SOFTWARE TESTING METHODOLOGIES

BY

B. Pravallika

Software Testing Methodologies

Text Books: 1.Software Testing Techniques: Boris Beizer

2. Craft of Software Testing: Brain Marrick

UNIT-I

Introduction: Purpose of testing. Dichotomies. Model for testing.

Consequences of bugs. Taxonomy of bugs.

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

Introduction

What is Testing?

Related terms : SQA, QC, Verification, Validation

Verification of functionality for conformation against given specifications
By execution of the software application

A Test
Passes: Functionality OK.
Fails: Application functionality NOK.
Bug/Defect/Fault: Deviation from expected functionality.
It’s not always obvious.

Purpose of Testing

1. To Catch Bugs

   - Bugs are due to imperfect Communication among programmers
     - Specs, design, low level functionality
   - Statistics say: about 3 bugs / 100 statements

2. Productivity Related Reasons
   - Insufficient effort in QA => High Rejection Ratio => Higher Rework => Higher Net Costs
   - Statistics:
     - QA costs: 2% for consumer products
       80% for critical software
   - Quality ⇔ Productivity

Purpose of Testing

Purpose of testing contd…

3. Goals for testing

Primary goal of Testing: Bug Prevention

- Bug prevented ⇒ rework effort is saved [bug reporting, debugging, correction, retesting]
- If it is not possible, Testing must reach its secondary goal of bud discovery.
- Good test design & tests ⇒ clear diagnosis ⇒ easy bug correction

Test Design Thinking

- From the specs, write test specs. First and then code.
- Eliminates bugs at every stage of SDLC.
- If this fails, testing is to detect the remaining bugs.

4. 5 Phases in tester's thinking

Phase 0: says no difference between debugging & testing

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

Purpose of Testing

Purpose of testing contd…

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

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

Phase 2: says Software does not work

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

Phase 3: says Test for Risk Reduction

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

Purpose of Testing

5 Phases in tester's thinking continued…

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

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

Cumulative goal of all these phases:

- Cumulative and complementary. One leads to the other.
- Phase2 tests alone will not show software works
- Use of statistical methods to test design to achieve good testing at acceptable risks.
- Most testable software must be debugged, must work, must be hard to break.

Purpose of Testing

purpose of testing contd..

5. Testing & Inspection

- Inspection is also called static testing.
- Methods and Purposes of testing and inspection are different, but the objective is to catch & prevent different kinds of bugs.
- To prevent and catch most of the bugs, we must
  - Review
  - Inspect &
  - Read the code
  - Do walkthroughs on the code

& then do Testing

Purpose of Testing

Further…

Some important points:

Test Design — After testing & corrections, Redesign tests & test the redesigned tests

Bug Prevention

Mix of various approaches, depending on factors
culture, development environment, application, project size, history, language

Inspection Methods

Design Style

Static Analysis

Languages – having strong syntax, path verification & other controls

Design methodologies & development environment

Its better to know:

Pesticide paradox

Complexity Barrier

Dichotomies

- division into two especially mutually exclusive or contradictory groups or entities
- the *dichotomy* between theory and practice

Let us look at six of them:

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

Dichotomies

1. Testing Vs Debugging

- Testing is to find bugs.
- Debugging is to find the cause or misconception leading to the bug.

- Their roles are confused to be the same. But, there are differences in goals, methods and psychology applied to these

#	Testing	Debugging
1	Starts with known conditions. Uses predefined procedure. Has predictable outcomes.	Starts with possibly unknown initial conditions. End cannot be predicted.
2	Planned, Designed and Scheduled.	Procedures & Duration are not constrained.
3	A demo of an error or apparent correctness.	A Deductive process.
4	Proves programmer's success or failure.	It is programmer's Vindication.
5	Should be predictable, dull, constrained, rigid & inhuman.	There are intuitive leaps, conjectures, experimentation & freedom.

Dichotomies

Dichotomies contd…

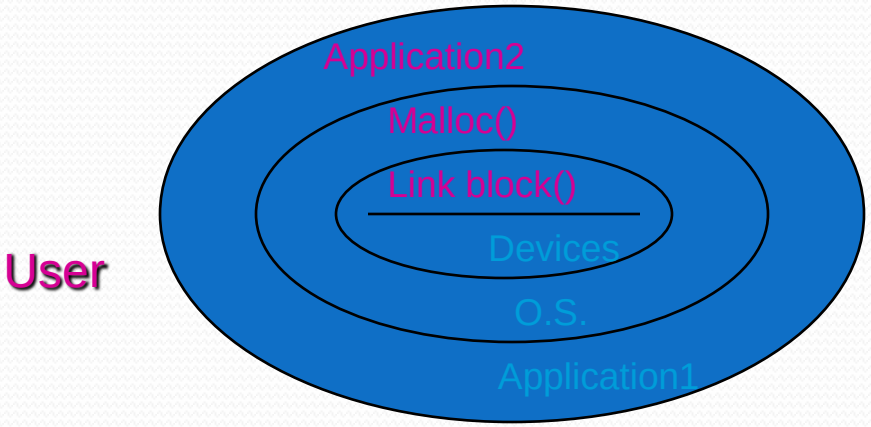
#	Testing	Debugging
6	Much of testing can be without design knowledge.	Impossible without a detailed design knowledge.
7	Can be done by outsider to the development team.	Must be done by an insider (development team).
8	A theory establishes what testing can do or cannot do.	There are only Rudimentary Results (on how much can be done. Time, effort, how etc. depends on human ability).
9	Test execution and design can be automated.	Debugging - Automation is a dream.

Dichotomies

Dichotomies contd..

2. Functional Vs Structural Testing

- **Functional Testing:** Treats a program as a black box. Outputs are verified for conformance to specifications from user's point of view.
- **Structural Testing:** Looks at the implementation details: programming style, control method, source language, database & coding details.
- **Interleaving of functional & Structural testing:**
  - A good program is built in layers from outside.
  - Outside layer is pure system function from user's point of view.
  - Each layer is a structure with its outer layer being its function.
  - Examples:



Dichotomies

- **Interleaving of functional & Structural testing: (contd..)**
  - For a given model of programs, Structural tests may be done first and later the Functional, Or vice-versa. Choice depends on which seems to be the natural choice.
  - Both are useful, have limitations and target different kind of bugs. Functional tests can detect all bugs in principle, but would take infinite amount of time. Structural tests are inherently finite, but cannot detect all bugs.
  - The Art of Testing is how much allocation % for structural vs how much % for functional.

Dichotomies

Dichotomies contd..

3. Designer vs Tester

- ✓ Completely separated in black box testing. Unit testing may be done by either.
- ✓ Artistry of testing is to balance knowledge of design and its biases against ignorance & inefficiencies.
- ✓ Tests are more efficient if the designer, programmer & tester are independent in all of unit, unit integration, component, component integration, system, formal system feature testing.
- ✓ The extent to which test designer & programmer are separated or linked depends on testing level and the context.

