a program. (& all other additional software)

Program model :- each person would have a different specialization in different languages. So, the modules are created by various persons but the entire project must be developed in a single language mostly.

And the way of representing bugs identified is called optimistics of bugs (bug model)

(And a bug detected can be insidious (deceiving but harmful))

Optimistic ways of bugs
(BBCCLCSAS)

1. Benign bug hypothesis → bug is not a serious issue (they are mild, faulty logical)
2. Bug locality hypothesis → ~~controlling of~~ (Bug of one component donot effect other) One bug donot affect other bug present in the
3. Control dominance → control statement will be dominating the other bugs.
4. Code/data seperation → (Seperation of concerns) localization
5. Lingua salvator est → Identifying syntax & semantic errors
6. Correction abide → Some bugs cannot be corrected because it changes the core logic
7. ~~Sto~~ Silver Bullets → Testing has to be happenned based on values & ethics. → (language grants immunity from bugs)
8. Angelic testes → Testers should be more knowledgable then developers
9. Sadism Suffices. → (test → established) (Tough bugs need methodology & techniques) ~~Unethical arguments made by testers against developers~~

Consequences of Bugs

* It involves the calculation of cost for bug identification based on bug frequency
* correction cost = discoverycost + debugging cost
* installation cost → If any additional components / plugins are needed
* one bug may lead to another bug called consequence of bugs (Testing 2 levels → unit/component, system, integration)
  ↓ based on these 4 parameters we decide which bugs should be resolved first.

> Importance of bug = \$ = Frequency * (correction cost + installation cost + consequence cost)

> If there are no additional components needed and no consequence of bugs, these too can be removed from above formula

Scaling of bugs
(MMADSVEICI)

(Consequence of bugs) (symptoms of bug attend us asthetically)

1. ~~mild~~ mild → A little → symptoms of bug attend us asthetically
2. moderate → will affect the project → outputs are misleading or redundant
3. Annoying → will affecting a component → systemic behaviour is affected by bug
4. Disturbing → will affect a lot → It refuses to handle legitimate transactions (eg: ATM)
5. serious → will cause errors → Accountability is lost as it loses track of its transactions
6. very serious → will cause crashes → This will cause system to do wrong transactions
7. extreme → entire program crashes
8. Intolerable → Program will be non functional → long term unrecoverable corruption
9. casatrophic → Program dosent load → decision to shutdown is not in our hands system fails
10. Infectious. → high level. → that erodes social physical environment

Taxonomy of bugs
→ There is universally no correct way of categorising bugs

1. Requirement, Features & functionality bugs
   1. requirements & specification bugs
2. Features bugs
3. Features Interaction bugs
4. Remedies for requirement specification future bugs & future Interaction bugs.

Taxanomy of the bugs

Requirements, features, functionality bugs
24.3% bugs

- Requirements & Specs → Incompleteness & ambiguity
- Feature bugs → wrong feature(s), missing feature according to specification
- feature Interaction bugs

Remedies
- → short term support
- → long term support

Structural bugs
25.2%

- control & sequence bugs
- logic bugs
- processing bugs
- Initialisation bugs
- dataflow bugs & anomolies

Remedies
→ except for control bugs, logic bugs use domain testing

Data bugs
22.3%

- due to data object → generic databugs → use control tables
- dynamic data vs static data

Dynamic data bugs	static data bugs
Hard to catch	Fixed in form
due to error in storage object initialisation	Appear in source code / database
due to unclean leftover garbage in shared resource	Software to produce object code
data validation, unit testing	compile time processing

- content structure & Attribute related bugs → Remedy:- • Strongly typed languages • local data migrate to global
  - content: arbitrary pattern & bugs
  - structure: shape, size & alignment of data object in memory
  - Attribute: semantics
- Information, parameter & control bugs
  - Information:- dynamic / local single transaction or task
  - Parameter: parameters passed to call
  - control: data used in a control structure for a decision

Prevention → Data validation

coding bugs
9.0%

Interface, Integration & System bugs
10.7%

Testing & Test design bugs
2.8%

Coding bugs

- coding errors
- syntax error
- coding & typographical errors
- documentation bugs

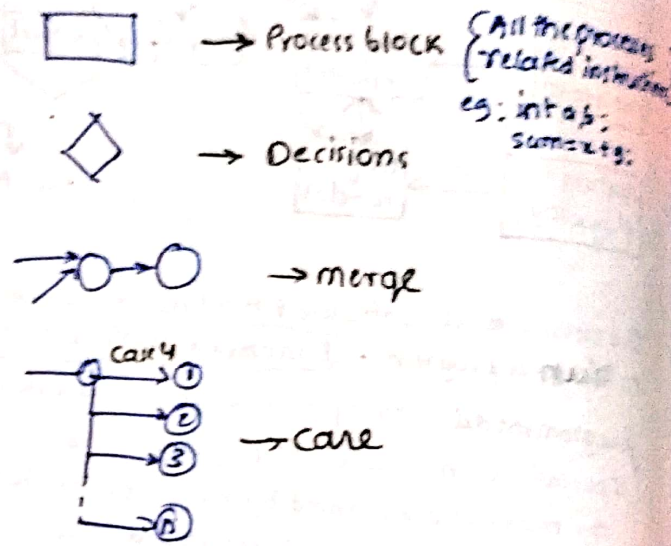
Sol: Inspection

Interface, Integration & System bugs

- external Interface bugs → domain testing
- Internal Interface bugs → domain & Integration testing
- hardware Architecture bugs
  - → centralised H/w interface
  - → due to paging mechanism, Protocols
- Operating System bugs
  - → due to misunderstanding of H/w architecture & interface by OS
  - → same remedies as for H/w architecture
- Software Architecture bugs → subroutines pass through unit & Integration testing without detection of bugs ↓ due to
  - • assuming there are no interrupts
  - • Bypassing data interlocks
  - ↓ stress tests done & good S/w Architecture must be used
- control & sequence bugs
  - → due to ignored timing & starting a process before its prerequisites are met
  - → remedies: good design, specialized internal sequence mechanisms, Path testing is used
- resource management bugs
  - → memory, buffers, longest
  - → due to wrong resource used, deadlock
  - → remedies:- design & path testing
- Integration bugs
  - → detected late in SDLC
  - → due to Inconsistency between components
  - remedies: good integration techniques
- System bugs
  - → Infrequent
  - → bugs in OS, hardware
  - Transaction flow techniques

Path testing → It is one of the oldest Structuring testing methods.

* It is suitable for latest software
* In path testing, mainly in which unit testing is done for a module
* And the flow chart is converted into Control flow graph.
* Control flow graph is similar to that of flow chart but only contains 4 symbols



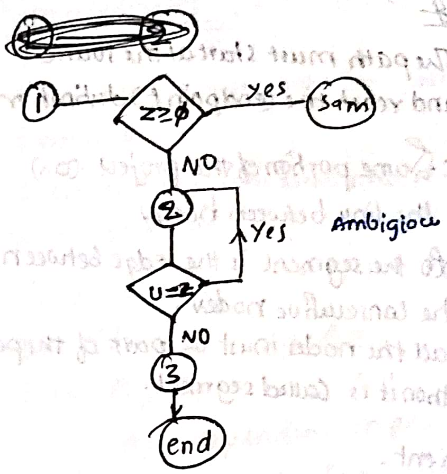
* And we can consider decision & connectors as a node.

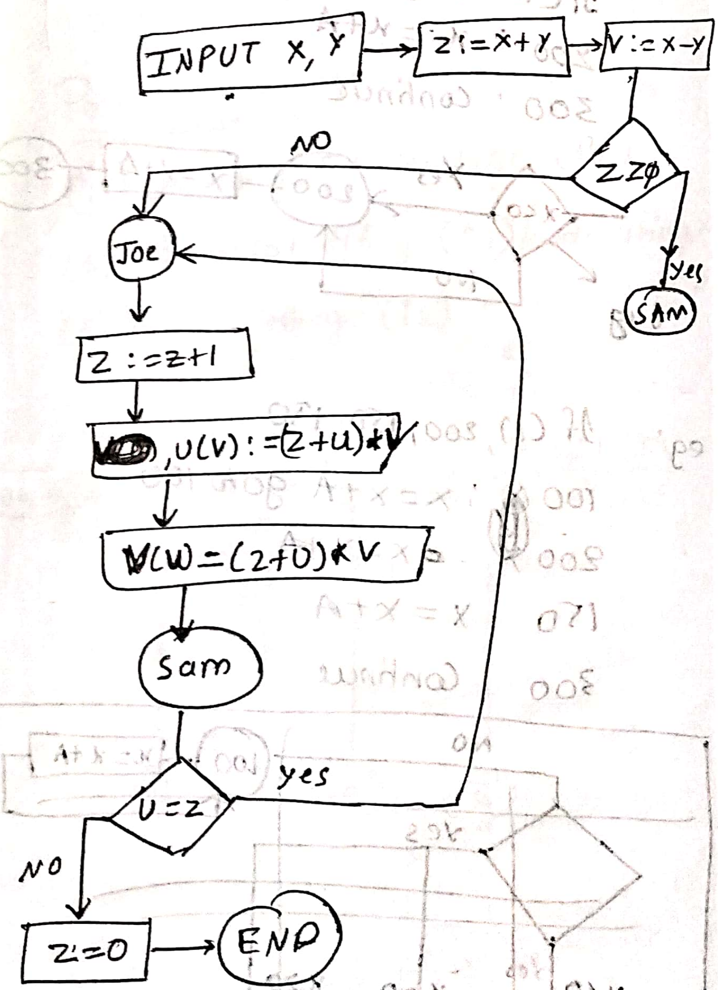
__Write a C program to find reverse of a number__

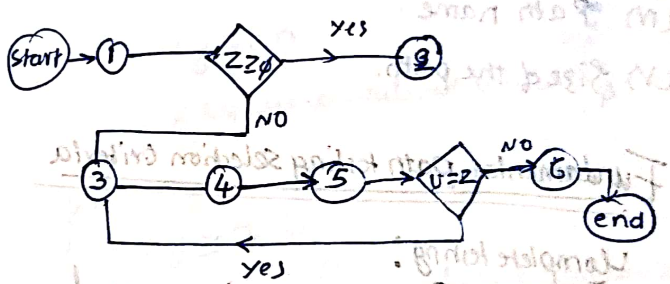
```
main()
{ int n, x; char ch[n]; int d=10;
  read n, x;
  for(i=0; i<n; i++)
  {
     ch[i]= x %.d
     d=d*10;
  }
  char rev[n];
  for(i=0; i<n; i++)
  { ~~for(j=n; j>0; j--)~~
     rev[i] = ch[j];
}
```

ex: INPUT X, Y

```
Z; =X+Y
V :=X − Y
IF Z≥ϕ GOTO SAM
JOE : Z : =Z+1
    V(U), U(V):=(Z+U)*V
SAM : IF U=Z GOTO JOE
    Z : =⊗ U
END
```







one-one instruction flowchart.

Path Testing :-

(i) Path :- The path must start at the source and reach the (endpoint) destination

(ii) Segment : Some portion of the project (or) the line between nodes.
So the segment is the edge between the consecutive nodes.
And all the nodes must be part of the path only then it is called segment.

(iii) Path segment.

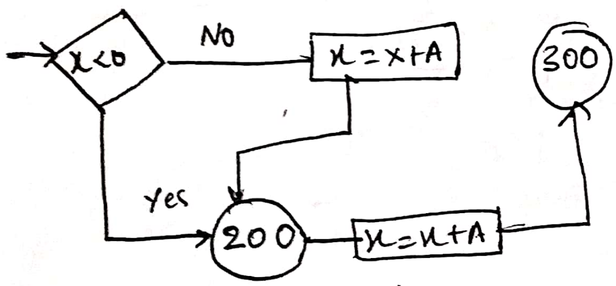
(iv) Path name.

(v) Size of the path.

Fundamental path testing selection Criteria

Complete testing :

1) exercise each & every path in progress from entry to exit.
2) exercise every instruction on statement at least once.
3) exercise branch & case statement, in each direction.



control flow graph for below psuedo code

If (x<0) goto 200
x=x+A
200 : x=x+A
300 : continue

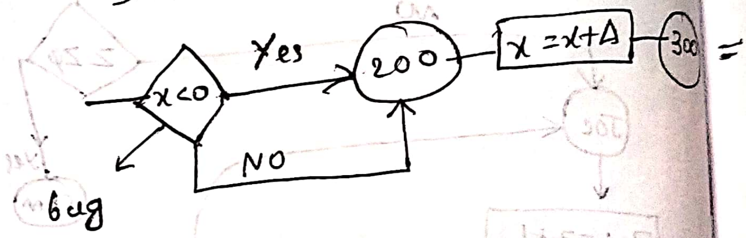
Path1 (NO) = x+2A
Path2 (yes) = x+A

* hidden bug occur when condition leads to the same statement in both cases (T or F) then the bug occur

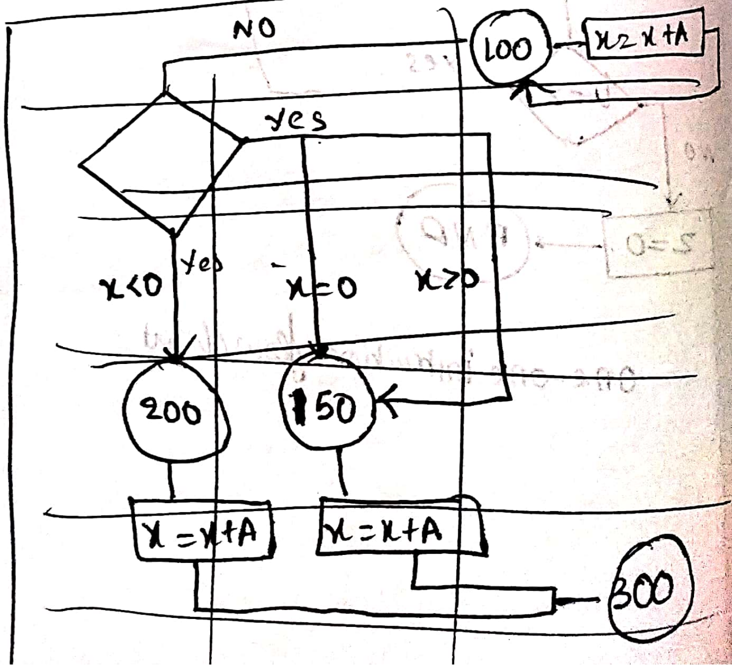
And these types of bugs are not identified by Path testing