#	Programmer / Designer	Tester
1	Tests designed by designers are more oriented towards structural testing and are limited to its limitations.	With knowledge about internal test design, the tester can eliminate useless tests, optimize & do an efficient test design.
2	Likely to be biased.	Tests designed by independent testers are bias-free.
3	Tries to do the job in simplest & cleanest way, trying to reduce the complexity.	Tester needs to suspicious, uncompromising, hostile and obsessed with destroying program.

Dichotomies

Dichotomies contd..

4. Modularity (Design) vs Efficiency

1. system and test design can both be modular.
2. A module implies a size, an internal structure and an interface, Or, in other words.
3. A module (well defined discrete component of a system) consists of internal complexity & interface complexity and has a size.

Dichotomies

#	Modularity	Efficiency
1	Smaller the component easier to understand.	Implies more number of components & hence more # of interfaces increase complexity & reduce efficiency (=> more bugs likely)
2	Small components/modules are repeatable independently with less rework (to check if a bug is fixed).	Higher efficiency at module level, when a bug occurs with small components.
3	Microscopic test cases need individual setups with data, systems & the software. Hence can have bugs.	More # of test cases implies higher possibility of bugs in test cases. Implies more rework and hence less efficiency with microscopic test cases
4	Easier to design large modules & smaller interfaces at a higher level.	Less complex & efficient. (Design may not be enough to understand and implement. It may have to be broken down to implementation level.)

So:

- Optimize the size & balance internal & interface complexity to increase efficiency

- Optimize the test design by setting the scopes of tests & group of tests (modules) to minimize cost of test design, debugging, execution & organizing – without compromising effectiveness.

Dichotomies

Dichotomies contd..

5. Programming in SMALL Vs programming in BIG

- ✓ Impact on the development environment due to the volume of customer requirements.

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

Dichotomies

6. Buyer Vs Builder (customer vs developer organization)

- Buyer & Builder being the same (organization) clouds accountability.
- Separate them to make the accountability clear, even if they are in the same organization.
- The accountability increases motivation for quality.

- The roles of all parties involved are:

- Builder:
  - Designs for & is accountable to the Buyer.
- Buyer:
  - Pays for the system.
  - Hopes to get profits from the services to the User.
- User:
  - Ultimate beneficiary of the system.
  - Interests are guarded by the Tester.
- Tester:
  - *Dedicated to the destruction of the s/w (builder)*
  - Tests s/w in the interests of User/Operator.
- Operator:
  - Lives with: Mistakes of the Builder, Oversights of Tester, Murky specs of Buyer, Complaints of User

- A model for testing - with a project environment - with tests at various levels.
- (1) understand what a project is. (2) look at the roles of the Testing models.

1. PROJECT:

- An Archetypical System (product) allows tests without complications (even for a large project).
- Testing a one shot routine & very regularly used routine is different.

- A model for project in a real world consists of the following 8 components:

1) Application: An online real-time system (with remote terminals) providing timely responses to user requests (for services).

2) Staff: Manageable size of programming staff with specialists in systems design.

3) Schedule: project may take about 24 months from start to acceptance. 6 month maintenance period.

4) Specifications: is good. documented. Undocumented ones are understood well in the team.

A Model for Testing

4) Acceptance test: Application is accepted after a formal acceptance test. At first it's the customer's & then the software design team's responsibility.

5) Personnel: The technical staff comprises of : A combination of experienced professionals & junior programmers (1 – 3 yrs) with varying degrees of knowledge of the application.

6) Standards:
   - Programming, test and interface standard (documented and followed).
   - A centralized standards data base is developed & administrated

A Model for Testing

1. PROJECT: contd …

6) Objectives: (of a project)

- A system is expected to operate profitably for > 10 yrs (after installation).
- Similar systems with up to 75% code in common may be implemented in future.

7) Source: (for a new project) is a combination of

- New Code - up to 1/3rd
- From a previous reliable system - up to 1/3rd
- Re-hosted from another language & O.S. - up to 1/3rd

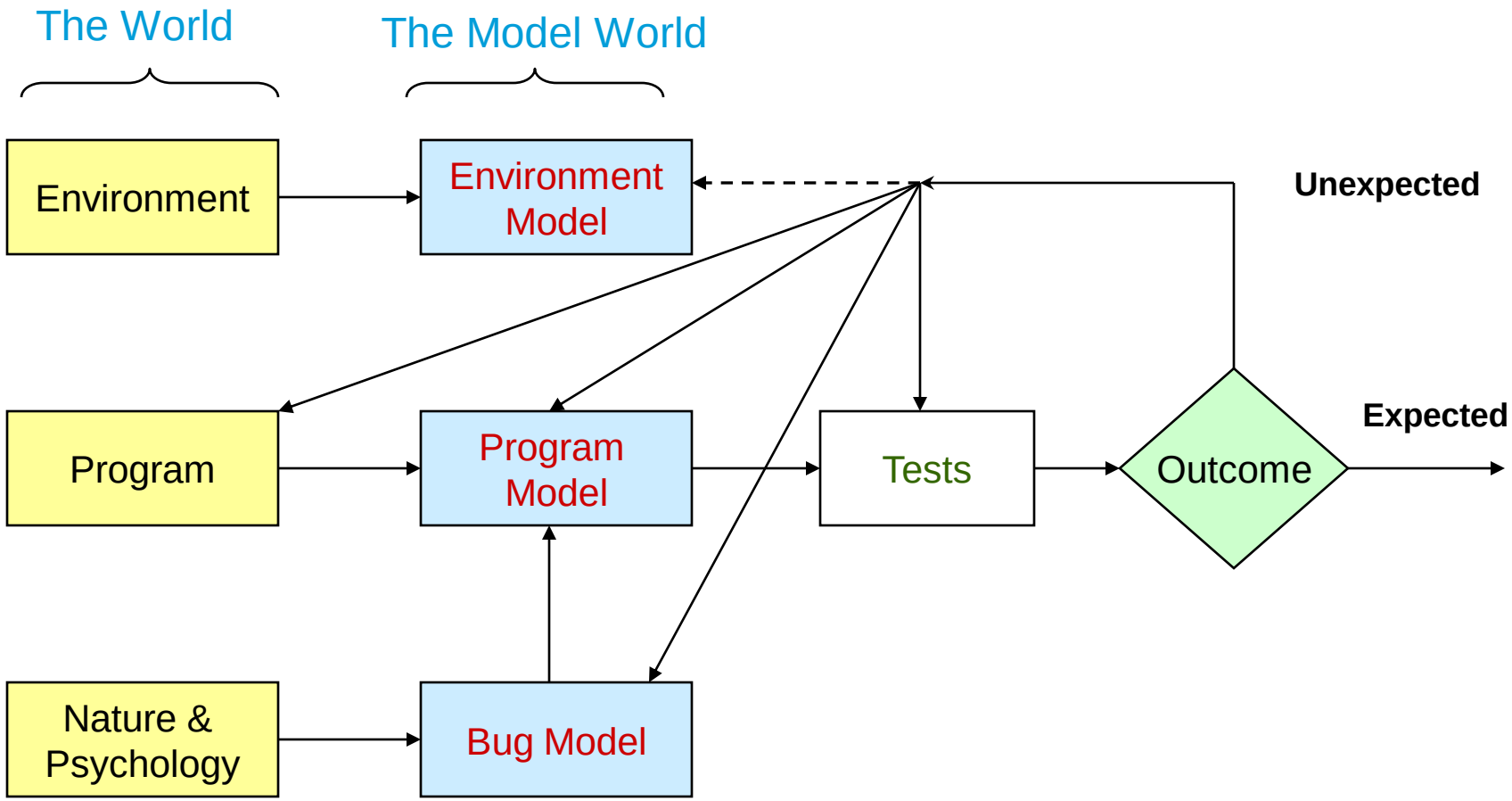
8) History: Typically:

- Developers quit before his/her components are tested.
- Excellent but poorly documented work.
- Unexpected changes (major & minor) from customer may come in
- Important milestones may slip, but the delivery date is met.
- Problems in integration, with some hardware, redoing of some component etc…..

A model project is

☺ A Well Run & Successful Project.
😐 Combination of Glory and Catastrophe.

A Model for Testing



A Model for Testing contd..

2. Roles of Models for Testing

1) Overview:

- Testing process starts with a program embedded in an environment.
- Human nature of susceptibility to error leads to 3 models.
- Create tests out of these models & execute
- Results is expected ⇒ It's okay
  unexpected ⇒ Revise tests and program. Revise bug model and program.

2) Environment: includes

- All hardware & software (firmware, OS, linkage editor, loader, compiler, utilities, libraries) required to make the program run.
- Usually bugs do not result from the environment. (with established h/w & s/w)
- But arise from our understanding of the environment.

3) Program:

- Complicated to understand in detail.
- Deal with a simplified overall view.
- Focus on control structure ignoring processing & focus on processing ignoring control structure.
- If bug's not solved, modify the program model to include more facts, & if that fails, modify the program.

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model)

- Categorize the bugs as initialization, call sequence, wrong variable etc..
- An incorrect spec. may lead us to mistake for a program bug.
- There are 9 Hypotheses regarding Bugs.

a. Benign Bug Hypothesis:

- The belief that the bugs are tame & logical.
- Weak bugs are logical & are exposed by logical means.
- Subtle bugs have no definable pattern.

b. Bug locality hypothesis:

- Belief that bugs are localized.
- Subtle bugs affect that component & external to it.

c. Control Dominance hypothesis:

- Belief that most errors are in control structures, but data flow & data structure errors are common too.
- Subtle bugs are not detectable only thru control structure.

(subtle bugs => from violation of data structure boundaries & data-code separation)

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model) contd ..

d. Code/data Separation hypothesis:

- Belief that the bugs respect code & data separation in HOL programming.
- In real systems the distinction is blurred and hence such bugs exist.

e. Lingua Salvator Est hypothesis:

- Belief that the language syntax & semantics eliminate most bugs.
- But, such features may not eliminate Subtle Bugs.

f. Corrections Abide hypothesis:

- Belief that a corrected bug remains corrected.
- Subtle bugs may not. For e.g.
  A correction in a data structure ‘DS’ due to a bug in the interface between modules A & B, could impact module C using ‘DS’.

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model) contd ..

g. Silver Bullets hypothesis:

- Belief that - language, design method, representation, environment etc. grant immunity from bugs.
- Not for subtle bugs.
- *Remember the pesticide paradox.*

h. Sadism Suffices hypothesis:

- Belief that a sadistic streak, low cunning & intuition (by independent testers) are sufficient to extirpate most bugs.
- Subtle & tough bugs are may not be … - these need methodology & techniques.

i. Angelic Testers hypothesis:

- Belief that testers are better at test design than programmers at code design.

A Model for Testing contd..

2. Roles of Models for Testing contd..

5) Tests:

- Formal procedures.
- Input preparation, outcome prediction and observation, documentation of test, execution & observation of outcome are subject to errors.
- An unexpected test result may lead us to revise the test and test model.

6) Testing & Levels:

3 kinds of tests (with different objectives)

1) Unit & Component Testing

a. A unit is the smallest piece of software that can be compiled/assembled, linked, loaded & put under the control of test harness / driver.
b. Unit testing - verifying the unit against the functional specs & also the implementation against the design structure.
c. Problems revealed are unit bugs.

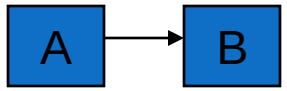
d. Component is an integrated aggregate of one or more units (even entire system)
e. Component testing - verifying the component against functional specs and the implemented structure against the design.
f. Problems revealed are component bugs.

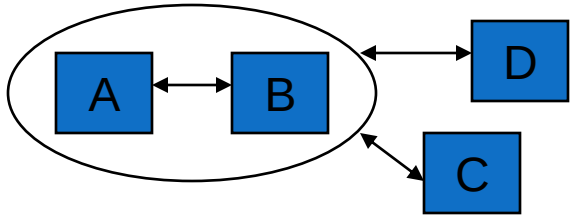
A Model for Testing contd..

2. Roles of Models for Testing contd …

2) Integration Testing:

- Integration is a process of aggregation of components into larger components.
- Verification of consistency of interactions in the combination of components.
- Examples of integration testing are improper call or return sequences, inconsistent data validation criteria & inconsistent handling of data objects.
- Integration testing & Testing Integrated Objects are different





- Sequence of Testing:
  - Unit/Component tests for A, B. Integration tests for A & B. Component testing for (A,B) component

A Model for Testing contd..

2. Roles of Models for Testing contd …

3) System Testing

a. System is a big component.
b. Concerns issues & behaviors that can be tested at the level of entire or major part of the integrated system.
c. Includes testing for performance, security, accountability, configuration sensitivity, start up & recovery

After understanding a Project, Testing Model, now let's see finally,

Role of the Model of testing :

- Used for the testing process until system behavior is correct or until the model is insufficient (for testing).
- Unexpected results may force a revision of the model.
- Art of testing consists of creating, selecting, exploring and revising models.
- The model should be able to express the program.

Consequences of Bugs

Consequences: (how bugs may affect users)

These range from mild to catastrophic on a 10 point scale.

- Mild
  - Aesthetic bug such as misspelled output or mal-aligned print-out.
- Moderate
  - Outputs are misleading or redundant impacting performance.
- Annoying
  - Systems behavior is dehumanizing for e.g. names are truncated/modified arbitrarily, bills for $0.0 are sent.
  - Till the bugs are fixed operators must use unnatural command sequences to get proper response.
- Disturbing
  - Legitimate transactions refused.
  - For e.g. ATM machine may malfunction with ATM card / credit card.
- Serious
  - Losing track of transactions & transaction events. Hence accountability is lost.

Consequences of Bugs

Consequences contd …

- Very serious
  System does another transaction instead of requested e.g. Credit another account, convert withdrawals to deposits.
- Extreme
  - Frequent & Arbitrary - not sporadic & unusual.
- Intolerable
  - Long term unrecoverable corruption of the Data base.
    (not easily discovered and may lead to system down.)
- Catastrophic
  - System fails and shuts down.
- Infectious
  - Corrupts other systems, even when it may not fail.

Consequences of Bugs

Consequences contd …

Assignment of severity

- Assign flexible & relative rather than absolute values to the bug (types).
- Number of bugs and their severity are factors in determining the quality quantitatively.
- Organizations design & use quantitative, quality metrics based on the above.

- Parts are weighted depending on environment, application, culture, correction cost, current SDLC phase & other factors.

- Nightmares

  - Define the nightmares – that could arise from bugs – for the context of the organization/application.

  - Quantified nightmares help calculate importance of bugs.
  - That helps in making a decision on when to stop testing & release the product.

Consequences of Bugs

Consequences contd …

When to stop Testing

1. List all nightmares in terms of the symptoms & reactions of the user to their consequences.
2. Convert the consequences of into a cost. There could be rework cost. *(but if the scope extends to the public, there could be the cost of lawsuits, lost business, nuclear reactor meltdowns.)*
3. Order these from the costliest to the cheapest. Discard those with which you can live with.
4. Based on experience, measured data, intuition, and published statistics postulate the kind of bugs causing each symptom. This is called 'bug design process'. A bug type can cause multiple symptoms.
5. Order the causative bugs by decreasing probability *(judged by intuition, experience, statistics etc.)*. Calculate the importance of a bug type as:

$$\text{Importance of bug type } j = \sum_{\text{all } k} C_{jk} \; P_{jk}$$

where,

C_{jk} = cost due to bug type j causing nightmare k

P_{jk} = probability of bug type j causing nightmare k

$$\text{(Cost due to all bug types } = \sum_{\text{all } k} \sum_{\text{all } j} C_{jk} \; P_{jk} \text{)}$$

Consequences of Bugs

6. Rank the bug types in order of decreasing importance.

7. Design tests & QA inspection process with most effective against the most important bugs.

8. If a test is passed or when correction is done for a failed test, some nightmares disappear. As testing progresses, revise the probabilities & nightmares list as well as the test strategy.

9. Stop testing when probability (importance & cost) proves to be inconsequential.

This procedure could be implemented formally in SDLC.

Important points to Note:

- Designing a reasonable, finite # of tests with high probability of removing the nightmares.
- Test suites wear out.
  - As programmers improve programming style, QA improves.
  - Hence, know and update test suites as required.

Taxonomy of Bugs ..

we had seen the:

1. Importance of Bugs - statistical quantification of impact
2. Consequences of Bugs - causes, nightmares, to stop testing

We will now see the:

3. Taxonomy of Bugs - along with some remedies

In order to be able to create an organization's own Bug Importance Model for the sake of controlling associated costs…

Taxonomy of Bugs .. and remedies

Reference of IEEE Taxonomy: IEEE 87B

- Why Taxonomy ?

  To study the consequences, nightmares, probability, importance, impact and the methods of prevention and correction.

- Adopt known taxonomy to use it as a statistical framework on which your testing strategy is based.

- 6 main categories with sub-categories..

1)Requirements, Features, Functionality Bugs	**24.3% bugs**
2)Structural Bugs	**25.2%**
3)Data Bugs	**22.3%**
4)Coding Bugs	**9.0%**
5)Interface, Integration and System Bugs	**10.7%**
6)Testing & Test Design Bugs	**2.8 %**

Taxonomy of Bugs .. and remedies

Reference of IEEE Taxonomy: IEEE 87B

1) Requirements, Features, Functionality Bugs

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

I. Requirements & Specs.

- Incompleteness, ambiguous or self-contradictory
- Analyst's assumptions not known to the designer

- Some thing may miss when specs change
- These are expensive: introduced early in SDLC and removed at the last

II. Feature Bugs

- Specification problems create feature bugs
- Wrong feature bug has design implications

- Missing feature is easy to detect & correct
- Gratuitous enhancements can accumulate bugs, if they increase complexity

- Removing features may foster bugs

Taxonomy of Bugs .. and remedies

1) Requirements, Features, Functionality Bugs contd..

III. Feature Interaction Bugs

- Arise due to unpredictable interactions between feature groups or individual features. The earlier removed the better as these are costly if detected at the end.
- Examples: call forwarding & call waiting. Federal, state & local tax laws.
- No magic remedy. Explicitly state & test important combinations

Remedies

- Use high level formal specification languages to eliminate human-to-human communication
- It's only a short term support & not a long term solution

- Short-term Support:
  - Specification languages formalize requirements & so automatic test generation is possible. It's cost-effective.

- Long-term support:
  - Even with a great specification language, problem is not eliminated, but is shifted to a higher level. Simple ambiguities & contradictions may only be removed, leaving tougher bugs.

Testing Techniques

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

Taxonomy of Bugs .. and remedies

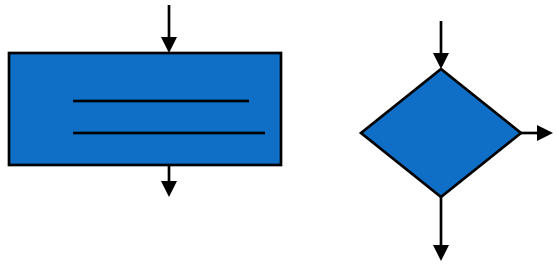
2. Structural Bugs

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

I. Control & Sequence bugs
II. Logic Bugs
III. Processing bugs
IV. Initialization bugs
V. Data flow bugs & anomalies

1. Control & Sequence Bugs:

- Paths left out, unreachable code, spaghetti code, and pachinko code.
- Improper nesting of loops, Incorrect loop-termination or look-back, ill-conceived switches.
- Missing process steps, duplicated or unnecessary processing, rampaging GOTOs.
- Novice programmers.
- Old code (assembly language & Cobol)



Prevention and Control:

- Theoretical treatment and,
- Unit, structural, path, & functional testing.

Taxonomy of Bugs .. and remedies

2. Structural Bugs contd..

II. Logic Bugs

- Misunderstanding of the semantics of the control structures & logic operators
- Improper layout of cases, including impossible & ignoring necessary cases,

- Using a look-alike operator, improper simplification, confusing Ex-OR with inclusive OR.
- Deeply nested conditional statements & using many logical operations in 1 stmt.