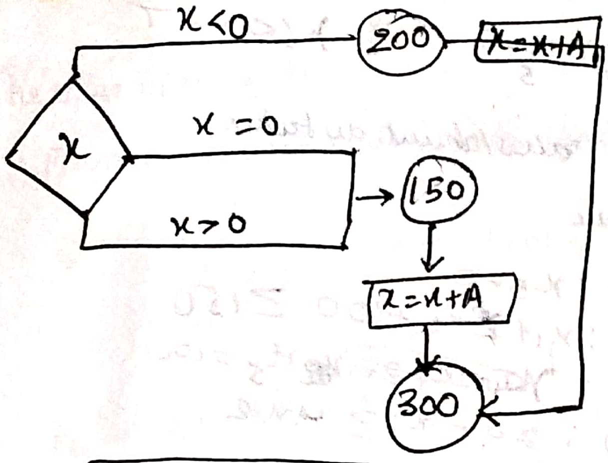
eg;

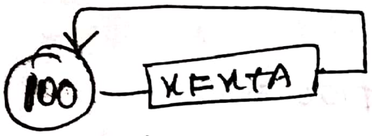
If (x<0) goto 200
200 : x:=x+A
300 : continue



eg: If (x), 200, 150, 150
100 : x=x+A goto 100
200 : x=x+A
150 x=x+A
300 continue



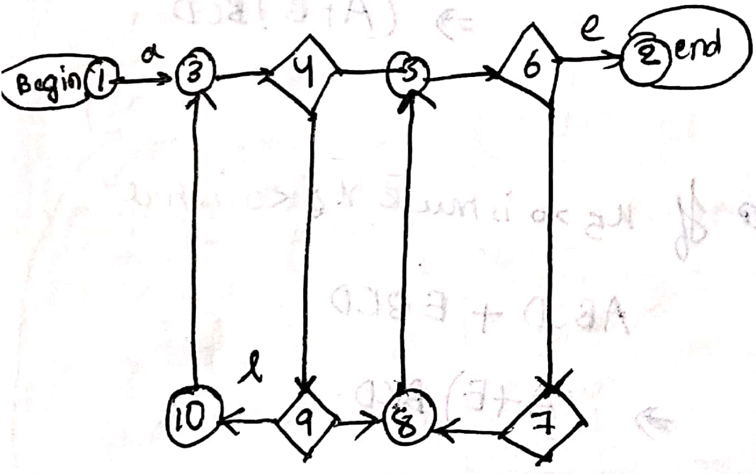




↓
It does not statisfies 2nd statement & it is an infinite loop bug.

Path testing criteria :-

(1) Path testing / Path coverage (P_α)
(2) Segment testing (P_1) / node testing (C_1)
(3) Branch testing (P_2) (C_2)



Path predicates :-

> mapping all input values into array is called Input vector.
> expression that is to be evaluated is called as the predicate

> If x=0 do A1
else if x=1 do A2
else A3

∴ A3 is dependent on predicate x=1
and x=0 is the Independent predicate.

x+y=90
∴ x & y are co-related variables.

y=2
If x+y ≥ 7
then x and y are co-related variables

x=2
If y>1
then x & y are Independent Predicates

If atleast one variable is common in two predicates, then they are called co-related predicates

Predicate Interpretation :

If $x_1 + 7 \geq 0$ — T
{ If $x_1 + y \geq 25$
$x_1 = x + 6$ — T
}

Contro

> If the result of predicate depends upon two variables, then that is called as two place predicate.

> we also need to identify the variables that depends upon the process of execution (condn), which is called process Interpretation

"Predicate interpretation is symbolic substitution of operators & values in the predicate"

Predicate expressions

$x_1 + 2x_2 + 100 \geq 150$
$x_3 = 100$
$x_4 - x_1 \geq 4x_2$

} Three Independent Statements (or) Three predicates

↓

$x_3 = 100$ is obsolete as it cannot be substituted in any statement further after.

> Consider

If. $x_5 > 0$. OR. If $x_6 \leq 0$ Then

$x_1 + 2x_2 + 100 \geq 150$
$x_3 = 100$
$x_4 - x_1 \geq 4x_2$

Case 1: $x_5 > 0$ - T & $x_6 < 0$ - F

Bat on whole T

∴ $x_1 + 2x_2 + 100 \geq 150$
$x_3 = 100$
$x_4 - x_1 \geq 4x_2$

Case 2: $x_5 > 0$ - F & $x_6 < 0$ - T

all statements are taken

Hence

A : $x_5 > 0$
B : $x_1 + 2x_2 + 100 \geq 150$ ~~$x_4 - x_1 \geq 4x_2$~~ $x_3 = 100$
C :
D : $x_4 - x_1 \geq 4x_2$
E : $x_6 < 0$
B : $x_1 + 2x_2 + 100 \geq 150$
C : $x_3 = 100$
D : $x_4 - x_1 \geq 4x_2$

- If both cases are false, no execution
- If $x_5 > 0$ is false & $x_6 < 0$ is True

∴ $\bar{A}BCD + EBCD \Rightarrow (\bar{A}+E)BCD$

- If $x_5 \geq 0$ is True & $x_6 < 0$ is false

~~ABCD~~ $ABCD + \bar{E}BCD$

$\Rightarrow (A+\bar{E})BCD$

- If $x_5 > 0$ is true & $x_6 < 0$ is true

$ABCD + EBCD$

$\Rightarrow (A+E)BCD$.

Path sensitization

Checking whether the path has entry and exit points

P_c → Path coverage

If (Path has loops) 100% Path coverage is not possible

else (simple statements) 100% Path coverage is possible

The entire summary from

Program → flow chart → Control flow graph
↓
Paths identifying
↓
Predicates

P_1 → segment coverage (C_1)
P_2 → Path / link coverage (C_2)

Testing Blindness → Testing done unnecessarily (or) evaluation Testing not identifying bugs

If x=2 goto A, B
A: Do something SAM
B: Do something else SAM
SAM: X is good

If we do not arrive at Path exit meaningfully i.e, even if conditions truth value is same or different, the control ends at same place

Testing blindness

① Allignment Blindness

Correct	Buggy
x = 2	x = 2
If x+y > 7	If y > 7

* In this, the predicate is wrong but may appear as (correct one) working. (i.e, above, x = 2 assignment in buggy version is unused)

② equality blindness

If one predicate is deciding the value of another predicate and the result lands as wrong.

e.g.

Correct	Buggy
If A = 3	If A = 3
If A+B = 6	If B = 6
3+B = 6	3+B = 6

↓
In here B is not declared

③ Self Blindness

> If the predicate itself is causing the problems.

Correct	Buggy
x = A	x = A
If x+1 ≥ 0	If x+A+2 > 0

∴ Buggy predicate is the multiple of the ~~correct~~ correct one

then
A+A+2 > 0
2A+2 > 0
2(A+1) > 0
↓
multiple of actual statement (x+1 ≥ 0) in correct version

Path sensitization: We are going to identify whether the particular path is correct or not.

Path sensitization	Heuristic Predicate for the Path sensitization
① have a path (already)	① Predicate defines the path
② we identify Path coverage C_1 + C_2	② And the path contains independent & unco-related predicates and that needs to be covered.
③ we can generate control flow graph & identify different paths	

④ we will be identifying the predicates present in the particular path.

⑤ using the predicates, (If more than one predicate present in the control flow graph) and we can form the compound predicates expression & we simplify that expression.

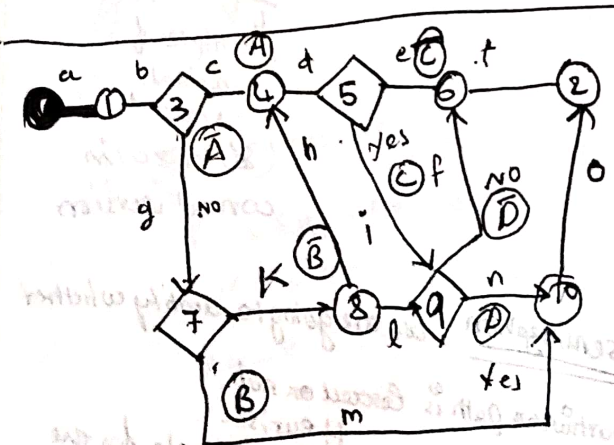
⑥ we need to document every input for every case (ex: A in (A+D)) and form a 1-D array which is the input vector and then that path is called as the achievable path. if we can identify input for all predicates

③ case(i): Independent predicates
case (ii) co-related predicates
case(iii) both

based on the path we may choose one of the iii case
and if we dont get achievable path, then we will choose another path using ③ ways

① Independent & uncorrelated
② correlated & independent
③ uncorrelated & dependent
④ co-related & dependent

If we consider 4 case, more test cases will be generated relatively



Path name	Predicates
abcdef	A $\bar{C}$
abgklno	$\bar{A}$ $\bar{B}$ D
abcdfno	A C D
abcdffo	A C $\bar{D}$

8 Combinations

Path instrumentation:

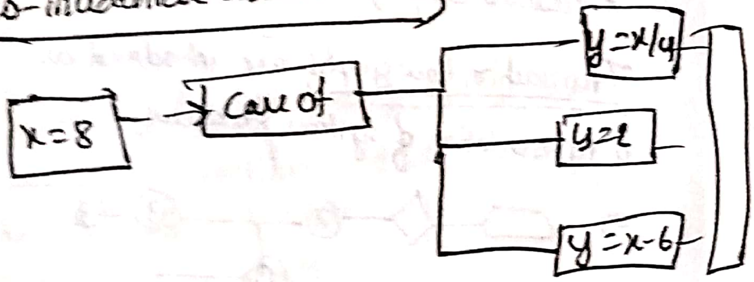
Achieving the outcome in path by using different value is called path instrumentation

a+b = 5

0+5
5+0
1+4
4+1
2+3
3+2

→ Instrumentation

Co-incidental correctness



> Their is not a correct (valid method fetching
> different expressions → same output
> But we dont know which will give the output

There are 3 different types

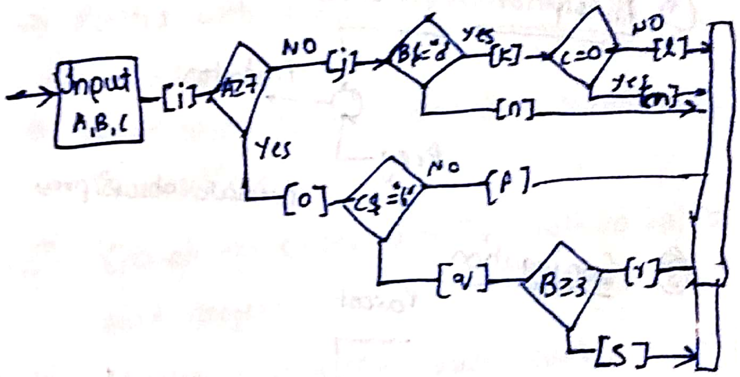
1. Interpretive Tracer program
2. link marker
3. link count.

To trace the bugs naming of each & every link (edge in diagram)

Path Instrumentation techniques

1. Tracer program (not suggestable)
2. Link marker
3. Link marker Counter

as for each line the machine generate a comment line (Italic)

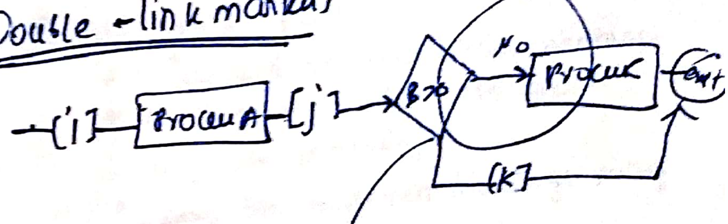


ijkl . . .

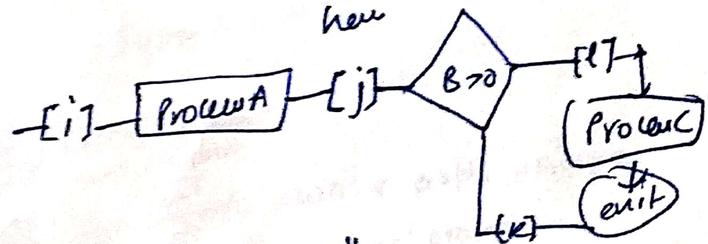
but ijn, ijp are shortest paths

> Bugs can easily affect in single link markers by 3rd parties.

Double - link markers



here no marker is placed here we can add a link here

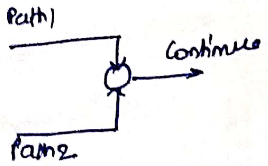


we need to add a link before and after (loop) descision wherever missing

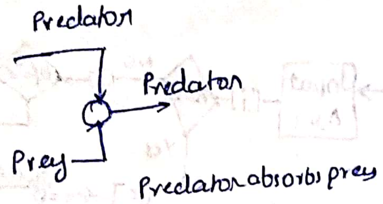
~~Assignment : Applications of path testing~~

Interpretation of mergers (Junctions)

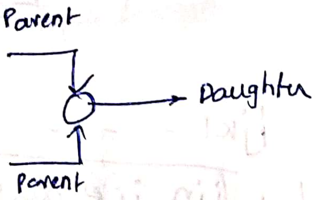
① junction



② Absorption



③ Conjugation



Transaction → "A unit of work seen from a system user's point of view."

→ "A transaction begins with birth as a result of some external act"

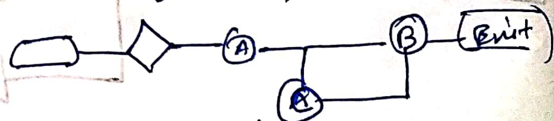
→ A transaction may consist of Sequence of operations.

examples of a transaction

- Accept input (tentative birth)
- validate input (birth)
- Transmit acknowledgement
- Search
- request
- accept
- validate
- process
- update
- cleanup (death)

Transaction flow graphs: are introduced as a representation of system's processing.

eg :



usage: Transaction flow graphs are used for specifying requirements of complicated systems

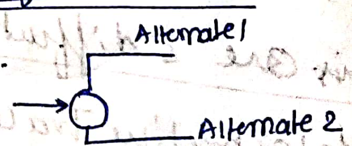
→ loops are infrequent in here

→ The most common loop you see is a retry after an error.

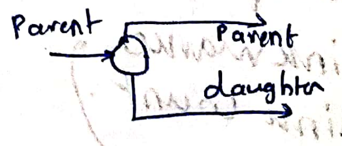
Interpretations of decision symbol (births)

There are 3

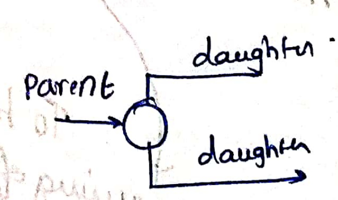
① Decision



② Biosis



③ mitosis



Transaction flow structure — UNIT-II

* Imagine we are making transaction flow graph for the process (methodology), but not the code because they may be unstructuredness

reasons for unstructuredness :-

1. Process involves human users
2. Part of flow from external systems
3. errors, failures, malfunctions & recovery actions.
4. Transaction count, complexity and customer environment.
5. new transactions & modifications
6. Approximation to reality
   - attempt to structure
   -

> Developing a transaction flow graph without the usage of mitosis, biosis, absorption and Conjugation makes it simple, which is equalent to the control flow graph

> The above reasons are reasons for bad structures of the making of transaction flow graph.

Transaction flow testing — steps

- First build / obtain transaction flows
  - > represent explicitly (for user convinence)
  - > design details in the main Tr-flows (documentation of TRFG)
  - > create from PDL (Program description language)
  - > HIPO charts & petrinet representations
- Then trace the Transaction.

Transaction flow testing Techniques

(I) Get the Transaction flows
→ from documentation

(II) Inspections, reviews and walkthroughs

⇒ Start from the preliminary design

1. Conducting walkthroughs
   - discuss enough transaction types (98% Transactions)
   - user needs & functional terms (design independent)
   - Traceability to requirements
2. Design tests for C1+C2 Coverage
3. Additional Coverage (> C1+C2)

→ Paths with loops, extreme values, domain boundaries.
→ wierd cases, long & potentially troublesome Transactions

4. Design test cases for Transaction splits and mergers.
5. Publish selected Test paths early
6. Buyer's Acceptance – functional & acceptance tests

(II) Path selection

1. covering set (C1+C2) of functionality Sensitive Transactions
2. Add difficult paths
   - review with designers & implementors
   - exposure of interface problems & duplicated processing
   - very few implementation bugs may remain.

Transaction-flow path coverage set belongs in System feature tests

(III) Sensitization

1. functionally sensitive paths – simple
2. error, exception, external protocol interface paths – difficult

Data-Flow testing:-

- we use control flow graph.
- In this testing, we are going to take the different data objects and its sequential execution.
- Based on Architecture there are two data flow machines
  - → Von-neuman Architecture
  - → MIMD (multi-instruction-Multiple data objects)

> In von-neuman Sequential execution of Instructions take place

> In von-neuman Architecture the Program & data stays in same memory

> In von-neuman Architecture

Fetch the Instructions from memory
↓
Interpret the instructions
↓
Fetch the operands from memory
↓
Processing or execute the instruction
↓
Increment the program counter
↓
goto starting step

→ In MIMD, we can execute various instructions fast and many at a time (also depends on the compiler)

Data flow graph

- The flow of an data object in different instruction level & data functions is called data flow
- In data flow, it contains nodes & directed edges

Data flow graph consists of nodes & directed links

---

for given L, t and d (cal.) Z & Hc

Cos L

PAR DO: CSL = cos L, SNT = sin t
CTL = cos L, CST = cos t
SNL = sin L, TNT = tan t

END PAR:

PAR DO: CSC = CSL * SNT
TNM = CTL * CST
TZF = − SNL * TNT

END PAR:

PAR DO: $C = \cos^{-1} CSC$
$M = \tan^{-1} TNM$
$ZPF = \tan^{-1} TZF$

END PAR:

PAR DO:
MPD = M+d
SNC = sin C

END PAR

PAR DO: TMD = tan MPD
SMP = Sin MPD
end PAR

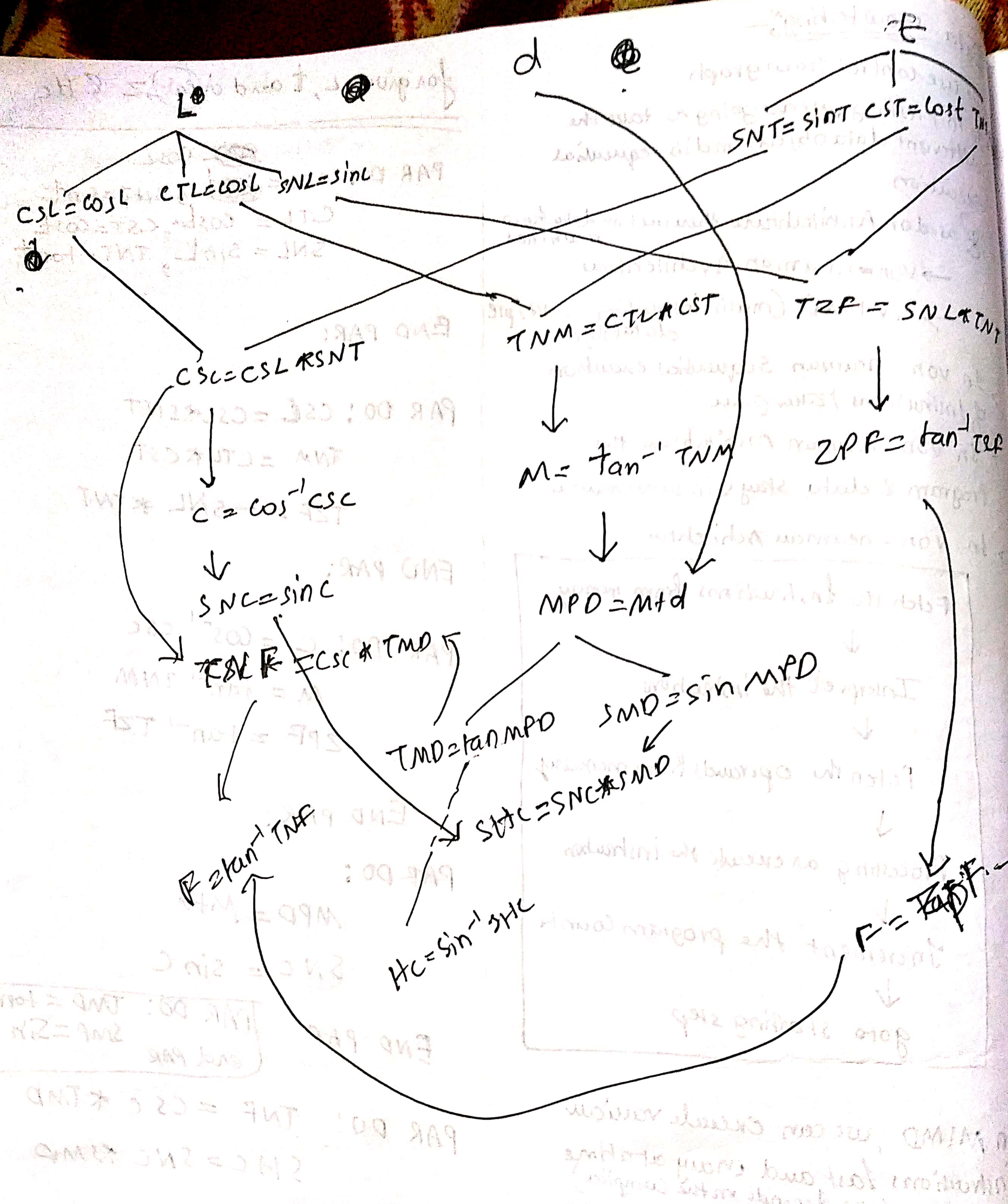
PAR DO: TNF = CSC * TMD
SHC = SNC * SMP

END PAR:

PAR DO: $Hc = \sin^{-1} SHC$
$F = \tan^{-1} TNF$

END PAR

Z = ZPF − F



L
d
t
CSL=cosL
CTL=cosL
SNL=sinL
SNT=sinT
CST=cost
CSC=CSL*SNT
TNM=CTL*CST
TZF=SNL*TNT
C=cos^-1 CSC
M=tan^-1 TNM
ZPF=tan^-1 TZF
SNC=sinC
MPD=M+d
TNF=CSC*TMD
TMD=tanMPD
SMD=sin MPD
SHC=SNC*SMD
F=tan^-1 TNF
HC=sin^-1 SHC

Data object state & usage

data objects can be

1. defined [d]
   - implicit object
   - object defined explicitly
   - something pushed on stack
   - file opened
   - record written
2. killed (or) undefined [k]
   - object is killed
   - stack is popped
   - return from ...
   - A = 12
   - A = 22
3. usage [u]
   - (c) in calculations
   - (p) used in a predicate

Dataflow Anomaly

An Anomaly is denoted by a two character sequence of actions
- is dependent on application

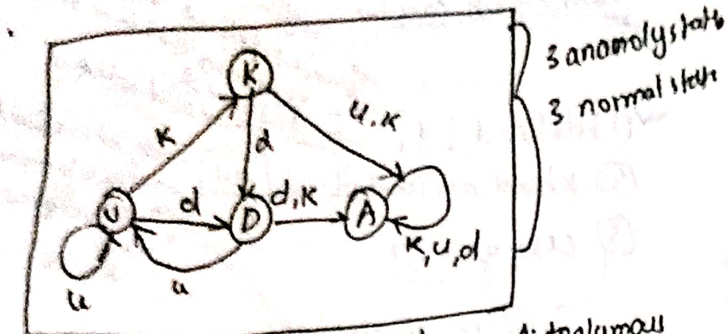
→ nine possible combinations are possible using u, d, k
some are bugs, some are suspicious, some are okay

- dd → suspicious
- dk → bug
- du → normal
- kd → normal
- kk → bug
- ku → bug
- ud → suspicious
- uk → normal
- uu → normal

→ single letter situations, six situations

- ~k → possibly anomalous
- ~d → okay
- ~u → possibly anomalous
- k~ → not anomalous
- d~ → possibly anomalous
- u~ → not anomalous

Forgiving the data flow anomaly graph



A: Anamalous state

① ~~data flow Anamoly graph~~ unforgiving data - flow anamoly flow graph

If once an object reaches anamolous state can never return to a grace state

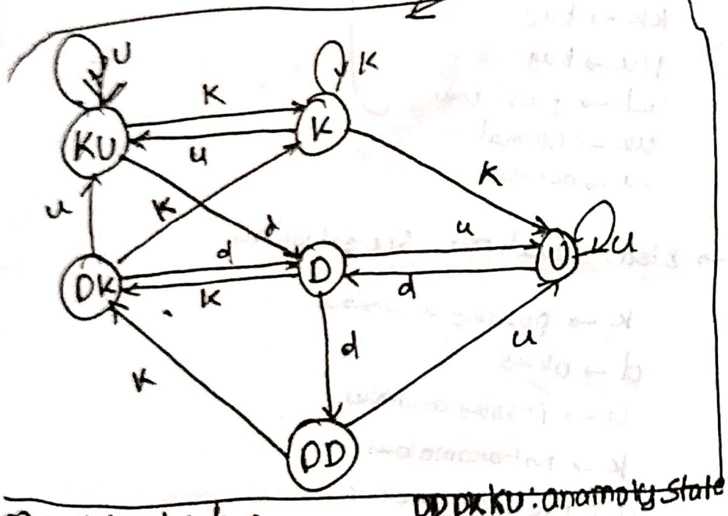
* once an object reaches the anamolous state, it cannot be used again (or) cannot be revoked

u: use state
D: define state
K: kill state

Thus it is called forgiving the data flow anamoly graph

Harry developed an improved version of data flowgraph

→ where even after the data object reaches to the anamoly state, it ~~can~~ can be revoked



② forgiving data flow graph anamoly

In this the redemption is posible from the anamolous state

DD, DK, KU: anamoly states

- KU → Killing & trying to use data object
- DK → data object defined and immediately killed
- DD → If data object is defined twice and not used.

Anamolies depend on
→ language
→ application context
→ state of programmers mind

Static & dynamic data flow anamolies

- Identifying the syntactic & symantic mistakes comes under static analysis
- Identifying intermediate errors while executing the program comes under the dynamic analysis

> using the static language processors, can identify the syntax errors

> Static analysis donot identify all syntax errors with 100% efficiency because of

(i) dead variables. (killed variable) (unsolvable)

when you kill a variable and try to use it again it leads to anamoly (KU)

If you use d(control) data flowgraph which have only one anamoly state then we are unable to identify dead variables so based on the code the compiler should select ① (or) ② method

(ii) Array (anamolies are possible)

Array is considered as a single data object

→ If we are going to consider an single element and using static analysis methods, then they cannot identify the position of array element

(iii) Pointers & records (without execution we cannot determine their emition)

we cannot ~~[illegible]~~ identify a particular record using static analysis methods. and pointers address also cannot be identified by static analysis methods, but is possible with dynamic analysis methods.

(iv) dynamic sub-routine (or) function names in a call

We cannot identify whether the parameters that are going to be passed to a sub routine are declared are not using static analysis method

eg:- inlinood(x, y)

(V) Recovery of Anamolies using alternative DFGs

using Forgiving dataflow anamoly graph we can identify certain bugs using Static analysis methods whereas in above 4 methods we cannot identify the bugs using static analysis method. But we can using dynamic analysis method

(vi) Concurrency Interrupt, System Issues

- the external events like Interrupts when the control transfers to interrupt handler, we cannot identify bugs using the Static analysis methods. whereas we can identify the bugs that are internal.
- using static analysis methods, we cannot predict whether the program can execute in a system without any error or not, system issues include memory issues, Processor issues etc

(VII) False Anomaly :

False anamolies are also not identified by Static analysis methods

So we mainly use dynamic analysis method
But static analysis methods are worth using But demands a huge human testing work

Data flow model :-

rules of constructing data flow model :-

- each and every instruction in the program is considered as a node with an unique name. And the entry and exit points cannot be considered as nodes, As every node must have an inlink & outlink.
- we need to write an intention description of on how we want to use the predicate above the predicate in model
- Combine all the similar nodes in to one node.
- And when we combine the nodes, we need to write operations in order

o—o—o

u,k,d
o———o

- what we write above the link, becomes the weight of the link.

ex:-

```
START
Input a,b,n
Z:=0
If a=1 then Z:=1
goto done1
r:=1 c:=1
Power :
   c:=c+a
   r:=r+1
   If r<=n then goto power
   Z:=(c-1) / (a-1)
Done1:
   Z:=b*z
end.
```

Data Flow testing

Strategies:-

> data flow model is often refered as structural testing strategy.