Prevention and Control:

Logic testing, careful checks, functional testing

III. Processing Bugs

- Arithmetic, algebraic, mathematical function evaluation, algorithm selection & general. processing, data type conversion, ignoring overflow, improper use of relational operators.

Prevention

- Caught in Unit Testing & have only localized effect
- Domain testing methods

Taxonomy of Bugs .. and remedies

Structural bugs contd..

IV. Initialization Bugs

- Forgetting to initialize work space, registers, or data areas.
- Wrong initial value of a loop control parameter.

- Accepting a parameter without a validation check.
- Initialize to wrong data type or format.
- Very common.

Remedies (prevention & correction)

- Programming tools, Explicit declaration & type checking in source language, preprocessors.
- Data flow test methods help design of tests and debugging.

V. Dataflow Bugs & Anomalies

- Run into an un-initialized variable.
- Not storing modified data.

- Re-initialization without an intermediate use.
- Detected mainly by execution (testing).

Remedies (prevention & correction)

- Data flow testing methods & matrix based testing methods.

Taxonomy of Bugs .. and remedies

3. Data Bugs

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

I. Generic Data Bugs

II. Dynamic Data Vs Static Data

III. Information, Parameter, and Control Bugs

IV. Contents, Structure & Attributes related Bugs

Taxonomy of Bugs .. and remedies

Data Bugs contd…

I. Generic Data Bugs

- Due to data object specs., formats, # of objects & their initial values.
- Common as much as in code, especially as the code migrates to data.
- Data bug introduces an operative statement bug & is harder to find.
- Generalized components with reusability – when customized from a large parametric data to specific installation.

Remedies (prevention & correction):

- Using control tables in lieu of code facilitates software to handle many transaction types with fewer data bugs. Control tables have a hidden programming language in the database.
- Caution - there's no compiler for the hidden control language in data tables

Taxonomy of Bugs .. and remedies

II. Dynamic Data Vs Static Data

Dynamic Data Bugs	Static Data Bugs
Transitory. Difficult to catch.	Fixed in form & content.
Due to an error in a shared storage object initialization.	Appear in source code or data base, directly or indirectly
Due to unclean / leftover garbage in a shared resource.	Software to produce object code creates a static data table – bugs possible
Examples	**Examples**
Generic & shared variable	Telecom system software: generic parameters, a generic large program & site adapter program to set parameter values, build data declarations etc.
Shared data structure	Postprocessor : to install software packages. Data is initialized at run time – with configuration handled by tables.
Prevention Data validation, unit testing	**Prevention** Compile time processing Source language features

Taxonomy of Bugs .. and remedies

Data Bugs contd..

III. Information, Parameter, and Control Bugs

Static or dynamic data can serve in any of the three forms. It is a matter of perspective.

What is information can be a data parameter or control data else where in a program.

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

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

Bugs

- Usually simple bugs and easy to catch.
- When a subroutine (with good data validation code) is modified, forgetting to update the data validation code, results in these bugs.

Preventive Measures (prevention & correction)

- Proper Data validation code.

Taxonomy of Bugs .. and remedies

Data Bugs contd..

IV. Contents, Structure & Attributes related Bugs

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

Bugs

- Severity & subtlety increases from contents to attributes as they get less formal.
- Structural bugs may be due to wrong declaration or when same contents are interpreted by multiple structures differently (different mapping).

- Attribute bugs are due to misinterpretation of data type, probably at an interface

Preventive Measures (prevention & correction)

- Good source lang. documentation & coding style (incl. data dictionary).
- Data structures be globally administered. Local data migrates to global.

- Strongly typed languages prevent mixed manipulation of data.
- In an assembly lang. program, use field-access macros & not directly accessing any field.

Taxonomy of Bugs .. and remedies

4. Coding Bugs

- Coding errors create other kinds of bugs.
- Syntax errors are removed when compiler checks syntax.
- Coding errors
  typographical, misunderstanding of operators or statements or could be just arbitrary.
- Documentation Bugs
  - Erroneous comments could lead to incorrect maintenance.
  - Testing techniques cannot eliminate documentation bugs.
  - Solution:

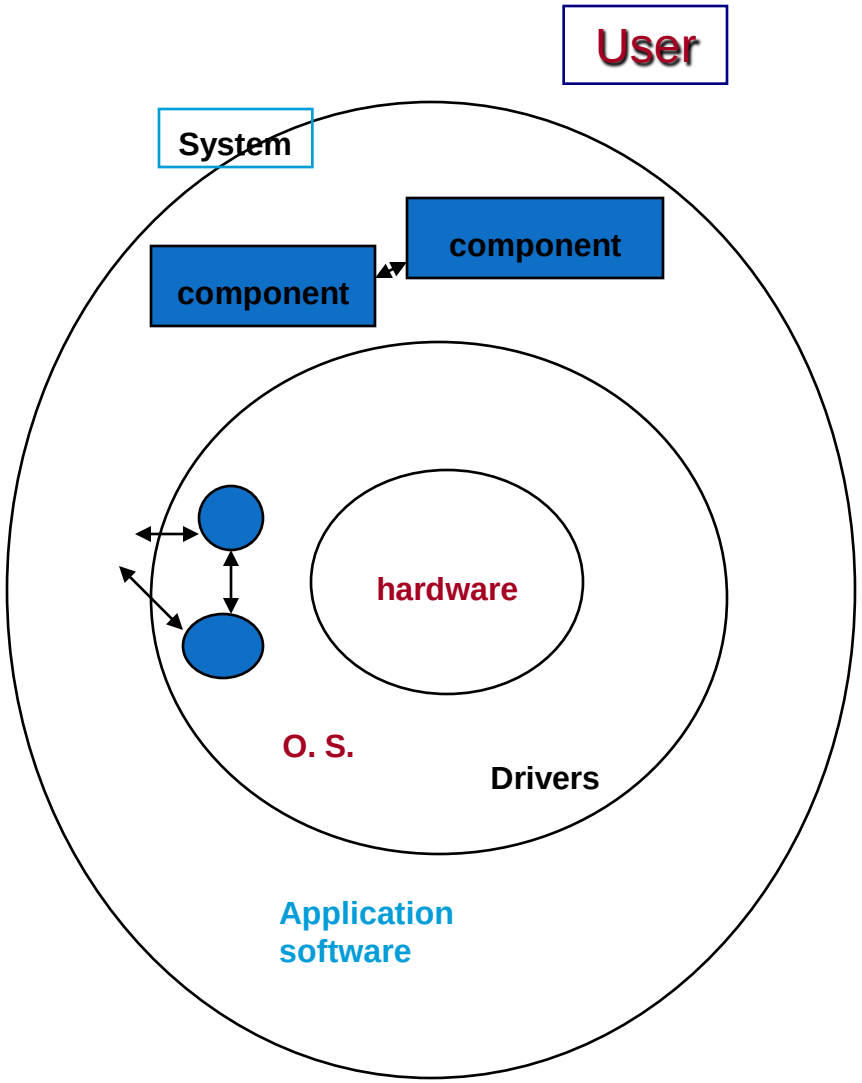
    Inspections, QA, automated data dictionaries & specification systems.