1. All definition clear paths & variables.
2. loop-free segments (Path segments) (no loops in path)
   - each & every node visited only once.
3. Simple path segment
   - each node is visited atmost twice.
4. All du paths. (du = defined & used)

Dataflow testing strategies

1. All Du-paths (ADUP)
2. All uses (AU) strategy
3. All P-uses / some C-uses (APU) & (SCU) or All C-uses/ some p uses (ACU) (SPU)

(4) All definition strategy (AD)

(5) All p-uses, All-c-uses strategy (APU, ACU)

(1) All Du paths (strongest DFT)

- every du path for every variable for every definition to every use

(2) All uses Strategy

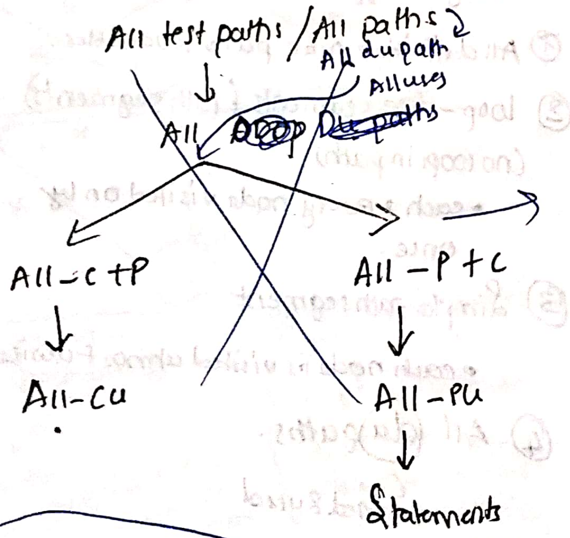
Atleast one definition-clear path segment from every definition of every variable to every one of that definition under some test

(5) (APU+C) | (ACU+P)

(4) cover every definition by atleast one p (or) c

(5) .

⇒ ordering of DFT strategies / control flow representation

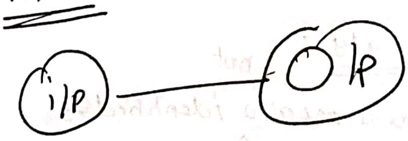


* 90% Path coverage can be acheived when we choose the right.

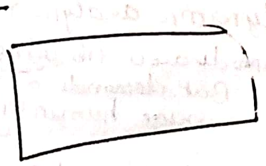
Slicing	Dyeing
(1) we are static analysis. using the slicing we are going to identify the acheivable paths	(1) Dyeing depends upon the dynamic analysis. In runtime we identify the data objects. It is also called as dynamic dyeing
(2) we eliminate the statements which are invalid	(2) we are going to get less coverage than P2 (APU) coverage < P2

Application, Tool, effectiveness (DFT)

DFM 0



DFM 1 - validation



Comparing Natta (NATF 48 B) with
1) Random testing
2) Branch testing
3) All uses

standard

(1)

	No of test cases conducted	bugs found
Random testing	35	93.7
P2	3.8	91.6
AU	11.3	96.[illegible]

The revised category standard

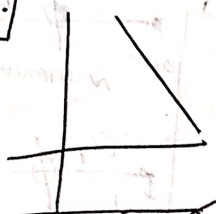
NATF U8A

	No of test case	bugs found
Random testing	100	79.5
P2	34	85.5
AU	84	90.0

② Another scientist speeds (90% data usage / coverage)

③ ASSET Testing system (Released by Weyuker)

decided by linear regression equation.



$A = a + b(d)$ → decisions (or) Predicates in the Program

Atmost (d+1) test cases can be used out of APU, AU, ADPU, ACU

ADUP > AU > APU > ACU > required APU

↓

Bug detection rate

④ Shimeall & Leveson.

$ADUP \sim \frac{1}{2} APU$

$AP \sim AC$

Tools

No tools are present to conduct DFT automatically

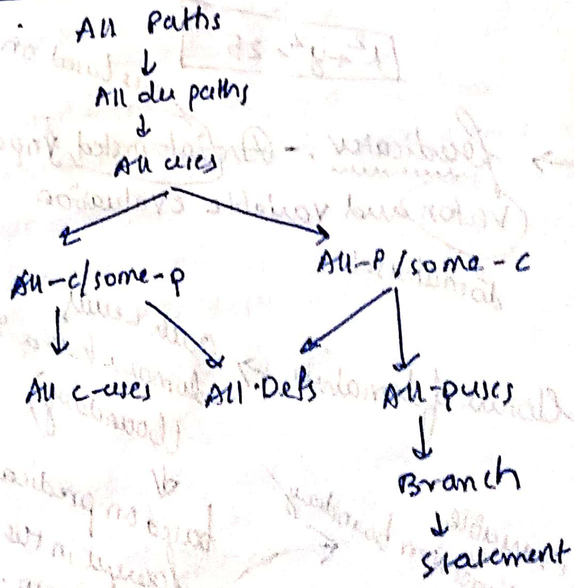
- New compilers are being packed with DFT strategies.

Effectiveness (AU, P1 & P2)

DFT

* Test design is same because in path testing & DFT we use control flow graph.

Coverage (DFT) = Coverage (P2)



Relative strength of structural test strategies

⇒ Domain testing :

- Another name for the domain testing is the partition testing.
- Partitions can be made out of the inputs by categorizing them.

→ we can check also domain boundaries & associated predicates which are used to define them.

→ domain testing do not need any specifications of structural implementation / information of program

→ An achievable path from routine entry to exit.
for each case statement we should have atleast one path

→ Domain can be considered as a set.

→ The mathematical analysis methods of domain testing if it is integrated into compiler, we require less test cases.

$x^2+y^2<25$

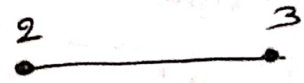
→ Predicates :- Are interpreted input vector and variable evaluation — is based on specification details / information details
domain set

Interpretation not needed
↑
→ data flow graphs
→ CFG
↓
Interpretation needed

→ Closure of domains → each & every domain has a range (boundary)
↓
based on predicates present in the program.
↓
each & every program have atleast one predicate
↓
The boundary of domain can be open or closed

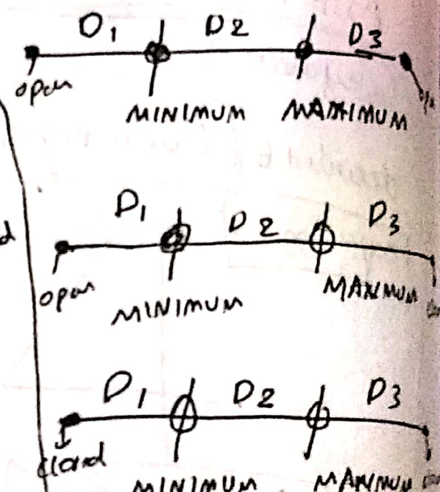
1 variable
↓
1 dimension boundary
↓
n variable
n dimension boundary

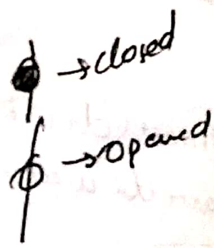
- The boundary point belongs to domain it is closed (2,3) then 2.3 exist within the (2,3) boundary



- The boundary point belongs to some other domain is open

- Consider a variable x under 3 domains as x>0, x=0 & x x<0





Bug Assumption :-

- for each testing you assume what bugs may arise from every testing.

① double zero representation

Some language support +0 & −0
bug

② floating point check for zero

→ This bug occurs, when the floating Point variable is assigned to '0'.

→ ·even in the routines.

→ when subtracted from itself.

③ Contradictory domains :- → two domains having same specification

→ If ambiguity is present in the domain then they are contradictory.

④ Ambiguity of domain:

→ boundary related errors, & the domain is not clearly defined.

→ or if you take a combination of two domains then ambiguity of domain arises ↳ and its combination is not complete

⑤ closure reversal :-

→ it is reverse of predicates

→ it contains the compliments

⑥ Overspecified domain :-

→ we cannot make compliment of particular variables.

→ more constraints, then the domain becomes a null domain

eg:- byte.

⑦ faulty logic

→ predicate analysis of domain is wrong, even if program executes & output is received

eg: 2^x, x^2 → $x=9$

Output incorrect

⑧ Boundary related bugs:-

Boundary related bugs.

Restrictions in domain testing:-

- we make the input vectors classified into different categories.
- If this classified criteria is not followed by any variable then we cannot conduct domain testing
- restrictions are needed whether to conduct domain testing

① Coincidental Correctness: Domain testing cannot be analysed using domain testing, even though the boundaries of domains are clearly specified

- So our program must be free of this problem

② representative output/outcome:

- each and every boundary of domain have atleast one predicate (and that predicate can be simple or complex).
- eg: If x>7

  If x>7 AND y<14

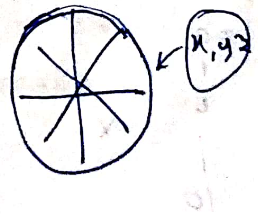
  If y>7 AND y<14

  The machine decides boundary based on some other detailed expression on connective

like

If x=0 And y>7 or y<14

domain



If one part is tested & found valid, it is assumed that all other parts are valid & no testing is d

③ Simple boundaries & Compound Predicates :-

boundary seems as simple.
but have a complex predicate

④ Functional Homogeneity bugs .

even though we have bugs, we dont change the originality of the predicate

$a^x >= 6$ cannot be changed to $a^x >= 6$

If bugs appear

⑤ Linear vector space

- research paper published about DT
- There are linear & non-linear ~~predicates~~
  - equations at predicate level
- In linear equation the output rate was/ successful rate was 99.99 %.
- for non-linear equations at predicate level, we are going to convert into linear equations without change in originality of the predicate.

⑥ loop-free-s/w

- Domain testing is hectic if there are loops present
- In the loops, for the same variable in each increment the boundary of domain is changed

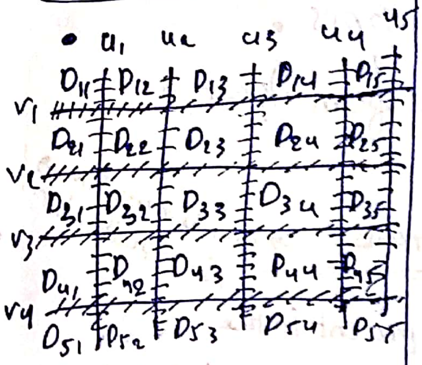
∴ for(i=0; i<=10; i++)

i=0
1
2
|
10

X

∴ Therefore we converts looping structures into simpler statements

Nice domains	ugly domain



eg: 4, 4, 4, 8
4, 5, 6, 7, 8, 9

Domains are classified into:

① Specified domain
They are ambiguous & contradictory

If two domains have same specification then they are inconsistent

② Implemented domains : they never become incomplete & inconsistent because each & every possible input value is checked against all the domain range

If you are not getting the output for inputs for a domain

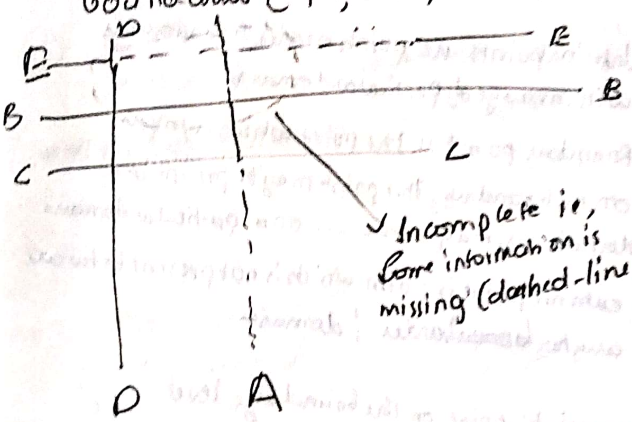
- Domain is not decided by any particular single person

Nice domain properties.

① It contains both linear or non linear Boundaries

② But nice domains only accept the linear Boundaries, if any non-linear boundaries are present then they are converted into linear Boundaries

③ Nice domains support complete boundaries (i.e, $-\infty$, to $+\infty$)



- for complete Boundaries you can use only one set of test case to check all the variables present in the particular Boundary level.
- Nice domains are complete domain
- by conducting one testing, we assume even other path are correct without even testing.

④ Systematic Boundaries are used in Nice domains which describe

$f_1(x) >= k_1$
$f_2(x) >= k_2$
⋮
$f_2(x) >= k_1$

$f_1(x) = g(1, c)$
$f_2(x) = g(2, c)$
⋮
$f_i(x) >= g(i, c)$

decent function

- Systematic boundaries are easy to explain.

eg: entering value for x

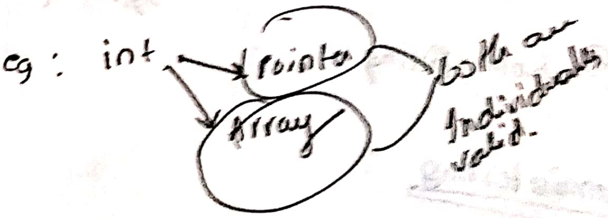
∴ $x_1\ x_2\ x_3\ x_4$ - - -
The x boundary is shared by all other $x_1\ x_2\ x_3$ - - - -
in the above function example

⑤ Orthogonal Boundaries:

- The primitives can be summed up to construct various sets of structures for which each structure is valid & legal.
- The more orthogonal the Domains are the less testing is required

Orthogonality of domains $\propto$ $\frac{1}{\text{test cases required}}$

eg:



⑥ closure consistency:

- Clear Inputs
- clear specifications for domain

Only then we are going to get an achievable path

⑦ Convex:

- Taking two separate entities & making them appear as one entity of a domain then it is convex

(8) Simply connected Boundaries

If the connections between the Boundaries are simple irrespective of no. of domains

If all three properties are satisfied, only then we can call it as nice domain, even if a property is not specified correctly then it becomes an ugly domain

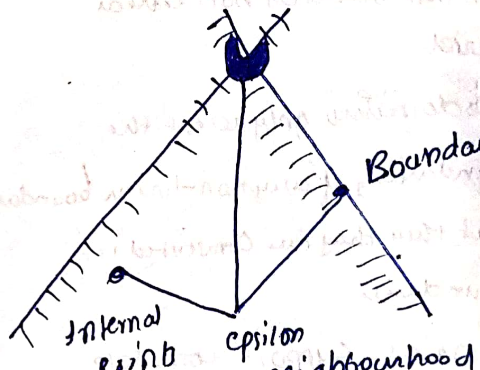
ugly domain properties

1. non-linear Boundaries
2. Contradiction & Ambiguity
3. System Topology / Simplifying system topology
4. holes filling

Domain testing :

* every domain is defined by boundaries.
* test points should be considered near the boundaries / on the boundaries
* Classifying the domain boundaries, and if values are Out of the boundaries defined.
* redundant elimination of boundaries w.r.t domain
* after conducting testing, we should go for post analysis which shows whether an boundary is faulty (or) not
* we need to test boundaries for each Individual Boundary

Domain bugs in domain testing



- Interior points are points nearer to boundary within range of particular domain.
- Boundary point is the point which is taken on the boundary, this point may be present in a domain (or) may not be ... on a particular domain
- extreme point is point which is not present between any two boundaries / domain

onpoint: point on the boundary level

offpoint: Off point is point on the boundary which is not tested currently with respect to closed Boundary

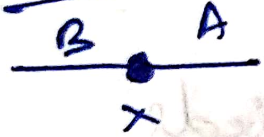
Off point is point on our boundary which is currently being tested with respect to open boundary

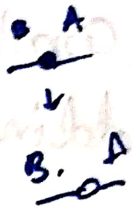
COOOOI
closed off outside
open off Inside

① **closure bugs** → Initially bug evaluated predicate is wrongly evaluated

eg. (x > 0) is intention
↓
but
(x >= 0)

② **Shifted bugs** → moving of the boundary points causes these bugs





③ **Tilted Bugs** → position of domain is slightly changed

④ **Missing Bugs** → missing boundaries

⑤ **extra Bugs** : → extra boundaries

* All the bugs are present in 1D, 2D, 3D domain.

UNIT - III

Path products and path expression

Path expression → is a set of algebraic representations of sets of paths in a graph

↓ A path expression can be converted into a regular expression that can be used to examine structural properties of flow graphs

"flow graphs are the abstract representation of programs"

→ most testing & debugging tools use flow graph analysis techniques ⇒ motivation

"normally flow graphs are used to denote only control flow connectivity"

↓ The simplest weight we can give to a link is a name

↓ using link names as weight, we can derive an equivalent algebraic expression

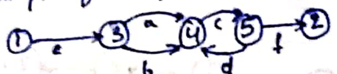
↓ To Trace, you traverse a succession of link names

↓ eg: If you traverse abcd, then the name of that path segment is abcd

↓ The path name is also called as a Path product → Path product

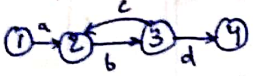
examples of some paths

(1)



eacf, ebcf, ebcf, eacd, ebcd ..etc

(2)



abd, abc, abcbcbd ..etc

Path expression

- Taking a pair of nodes in a graph & set of paths between the node → set of paths must be represented in uppercase letters

on using x = ac, abe
ac + abe
+ denotes 'or'

"Any expression that consists of two successive path segments is conveniently expressed by concatenation or path product"

→ so any expression that consists of two successive path segments is on path segments is on denotes a set of paths between two nodes is called a path expression

Path product rule

If x = abcde y = fghijk
then xy = abcdefghijk (If x is followed by y)

Similarly

aX = aabcde
XaY = abcdeafghijk

If x represents a set of paths on path expression and y also represents another set of paths on path expression

then product is all combinations of x & y paths combined

eg: X = abc + def + ghi
Y = uvw + z

then XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz

"If a link segment (or) name repeats, we express no. of repetitions in terms of power"

$a^1 = a, a^2 = aa, a^3 = aaa, a^4 = aaaa \ldots a^n = a \ldots n$ times

Similarly

$X = abc : X^2 = abcabc : X^3 = abcabcabc$"

** "The path product is not commutative" **

$XY != YX$

** "The path product is associative" **

$A(BC) = (AB)C = ABC$

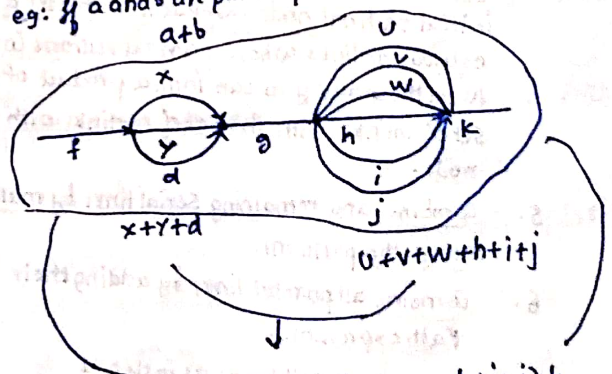
** "The path without any links should also be represented" **

$a^0 = 1$ link
$X^0 = 1$ Path's set **

Path sums rule (dealing with the '+')

"Path sum denotes the paths in parallel between nodes"

eg: If a and b are parallel paths between two nodes a+b



⇒ $f(x+y+d)g(u+v+w+h+i+j)k$

Distributive law :- $A(B+C) = AB+AC$, $(B+C)D = BD+CD$ } holds good

eg :- $e(a+b)(c+d)f = eacf + eadf + ebcf + ebdf$

Absorption rule : $X + X = X$

↓ X denotes a set of paths

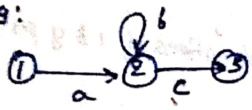
eg: $X = a + aa + abc + abcd + def$

then $X + a = X$; $X + aa = X$

$X + abc = X$; $X + abcd = X$; $X + def = X$

Loops:- "loops are infinite set of parallel paths"

eg:



∴ path expression would be

$X = ab^*c = abc + ac + abbc + abbbc + abbbbc + \cdots$

Reduction procedure Algorithm :-

"This section presents a reduction procedure for converting the flow graph whose links are labelled with names into a path expression that denotes the set of all entry/exit paths in that flow graph. This procedure is a node by node removal algorithm"

Procedure :-

1. Combine all serial links by multiplying their path expression.
2. Combine all parallel links by ~~multiplying~~ adding their path expression.
3. Remove all self loops (from any node to itself) by replacing them with a link of X^*, where X is the path expression of that link in that loop.

Loop :

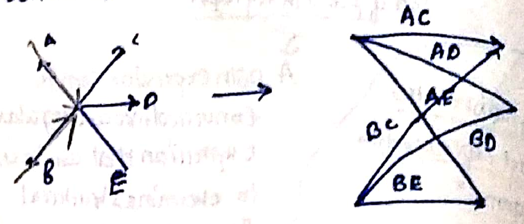
4. Select any node for removal other than the initial on final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks with that node.
5. Combine any remaining serial links by multiplying their path expressions.
6. Combine all parallel links by adding their path expressions.
7. Remove all self loops as in step-3
8. Does the graph consist of single link between the entry node and exit node ?

   If yes then the path expression for that link is path expression for original flow graph

   Otherwise return to step 4

end Loop

Note :- "A flow graph can have many equivalent path expressions between a given pair of nodes; that is there are many different ways to generate the set of all paths between two nodes without affecting the content of that set.



∴ $(a+b)(c+d+e) = ac + ad + ae + bc + bd + be$

The above is the cross term step (↗ step ④) : the fundamental step of the reduction algorithm

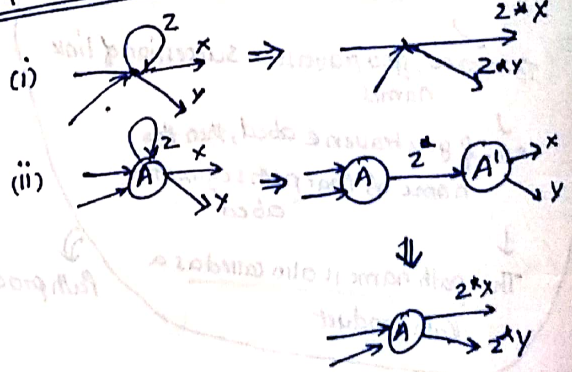
↓

It eliminates one node from total no. of nodes

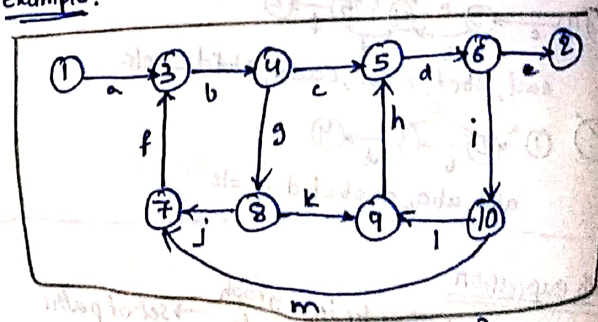
↓

If this procedure is continued successfully there will be only one entry node and one exit node

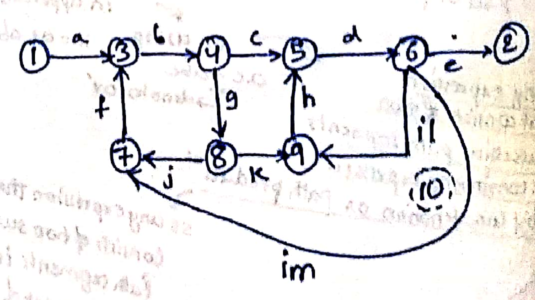
loop removal operations :- There are two ways to look at the loop removal operation



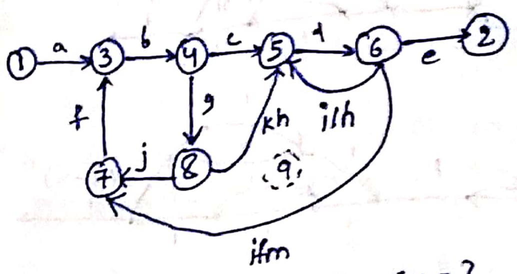
example :-



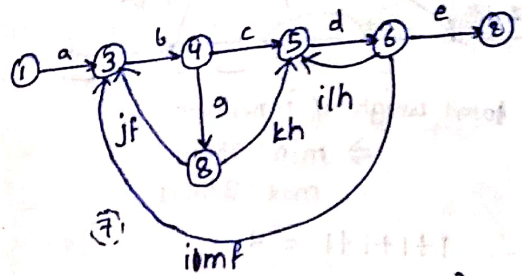
* remove node ⑩ Apply steps {4, 5}



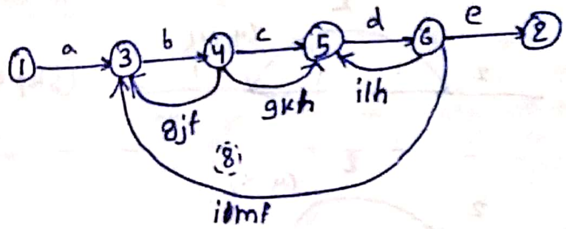
* remove node ⑨ Apply steps {4,5}



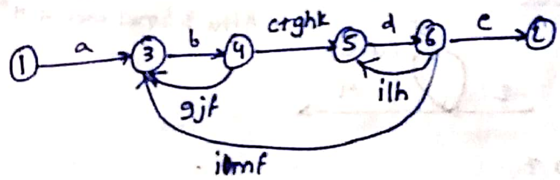
* remove node ⑦ Apply steps {4,5}



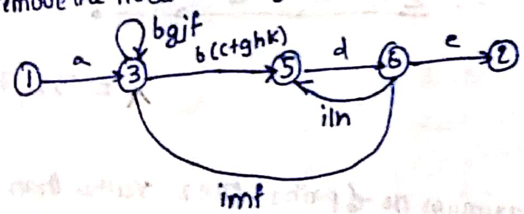
* remove node ⑧ Apply steps {4,5}



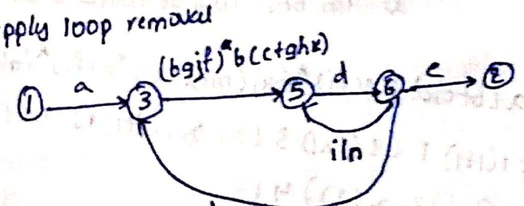
* c & gkh are parellel terms



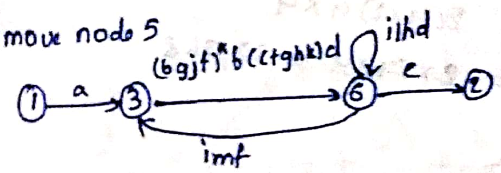
* remove the node 4 : it gives a loop term



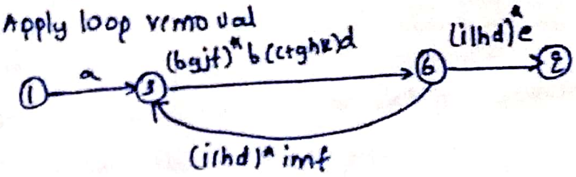
* Apply loop removal



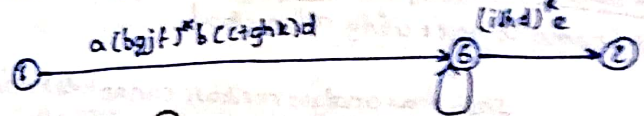
* remove node 5

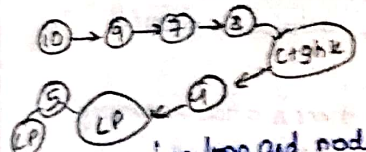


* Apply loop removal



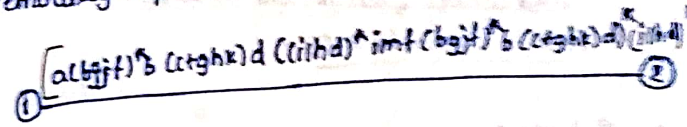
* removing node 3





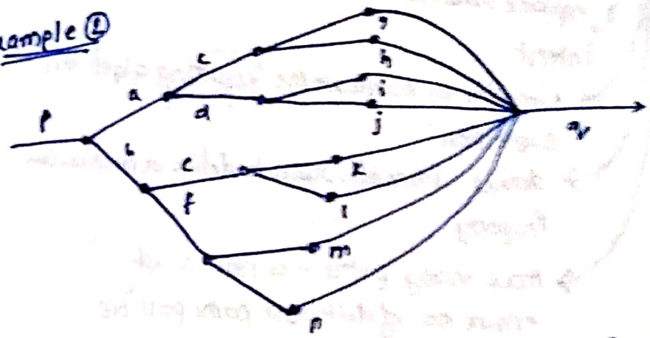
(ilhd)*imf (bgjf)*b(c+ghk)d

* removing loop and node 6



is the final path expression obtained

example ②



⇒ p(a(c(g+h)+d(i+j))+b(e(k+l)+f(m+n)))q

Regular expressions & flow anamoly detections

Problem "The generic flow-anamoly detection problem is looking for any specific sequence of options considering all possible paths through a routine"

method "Annotate each link in the graph with appropriate operator on the null operator 1" & "simplify"

"you now have a regular expression that denotes all possible sequences of operators in that graph"

Huangs theorem :- let A,B,C be non empty sets of character sequences whose smallest string is at least one character long. Let T be a two character string. Then if T is a substring of AB^nC, then T will appear in AB²C

eg:- let A=PP ; B=SRR ; C=RP ; T=SS

According to the theorem SS will appear in pp(srr)rrp if it appears in PP(SRR)*RP

∴ A = P+PP+PS

B = PSR+PS(R+PS)

C = RP

T = P4

∴ (P+PP+PS)(PSR+PS(R+PS))²RP

There is no sequence in AB²C of P4

limitations:- Huang's theorem, beyond three characters it becomes complex using this method
↓
static flow analysis methods cannot determine whether a path is achievable or not.
↓
we cannot tell whether there is a flow-anomaly and where, but we can tell whether the path is acheivable or not.