Taxonomy of Bugs .. and remedies

5. Interface, Integration and Systems Bugs

There are 9 types of bugs of this type.

1) External Interfaces
2) Internal Interfaces
3) Hardware Architecture Bugs
4) Operating System Bugs
5) Software architecture bugs
6) Control & Sequence bugs
7) Resource management bugs
8) Integration bugs
9) System bugs



Taxonomy of Bugs .. and remedies

5. Interface, Integration and Systems Bugs contd..

1) External Interfaces

- Means to communicate with the world: drivers, sensors, input terminals, communication lines.
- Primary design criterion should be - robustness.

- Bugs: invalid timing, sequence assumptions related to external signals, misunderstanding external formats and no robust coding.
- Domain testing, syntax testing & state testing are suited to testing external interfaces.

2) Internal Interfaces

- Must adapt to the external interface.
- Have bugs similar to external interface

- Bugs from improper
  - Protocol design, input-output formats, protection against corrupted data, subroutine call sequence, call-parameters.

- Remedies (prevention & correction):

- Test methods of domain testing & syntax testing.
- Good design & standards: good trade off between # of internal interfaces & complexity of the interface.
- Good integration testing is to test all internal interfaces with external world.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

3) Hardware Architecture Bugs:

- A s/w programmer may not see the h/w layer / architecture.
- S/w bugs originating from hardware architecture are due to misunderstanding of how h/w works.
- **Bugs are due to errors in:**
  - Paging mechanism, address generation
  - I/O device instructions, device status code, device protocol
  - Expecting a device to respond too quickly, or to wait for too long for response, assuming a device is initialized, interrupt handling, I/O device address
  - H/W simultaneity assumption, H/W race condition ignored, device data format error etc..
- Remedies (prevention & correction):
  - Good software programming & Testing.
  - Centralization of H/W interface software.
  - Nowadays hardware has special test modes & test instructions to test the H/W function.
  - An elaborate H/W simulator may also be used.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

4) Operating System Bugs:

- **Due to:**
  - Misunderstanding of H/W architecture & interface by the O. S.
  - Not handling of all H/W issues by the O. S.
  - Bugs in O. S. itself and some corrections may leave quirks.
  - Bugs & limitations in O. S. may be buried some where in the documentation.
- **Remedies (prevention & correction):**
  - Same as those for H/W bugs.
  - Use O. S. system interface specialists
  - Use explicit interface modules or macros for all O.S. calls.
  - The above may localize bugs and make testing simpler.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

5) Software Architecture Bugs: (called Interactive)

The subroutines pass thru unit and integration tests without detection of these bugs. Depend on the Load, when the system is stressed. These are the most difficult to find and correct.

- **Due to:**
  - Assumption that there are no interrupts, Or, Failure to block or unblock an interrupt.
  - Assumption that code is re-entrant or not re-entrant.
  - Bypassing data interlocks, Or, Failure to open an interlock.
  - Assumption that a called routine is memory resident or not.
  - Assumption that the registers and the memory are initialized, Or, that their content did not change.
  - Local setting of global parameters & Global setting of local parameters.
- **Remedies**:
  - Good design for software architecture.
- **Test Techniques**
  - All test techniques are useful in detecting these bugs, Stress tests in particular.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

6) Control & Sequence Bugs:

- **Due to:**
  - Ignored timing
  - Assumption that events occur in a specified sequence.
  - Starting a process before its prerequisites are met.
  - Waiting for an impossible combination of prerequisites.
  - Not recognizing when prerequisites are met.
  - Specifying wrong priority, Program state or processing level.
  - Missing, wrong, redundant, or superfluous process steps.

- **Remedies**:
  - Good design.
  - highly structured sequence control - useful
  - Specialized internal sequence-control mechanisms such as an internal job control language – useful.
  - Storage of Sequence steps & prerequisites in a table and interpretive processing by control processor or dispatcher - easier to test & to correct bugs.

- **Test Techniques**
  - Path testing as applied to Transaction Flow graphs is effective.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

7) Resource Management Problems:

- Resources: Internal: Memory buffers, queue blocks etc. External: discs etc.

- **Due to:**
  - Wrong resource used (when several resources have similar structure or different kinds of resources in the same pool).
  - Resource already in use, or deadlock
  - Resource not returned to the right pool, Failure to return a resource. Resource use forbidden to the caller.

- **Remedies**:
  - Design: keeping resource structure simple with fewest kinds of resources, fewest pools, and no private resource mgmt.
  - Designing a complicated resource structure to handle all kinds of transactions to save memory is not right.
  - Centralize management of all resource pools thru managers, subroutines, macros etc.

- **Test Techniques**
  - Path testing, transaction flow testing, data-flow testing & stress testing.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

8) Integration Bugs:

Are detected late in the SDLC and cause several components and hence are very costly.

- **Due to:**
  - Inconsistencies or incompatibilities between components.
  - Error in a method used to directly or indirectly transfer data between components. Some communication methods are: data structures, call sequences, registers, semaphores, communication links, protocols etc..
- **Remedies**:
  - Employ good integration strategies. ***
- **Test Techniques**
  - Those aimed at interfaces, domain testing, syntax testing, and data flow testing when applied across components.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

9) System Bugs:

- Infrequent, but are costly

- **Due to:**
  - Bugs not ascribed to a particular component, but result from the totality of interactions among many components such as:
    programs, data, hardware, & the O.S.

- **Remedies:**
  - Thorough testing at all levels and the test techniques mentioned below

- **Test Techniques**
  - Transaction-flow testing.
  - All kinds of tests at all levels as well as integration tests - are useful.

6. Testing & Test Design Bugs

Bugs in Testing (scripts or process) are not software bugs.

It's difficult & takes time to identify if a bug is from the software or from the test script/procedure.

1) Bugs could be due to:

- Tests require code that uses complicated scenarios & databases, to be executed.
- Though an independent functional testing provides an un-biased point of view, this lack of bias may lead to an incorrect interpretation of the specs.
- Test Criteria
  - Testing process is correct, but the criterion for judging software's response to tests is incorrect or impossible.
  - If a criterion is quantitative (throughput or processing time), the measurement test can perturb the actual value.

Taxonomy of Bugs .. and remedies

Testing & Test Design Bugs contd…

- Remedies:

1. Test Debugging:

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

2. Test Quality Assurance:

To monitor quality in independent testing and test design.