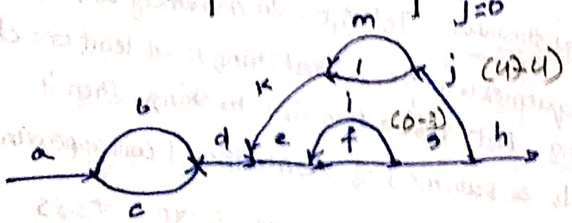
Applications:-

- convert program (graph into a path expression)
- replace link names with link weights for our interest
- simplify or evaluate the "resulting algebraic expression"
- devise arithmatic rules to define a particular property.
- How many paths in a flow graph
  - max no. of different paths possible
  - what is fewest no. of paths possible
  - How many different no. of paths are there
  - average no. of paths. (meaningless ofcourse)
- Maximum path count Arithmetic

. There are three cases: Parellel links, serial links, loops

case	Path expression	weight expression
Parellels	A+B	$W_A + W_B$
Series	AB	$W_A W_B$
Loop	A^n	$\sum_{j=0}^{n} W_A^{\,j}$

eg:-



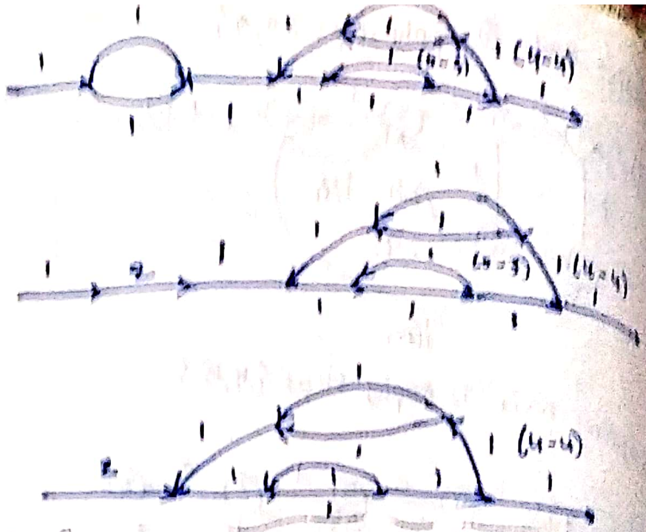
Path expression:-

a(b+c)d{e(fi)* fgj(m+l)k3* e(fi)* fgh

step 1: replace link name with maximum no. of paths of that link (1)

step 2: combine parellel loop paths outside the loop

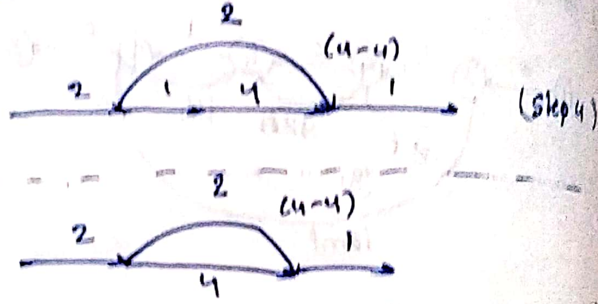
step 3: remove the nodes



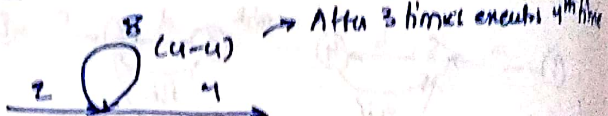
step 4: total weight of inner loop
⇒ min: 0 times
max: 3 times

∴ 1+1+1+1 = 4

step 5: multiply link weights 1×4 = 4



step 6: multiply weight of link 2×4 = 8



step 7:

∴ 2 → 8^4 → 4

= 32,768

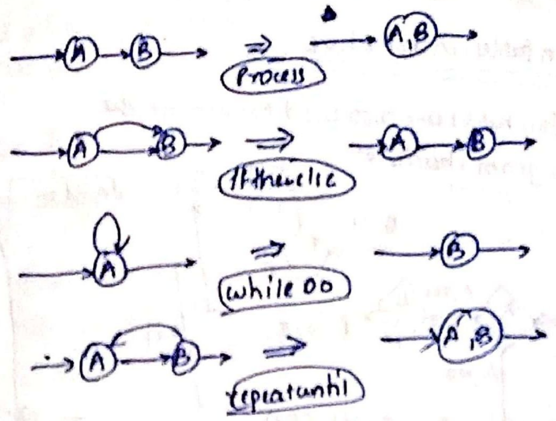
∴ maximum no. of paths 8192 rather than 32,768 as 4th time will be taken 0 or 4 times

a(b+c)d{e(fi)* fgj(m+l)k3* e(fi)* fgh
= 1(1+1) 1 (1(1×1) 3 1×1 × 1(1+1) 1) 4 1(1×1) 3 ...
= 2 (131 × (2)) 4 1 3
= 2(4×2) 4 × 4
= 2×8^4×4 = 32,768

Structured flowgraph

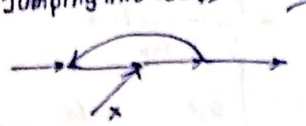
: the flowgraphs that do not involve ad-hoc rules such as not using goto's

"A structured flowgraph is one that can be reduced to a single link by successive application of transformations."



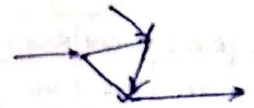
unstructured subgraphs

1. Jumping into loops → Intention of the program is deviated when reduction procedure is applied

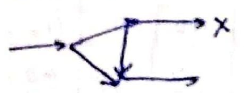


2. Jumping out of loop → Statements that are outside of the loop are reached

3. Branching into decisions → Intention of statements is changed



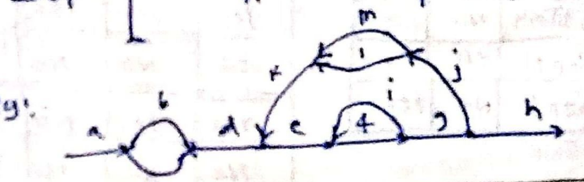
4. Branching out of decisions → Intention of condition & statements is changed



Lower path count arithmetic (lower bound on n. of paths)

Case	Path expression	weight expression
Parallels	A+B	$W_A + W_B$
series	AB	$\max(W_A, W_B)$
Loop	A^n	1, W_1

eg:

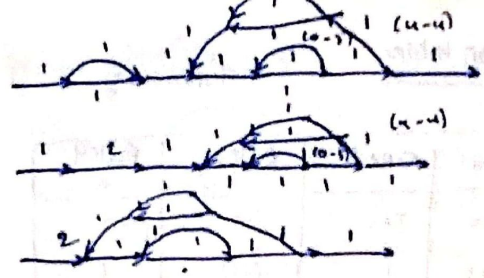


mean processing time of a routine

$(P_A T_A + P_B T_B)/(P_A + P_B) = T_{A+B}$; $P_{A+B} = P_A + P_B$ → Parallel

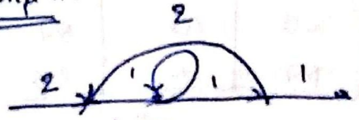
$T_{AB} = T_A + T_B$; $P_{AB} = P_A P_B$ → series

$T_A = T_A + T_L P_L/(1-P_L)$; $P_A = P_A/(1-P_L)$ → loop

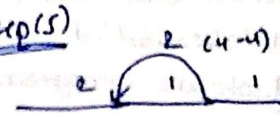


From step (4) ~~same as previous~~ different than previous

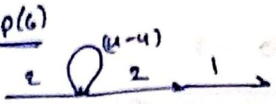
step(4):



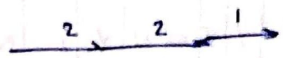
step(5)



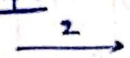
step(6)



step(7)



step 8



= 2

∴ Atleast 2 path to cover

Logic Based testing:

- functional requirements of many programs are specified by decision tables.
- Consistency and completeness can be analysed by boolean algebra. Boolean algebra is trivialised by K-V charts
- Boolean algebra (sentential calculus) is most basic of all logic systems
- Higher order logic systems are used for formal specifications

Knowledge based system :- (artificial intelligence system)

"knowledge based system has become the programming construct of choice for many applications".

- Knowledge based system incorporate knowledge from a knowledge domain such as medicine, law, engineering into a database. Then the data is queried and the solutions are extracted"
- Decision tables are extensively used in business data processing
- Decision tables are preprocessors as extensions to COBOL are in common use.
- Boolean algebra is embedded in the implementation of these processors.

examples of decision tables — Condition entry

eg: 1

	Rule1	Rule 2	Rule3	Rule4
Condition1	Yes	Yes	NO	NO
Condition2	Yes	I	NO	I
Condition3	NO	Yes	NO	I
Condition4	NO	Yes	NO	Yes
Action1	Yes	Yes	NO	NO
Action2	NO	NO	Yes	NO
Action3	NO	NO	NO	Yes

(Condition stubs / Action stubs; Action entry)

- "A rule specifies whether a condition should or should not meet for the rule to be satisfied".
- Condition and predicate are synonymous.

eg: 2 : Printer Troubleshooting.

		Rules							
Conditions	Printer not ...	Y	Y	Y	Y	N	N	N	N
	Printer does not Print	Y	Y	N	N	Y	Y	N	N
	A redlight is flashing	Y	N	Y	N	Y	N	Y	N
	Printer is unrecognized	Y	Y	N	Y	N	N	Y	N
Actions	check the Powercable			X					
	check the computer Cable	X		X					
	check Printer software	X		X		X		X	
	check/replace Ink	X	X		X	X			
	check for Paperjam		X		X				

"In addition to the stated rules a default rule must also be provided"

Decision table processors :-

- decision tables can be automatically translated into code and such are a higher-order language

Decision tables as a basis for testcase design

1. specifications are given as decision tables or can be converted into one

2. The order in which the predicates are evaluated does not affect the interpretation of the rules.

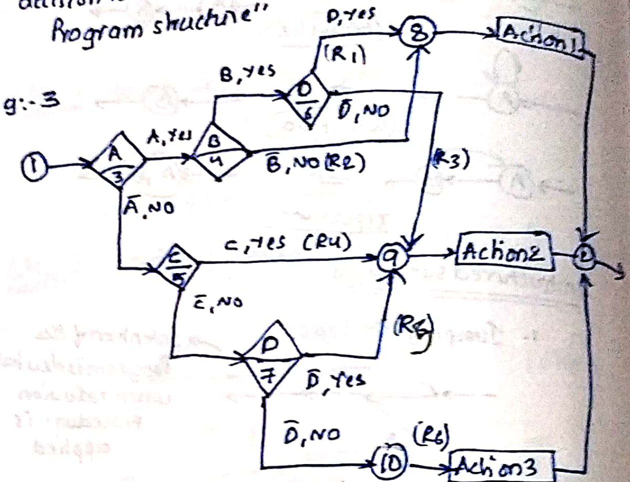
3. The order in which rules are evaluated will not affect the resulting action.

4. once a rule is satisfied and action is selected, no other rule need to be evaluated.

Decision tables and structure

"decision tables are also used to examine the Program structure"

eg:- 3



"If the decision appears on a path put a Yes or no as appropriate - If the decision doesn't appear on the path Put a I"

	Rule1	Rule 2	Rule3	Rule4	Rule 5	Rule6
conditionA	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
Action3	NO	NO	NO	NO	NO	Yes

for above sample program this is the decision table (3)

Expanding the immaterial cases

eg: 4

	Rule1	Rule2
Condition1	Yes	Yes
Condition2	I	NO
Condition3	Yes	I
Condition4	NO	NO
Action 1	Yes	NO
Action 2	NO	Yes

⇒

Rule 1.1	Rule1.2	Rule1.1	Rule...
Yes	Yes	Yes	Yes
Yes	NO	NO	NO
Yes	Yes	Yes	NO
NO	NO	NO	NO
Yes	Yes	NO	NO
NO	NO	Yes	Yes

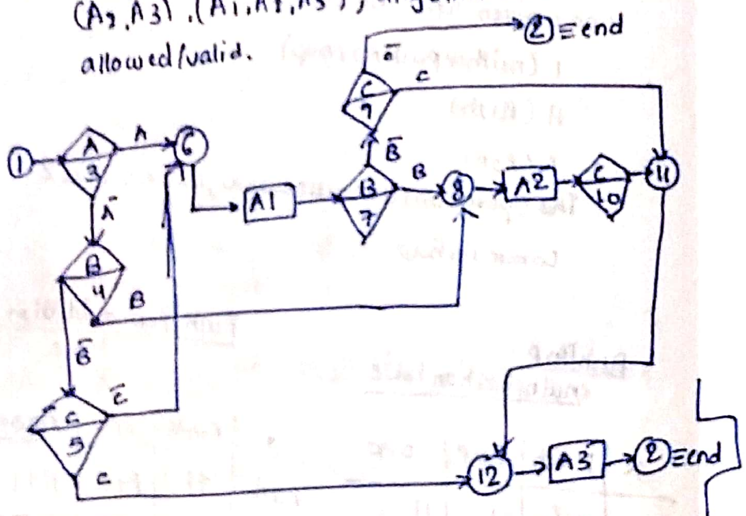
Similarly for eg - 3

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 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	~~Y~~Y	NN

"As a first check count number of Y & N in each row, we can identify bug that way"

eg: 5 : A troublesome program

- If condition A is met, do process A1 leaving about other actions are performed on other conditions are met or not.
- If condition B is met, do process A2 no matter Other conditions are met or not.
- If condition C is met do process A3 no matter Other conditions are met or not.
- If above 3 Conditions are not met then do processes A1, A2, A3
- valid processes are (A1), (A2), (A3), (A1, A3), (A2, A3), (A1, A2, A3); any other order is not allowed/valid.