3. Test Execution Automation:

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

4. Test Design Automation:

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

Taxonomy of Bugs .. and remedies

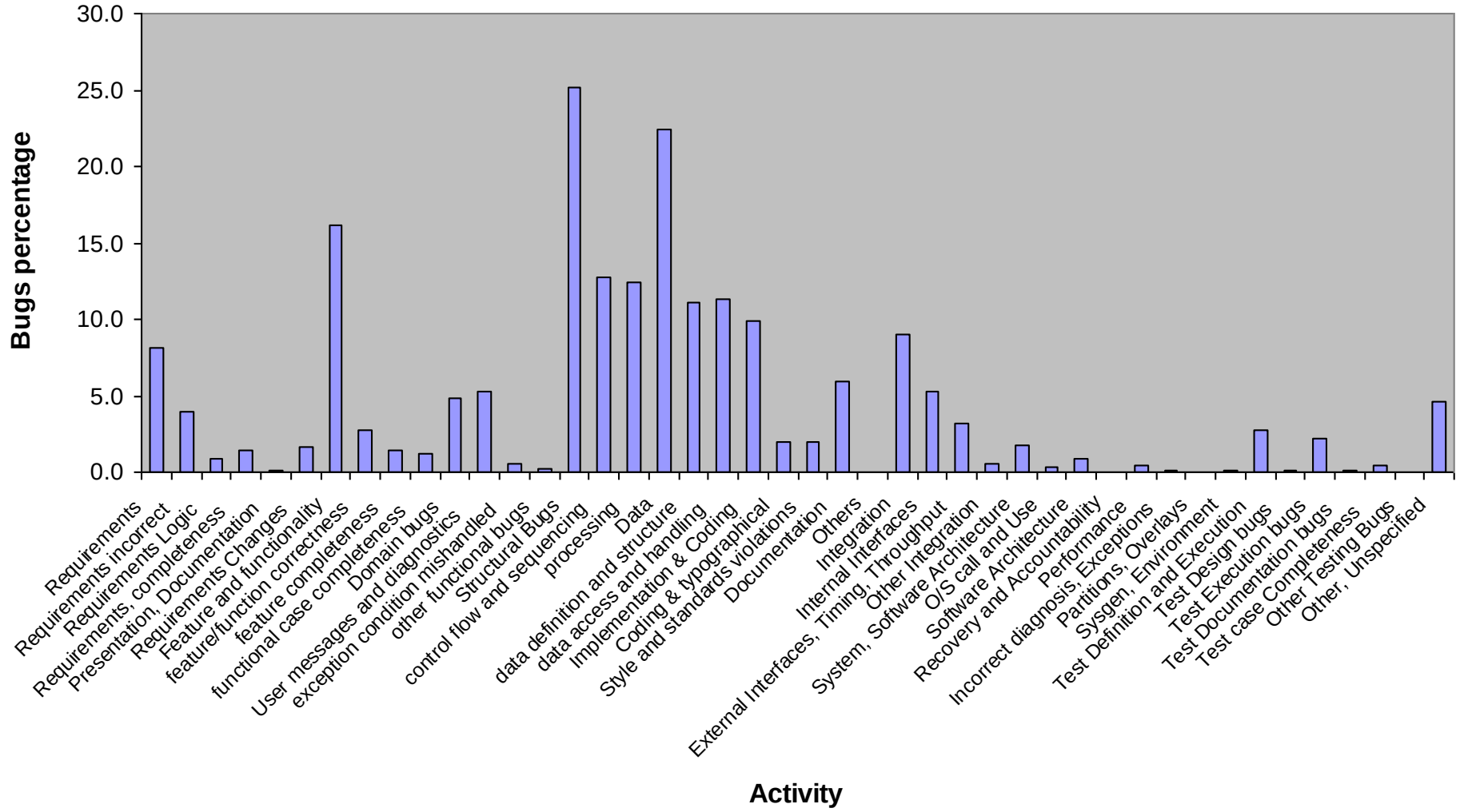
A word on productivity

At the end of a long study on taxonomy, we could say

Good design inhibits bugs and is easy to test. The two factors are multiplicative and results in high productivity.

Good test works best on good code and good design.

Good test cannot do a magic on badly designed software.



Source: Boris Beizer

Control Flow Graphs and Path Testing

Introduction

Path Testing Definition

A family of structural test techniques based on judiciously selecting a set of test paths through the programs.

- ✓ **Goal:** Pick enough paths to assure that every source statement is executed at least once.
- ✓ It is a measure of thoroughness of **code coverage.**
- ✓ It is used most for unit testing on new software.
- ✓ Its effectiveness reduces as the software size increases.
- ✓ We use Path testing techniques indirectly.
- ✓ Path testing concepts are used **in** and **along** with other testing techniques

Code Coverage: During unit testing: # stmts executed at least once / total # stmts

Path Testing contd..

Assumptions:

- Software takes a different path than intended due to some error.
- Specifications are correct and achievable.
- Processing bugs are only in control flow statements
- Data definition & access are correct

Observations

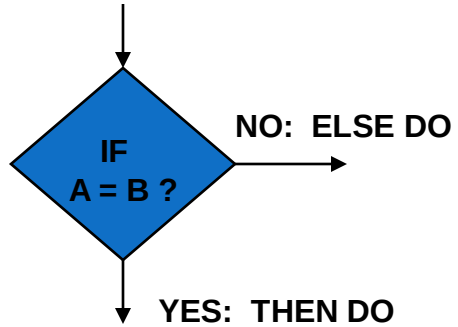
- Structured programming languages need less of path testing.
- Assembly language, Cobol, Fortran, Basic & similar languages make path testing necessary.

Control Flow Graph

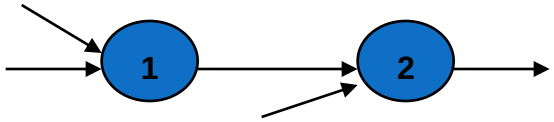
A simplified, abstract, and graphical representation of a program's control structure using process blocks, decisions and junctions.

Process Block

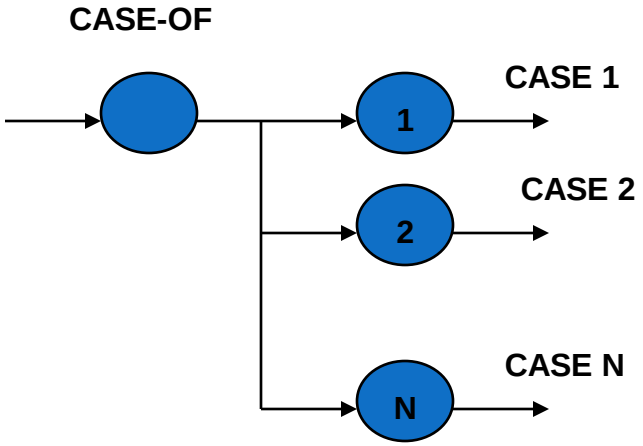
Decisions



Junctions



Case Statement



Control Flow Graph Elements

Control Flow Graphs and Path Testing

Control Flow Graph Elements:

Process Block:

- A sequence of program statements uninterrupted by decisions or junctions with a single entry and single exit.