"for the B & C cases A2 & A3 Processes would be bypassed"

	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	~~$\bar{A}BC$~~	$\bar{A}B\bar{C}$	$AB\bar{C}$	ABC	$A\bar{B}C$	$A\bar{B}\bar{C}$
condition A	N	N	N	~~N~~	N	Y	Y	Y	Y
Condition B	N	N	Y	X	Y	Y	Y	N	N
Condition C	N	Y	Y	X	N	N	Y	Y	N
Action 1	Y	N	N	X	N	Y	Y	Y	Y
Action 2	Y	N	Y	X	Y	Y	Y	N	N
Action 3	Y	Y	Y	X	N	N	Y	Y	N

Path expressions

"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

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 labelled with a letter say A and the No or False branch with same letter overscored
2. The truth value of a path is the product of individual labels. concatenation or products mean "AND".
3. If two or more paths merge at a node, the fact is expressed by a use of + sign which means OR.

eg: for the eg: 5 (N → node)

$N6 = A + \bar{A}\bar{B}\bar{C}$

$N8 = (N6)B + \bar{A}B = AB + \bar{A}\bar{B}\bar{C}B + \bar{A}B$

$N11 = (N8)C + (N6)\bar{B}C$

$N12 = N11 + \bar{A}\bar{B}C$

$N2 = N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}$

Rules of Boolean algebra :-

- AND (X), OR (+), NOT ($\bar{X}$)

1. $A + A = A$
   $((\bar{A} + \bar{A}) = \bar{A})$
2. $A + 1 = 1$
3. $A + 0 = A$
4. $A + B = B + A$ // commutative
5. $A + \bar{A} = 1$
6. $AA = A$
7. $A \times 1 = A$
8. $A \times 0 = 0$
9. $AB = BA$
10. $A\bar{A} = 0$
11. $\bar{\bar{A}} = A$
12. $\bar{0} = 1$
13. $\bar{1} = 0$
14. $\overline{A+B} = \bar{A}\bar{B}$ // demorgans law
15. $\overline{AB} = \bar{A} + \bar{B}$
16. $A(B+C) = AB + AC$ // distributive
17. $(AB)C = A(BC)$ // associative
18. $(A+B) + C = A + (B+C)$ // additive
19. $A + \bar{A}B = A + B$ // absorption
20. $A + AB = A$

Points to ponder

- Individual letters in Boolean Algebra are called Literals.
- The product of several literals is called a product term.
- An Arbitary boolean expression that has been multiplied out so that it consists of sum of Products (ABC+DEF+GH) is said to be in sum of products.
- The resultant of simplifications is again the form of sum of products and each product term in such a simplified version is called prime implicant.

$\therefore N6 = A + \bar{A}\bar{B}\bar{C} = A + \bar{B}\bar{C}$

$N8 = (N6)B + \bar{A}B$
$= (A + \bar{B}\bar{C})B + \bar{A}B$
$= AB + \bar{B}\bar{C}B + \bar{A}B$
$= AB + 0C + \bar{A}B$
$= AB + 0 + \bar{A}B$
$= AB + \bar{A}B$
$= (A + \bar{A})B$
$= B$

$\therefore N8 \neq B + \bar{A}\bar{C}$ (shouldn't be) But B

$N11 = (N8)C + (N6)\bar{B}C$
$= BC + (A + \bar{B}\bar{C})\bar{B}C$
$= BC + A\bar{B}C$
$= C(B + \bar{B}A)$
$= C(B + A)$
$= AC + BC$

$N12 = N11 + \bar{A}\bar{B}C$
$= AC + BC + \bar{A}\bar{B}C$
$= AC + BC + \bar{A}\bar{B}C$
$= C(B + \bar{A}\bar{B}) + AC$
$= C(\bar{A} + B) + AC$
$= C\bar{A} + AC + BC$
$= C + BC$
$= C$

$N2 = N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}$
$= C + B\bar{C} + (A + \bar{B}\bar{C})\bar{B}\bar{C}$
$= C + B\bar{C} + \bar{B}\bar{C}$
$= C + \bar{C}(B + \bar{B})$
$= C + \bar{C}$
$= 1$

∴ loops complicate things

Similarly

$N12 \neq C + \bar{A}\bar{B}\bar{C}$

But just C

" Implicants are the product terms with special properties that we are trying to implement i.e,
An implicant P of a function F is a product term P such that F=1 wherever P=1 "

" A prime implicant P of F is any implicant that is not covered by any other implicant of F "

" An essential prime implicant of function F is one that covers a minterm of F not covered by any other prime implicant of F " ★★

Push/Pop arithmetic

: useful in debugging and test cases design

case	Path expression	weight expression
Parallels	A+B	$W_A + W_B$
Series	AB	$W_A W_B$
Loop	A^*	W_A^*

: on a given link notations
1 (neither push nor pop)
H (Push)
P (Pop)

: The operations are associative, distributive & commutative

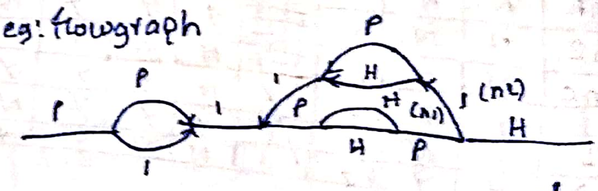
Push/Pop multiplication table

x	Push	Pop	None
Push	H^2	1	H
Pop	1	P^2	P
None	H	P	1

Push/Pop addition table

+	Push	Pop	none
Push	H	P+H	H+1
Pop	P+H	P	P+1
none	H+1	P+1	1

eg: flowgraph



$P(P+1)1\{P(H+H)\}^{n_1}HP1(P+H)1\}^{n_2}P(H+H)^{n_1}HPH$

: let

M1 : no. of times innerloop is taken.

M2 : no. of times outerloop is taken.

M1	M2	Push/Pop
0	0	$P+P^2$
0	1	$P+P^2+P^3+P^4$
0	2	$\sum_{1}^{6} P^i$
0	3	$\sum_{1}^{8} P^i$
1	0	1+H
1	1	$\sum_{0}^{3} H^i$
1	2	$\sum_{0}^{5} H^i$
1	3	$\sum_{0}^{7} H^i$
2	0	H^2+H^3
2	1	$\sum_{4}^{7} H^i$
2	2	$\sum_{6}^{11} H^i$
2	3	$\sum_{8}^{15} H^i$

: Stack will only be popped if inner loop is not taken

: stack will be left alone if iterated once

: for all other values of innerloop the stack will be only pushed.

KV charts :-

: If expressions in four, five, six variables, then it is tedious to deal them with algebra.

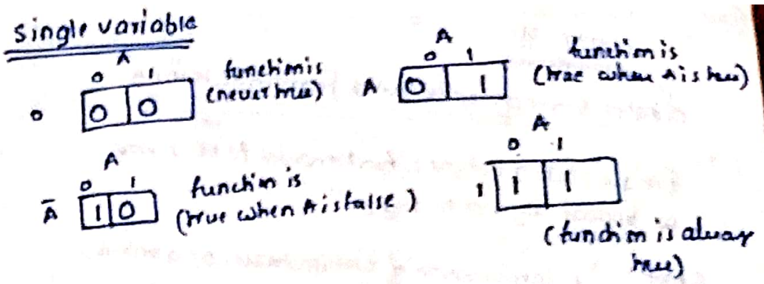
: So we use Karnaugh-Veitch charts to reduce them graphically

(2^{2^n}) combinations

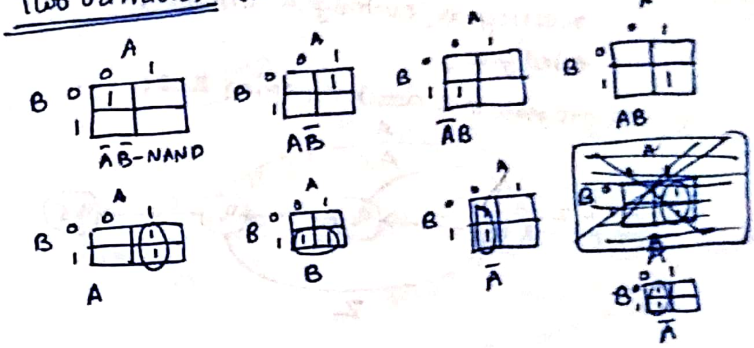
1 → True

0 → False

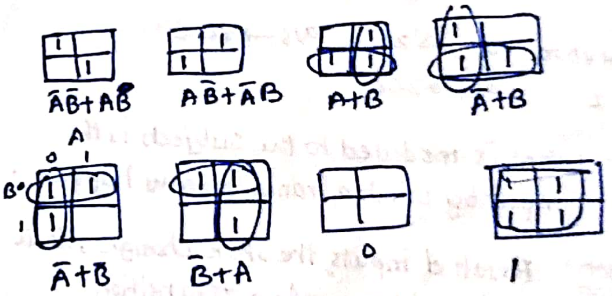
Single variable



Two variables (16 combinations (2^{2^2}))

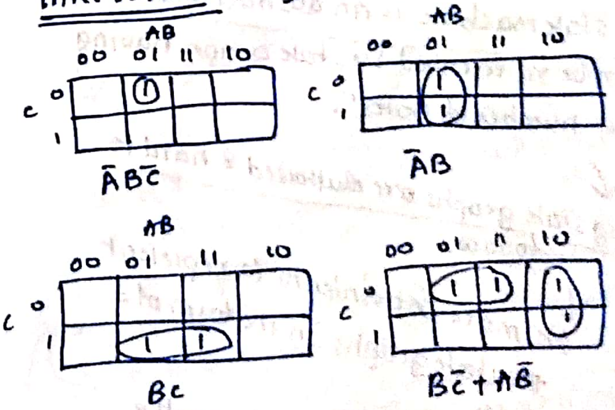


another 8 combinations



" For one variable there are $2^{2^1} = 4$ functions, 16 functions of 2 variables - - -

Three variables :- (give few examples)



UNIT- IV

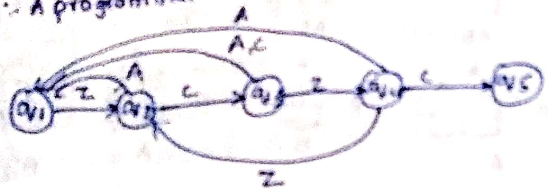
States, state graphs and Transition testing

* "Finite state machine is fundamental to s/w E same as boolean algebra to logic"

State:- "A combination of circumstances or attributes belonging for time bring to a person or thing".

e.g: engine states with respect to its transmission reverse gear, nutral gear, first gear, second gear, third gear.

eg:- A program that detects the string zczc



$a_1 \rightarrow$ None $a_2 \rightarrow z$ $a_3 \rightarrow zc$ $a_4 \rightarrow zcz$ $a_5 \rightarrow zczc$

Input:- what is modelled to the subjects is the input by which a transition may happen.

Transition: Result of inputs, the state changes, then it is said to made a Transition.

" There is one outlink from every state for every input"

" A finite state machine is an abstract device that can be represented by state graph having a finite number of states".

↙ Big state graphs are cluttered & hard to follow

↓ It's more convinient to represent the state graphs in the form of a table

↙ where each row of table represents a state

↙ And each column represents a input.

State	z	c	A
None	z	None	None
z	z	zc	None
zc	zcz	None	None
zcz	z	zczc	None
zczc	zczc	zczc	zczc

"State graphs donot represent time but sequence as a transition may take a second or a century"

↙ " But in Practical there is no correspondence between lines of code and states". The state table just forms the basis".

Good State graphs vs Bad state graphs

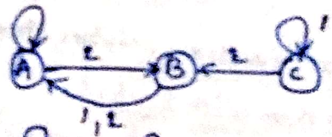
Q: how to judge! ..

Some principles involved:-

1. Total number of states = Product of possibilities that make up the state
2. For every state and input only one transition to another state (possibly same state)
3. For every transition there is one output action specified; The output may be trivial/ ...
4. For every state there is a sequence of inputs that will drive the system back to the same state.

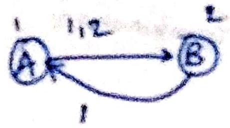
Important graphs: (bad graphs)

State A cannot be reached ever again



State c is never reached

States A & B are not reachable (no transition)



Two transitions are specified for an input of 1 in state A

State Bugs — Number of states

: no. of states can be computed as follows
→ Identify all the component factors of a state
→ Identify all the allowable values for each factor
→ Number of states is product of number of all allowable values of all factors.

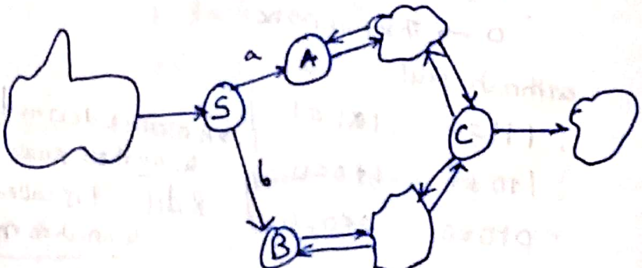
There may be a difference between developers state count and testers state count.

" A robust peice of software will not ignore impossible states but will recognize them and raise an illogical condition handler"

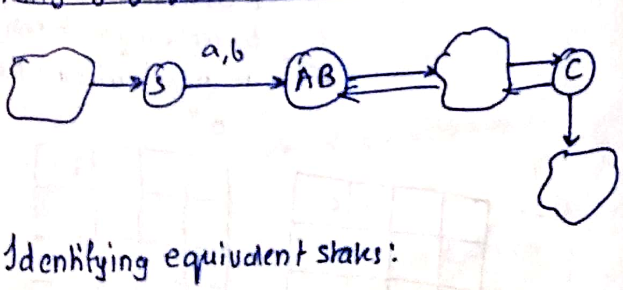
~~Equalent~~ equivalent states :

" Two states are said to be equivalent if every Sequence of inputs starting from one state produce Same sequence of outputs when started from other side".

This notion can also be extended to set of states



merging of equivalent states



: Identifying equivalent states:

- The rows corresponding to the two states are identical with respect to input/output/next state but name of next state could differ.
- If two sets of rows have identical state graphs with respect to transitions & outputs, then they can be merged.

Transition Bugs (unspecified and contradictory Transitions)

- every input, state must have a transition
- If a transition is impossible, then there needs to be mechanism that prevents input from occurring in that state.
- A program can't have contraddictions and ambiguities.
- Ambiguities are mostly impossible, because Program does something for every input. (Inc) eg. R/W example

Unreachable states → A state which no input sequence can reach
- is similar to unreachable code
- unreachable state is not impossible (reason: incorrect transitions)
  → result: 1. There is a bug 2. Transitions are there but not known

Dead states → If dead state is once entered, can never be left.
→ may not be a bug but should review

Output errors → wrong no. of states
→ wrong transitions
→ There maybe output errors even when there are no dead, unreachable, impossible states.
→ unreachable states
→ dead states

Impact of bugs

Principles of state testing
1. define a set of covering input sequences that get back to the initial state when starting from initial state
2. For each step in each input sequence define expected next state, transition, output

Set of tests containing
1. Input sequences ✓
2. Corresponding transitions ✓
3. output sequences ✓

→ Starting point of state testing

Limitations | **Extensions**

Insisting on explicit specification of transition & output for every combination

A set of input sequences that provide coverage of all nodes & links is mandatory.

Testability tips

1. Building explicit finite state machines
2. using switches
3. using flags
4. Identifying unacheivable paths
5. Identifying essential & Inessential finite state behaviour
   (as simple in a state is a flip or flop)
6. Following the designing guidelines

Graph, Matrices and applications

(Problem with the pictorial graphs)

↓ graphs were introduced as an abstraction of software structure

↓ Paths are used to trace and paths are subjected to errors

↓ you can miss a link here or there or visit a link twice unknowingly

↓ one method to avoid this problem is to represent graph as a matrix.

(Tool Building)

↓ It's hard to build algorithms over visual graphs so the properties or the graph matrices are fundamental to tool building

(The basic Algorithms)

↓ The basic tool consist of

: matrix multiplication ✓

: Partitioning Algorithm {to convert graphs}

: A collapsing process (to get path expression between two nodes)

(The matrix of a graph)

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

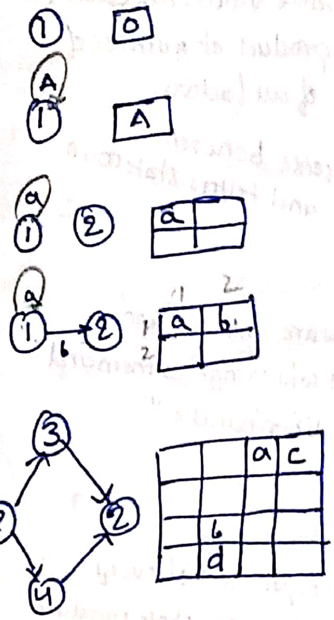
: size of matrix = number of nodes ✓

: The entry at a row & coloumn intersection is the link weight of the link that connects the two nodes in that direction.

: A connection from node i to j does not imply a connection from node j to i (A×B ≠ B×A)

: If there are multiple links at a node, then "+" sign denotes parellel links

some graphs & matrices



A simple weight :-

"A simple weight we can use is to note that there is or isn't a connection"

1 → There is a connection

0 → There is none

arithmetic rules :-

: 1+1=1   : 1×1=1

: 1+0=1   : 1×0=0

: 0+0=0   : 0×0=0

→ A matrix defined using these rules & defined is called a connection matrix

↓ "A connection matrix is obtained by replacing each entry with 1 if there is a link, otherwise 0"

∴

		a	c
	b		
	d		

↓

		1	1
	1		
	1		

: each row represents a outlink of the node

: each coloumn denotes the links corresponding to that node

: "A branch is a node with more than one non zero entry in its row"

: "A junction is a node with more than one nonzero entry in its coloumn"

: "A self loop is an entry along the diagonal"

Cyclomatic complexity

" Cyclomatic complexity is obtained by subtracting '1' from the total number of entries in each row, ignoring rows with no entries. Then adding these values and then adding '1' to the sum yeilds graph's cyclomatic complexity "

		1	1
	1		
	1		

2-1=1

1-1=0

1-1=0

1

1+1=2 (cyclomatic complexity)

Relations :-

: A relation is a property that exists between two objects of interest

→ **Transitive Relations** :

A relation is transitive if aRb and bRc implies aRc

↳ most relations used in testing are transitive

eg: ≥ , ≤ , faster than, slower than

eg (Intransitive) : is friend of, neighbour of

→ **Reflexive Relations** :

A relation R is reflexive if, for every a, aRa

↳ Reflexive relation is equivalent to a self loop at every node

↳ ex: equals, is a relative of

eg (not reflexive) : not equals

→ **Symmetric relations** :

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

↳ A graph whose relations are not symmetric is called a directed graph ; if symmetric undirected graph

→ **Anti symmetric relations**

A relation R is antisymmetric if for every a & b if aRb and bRa then a=b (same elements obviously)

→ **equivalence relations**

↳ An equivalence relation is a relation that satisfies the reflexive, transitive, symmetric properties → If it satisfies equivalence, it can form a class

Testing a member of equivalance class is as effective as testing whole class

Partial ordering relations

: A partial ordering relation satisfies the reflexive, transitive & asymmetric properties.

↳ Partial ordered graphs are loop free

The powers of a matrix

- each entry in graph's matrix expresses a relation between the pair of nodes.
- The square of matrix represents all path segments two links long.
- The third power represents all path segments three links long.

Matrix powers & products :

- let a matrix a_{ij} then square of matrix obtained by replacing every entry with

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

- let matrices A_{ik} B_{kj} then $A \times B = C_{ij}$ (new matrix)

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

Partitioning algorithm

- Consider any graph over a transitive relation.
The graph may have loops
- Partition such that every loop is contained within one group or another.
- Such a graph is partially ordered & we might embed the loops within a subroutine
- It's much harder to program a tool recognize the loops unless you have a solid algorithm & tool.

Node reduction algorithm :

" The matrix powers usually will tell us more than we want to know about graphs ".

1. select node for removal; replace its node by its equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms & simplify as far as you can
3. observe loop terms and adjust the out links that have a self loop
4. The result is a matrix whose size is reduced by 1

Continue until only two nodes of interest exist.

UNIT-V

Defect management

Defect → something which does not allow product to meet customer requirement
↓
And it is cause for customer dissatisfaction
↓
It can be
- A extra
- A missing requirement
- requirement not implemented correctly

(defect is something that makes customer unhappy)

Root causes of defect → defect is not an accident, but occurs but it something has not worked as planned

reasons

1. requirements not clearly defined by user
2. understanding problems is not effective
3. requirements are not resolved completely
4. designers are wrong
5. requirements & its design is different
6. Process used for development, testing are not capable

Effects of defects → The final effect of defect is customer dissatisfaction
↓
not fit for use

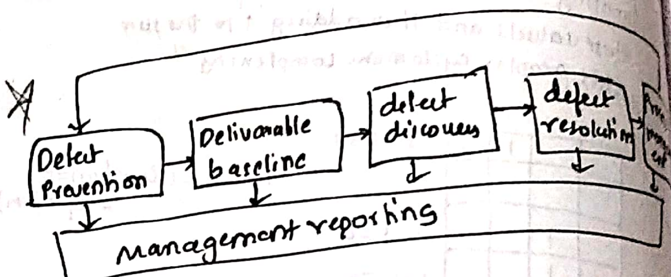
due to
- functionality not available
- performance is not acceptable
- security issues

Defects class classification

1. requirements defects
   - → functional defects
   - → interface defects
2. Design defects
   - → Algorithmic defects
   - → module interface defects
   - → system interface defects
   - → user interface defects
3. coding defects
   - → variable declaration
   - → Database defects
   - → documentation defects
4. Testing defects
   - → Test design defects
   - → Test environment defects
   - → Test tool defects

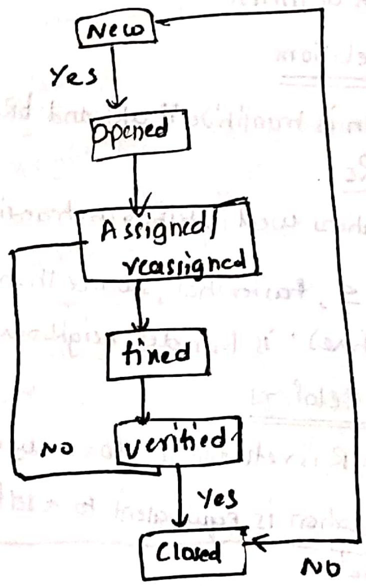
Defect management process

- primary goal is to prevent defect
- defect management must be risk driven



Defect life cycle

- Defect identified in verification /validation must be first analyzed to check if it is defect or not.



Defect template: Every organization must have a defect template to capture the defect

IPPRMDSDSAE

- ID
- Project
- Product
- Release version
- module
- Detected Build version
- Summary
- Description
- Steps to replicate
- Actual Result
- expected result

Attachments
Remarks
Defect severity
Reported by
Assigned to
Status
Fixed Build version

ARDRASF

Finding & reporting cause of bugs

- Correction, corrective actions, preventive actions

Defect naming: convention; named based on nature

Defect resolution
- defect fixing
- retesting
- resolving
- Taking actions on process, methods that produced defect
- regression (to find if it has any negative effects on unchanged part of system)

Defect correction: first step to initialize when defect is found

Corrective action: identifying process, methods that caused defect

Defect prevention: prevention & updating

Delivery baselining: once defect is fixed, retested and found to be closed, the product is created again

Process improvement: Continuous search to improve baseline the processes

Management reporting: maint information flow - to and (SWOT)

Defect fixing :-
- defect must be governed by probability of happening
- Frequency & effect of a defect on user & its impact must drive defect prevention mechanism

Identify critical risks due to defects in S/w → estimate expected probabilities and/or impact of such defects → minimize expected impact on probability by corrections, corrective actions & preventive actions

Examples of risks
1. missing key requirements
2. Application's critical part won't work
3. Vendor supplied S/w doesn't work
4. S/w does not support major business functions
5. Poor performance
6. h/w malfunctioning
7. h/w or s/w donot integrate
8. user unwilling to adopt to new system

"Actual impact of defect can be measured when the risk realizes or becomes a reality in production environment". handling risks:
> Accept risk as it is
> Bypassing the risk

Risk minimization:
- eliminate risk
- mitigation of risk
- contingency planning

Techniques for finding defects
- Static Techniques
- dynamic Techniques
- Operational Techniques

Reporting defect
1. correct defect
2. report system status
3. gather statistics
4. process improvement.

Testing Tools

- Testers use many tools during a software test life cycle
- Tools may be used for testing, for
  - → documenting defects
  - → preparing test artifacts

Software tools
- defect tracking tools
- regression testing tools
- configuration tools
- communication tools
- simulations

Hardware tools
- machines
- servers
- Printers
- routers
- Any hardware simulators.

Features of test tool
- A test tool is a vehicle for performing test process & each tool has a specific purpose.
- Tools are used in SDLC

Guidelines for Selecting a tool
→ The tool must match its intended use.
→ matching tool with skills of testers
→ affordable tools
→ Backdoor entry of tools must be prevented (Pirated)

Tools & skills of testers
- user skills
- programmer skills
- System skills
- Technical skills

Static testing tools
- Static testing tools are used during Static analysis of system

Types :-
→ code profiling
→ data profiling
→ Test data generators
→ syntax-checking tools

Dynamic Testing tools
- are used ranging from unit testing to the System testing and performance testing

→ Regression testing
→ Defect Tracking
→ Performance, load, stress testing tools

Advantages of using tools
→ Tools work faster than human beings
→ Tools are consistent in behaviour
→ Some areas of SDLC can be only completed using tools

Disadvantages:
→ wrong selection of tools may lead to disaster
→ People must be trained
→ cost-benefit analysis before acquiring a tool
→ maintaining scripts in changing requirements may be difficult.

when to use Automated test tools
- Number of iterations of testings
- Complexity of test cases
- Test case dependency

Partial Automation :- Automating parts of testing, but not all of software testing process

Points to be remembered when thinking of test automation
→ Platform & OS independence
→ data driven capability
→ version control is friendly
→ common drivers can be used
→ support distributed execution environment
→ extensible test process

Framework approach in Automation
"A framework is an integrated system that sets the rules of automation of a specified Product"

Synchronization
→ A distributed test case consists of two or more parts, each processes on a different system that interact with each other
- Automatic synchronization
- user defined synchronization

Difficulties while introducing new tools

1. organization obstacles
   - cost of tool
   - Training requirements
   - unwillingness to learn new skills
2. Tool problems
3. Environmental problems

Process of procurement of COTS

- goals to be met
- objectives of tool
- acquisition plan
- selection criteria
- identify candidate
- user review of candidate
- score candidate
- select tool
- procure tool
- evaluation tool
- implementation plan
- Training plan
- orientation
- Training
- evaluation report
- determine if goals met or not.

Procurement of tools from contractor

→ difference between IN-HOUSE software development & software developed by a contractor

- control over development process by vendor
- control over resources from vendor-side
- cultural differences between vendor & customer
- employee morale
- root causes of problems may not be effectively addressed by vendor

Process of procurement of tools from contractor

→ Same as above +

Request of proposal (REP) /
Request for quotation (RFQ) /
RFI (Request for information)
+
Technical evaluation
+
Acceptance testing