Junction:

- A point in the program where control flow can merge (into a node of the graph)
- Examples: target of GOTO, Jump, Continue

Decisions:

- A program point at which the control flow can diverge *(based on evaluation of a condition)*.
- Examples: IF stmt. Conditional branch and Jump instruction.

Case Statements:

- A Multi-way branch or decision.
- Examples: In assembly language: jump addresses table, Multiple GOTOs, Case/Switch

- For test design, Case statement and decision are similar.

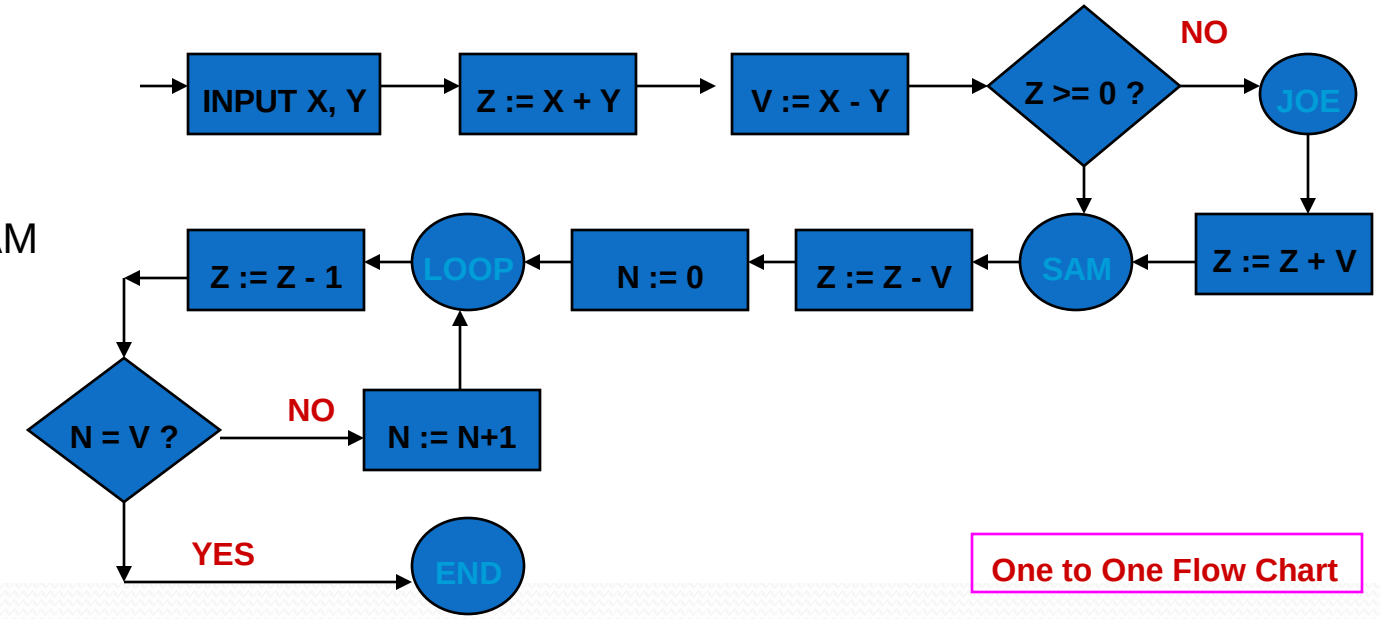
Control Flow Graph Vs Flow Charts

Control Flow Graph	Flow Chart
Compact representation of the program	Usually a multi-page description
Focuses on Inputs, Outputs, and the control flow into and out of the block.	Focuses on the process steps inside
Inside details of a process block are not shown	Every part of the process block are drawn

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

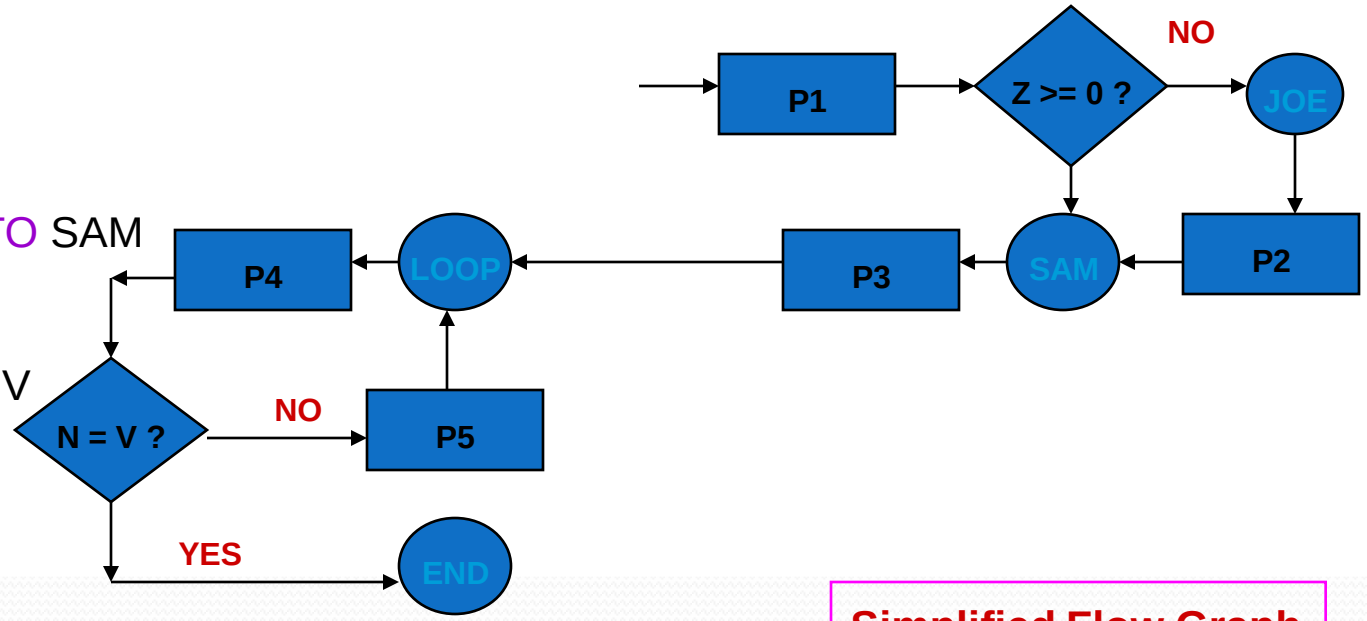


One to One Flow Chart

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

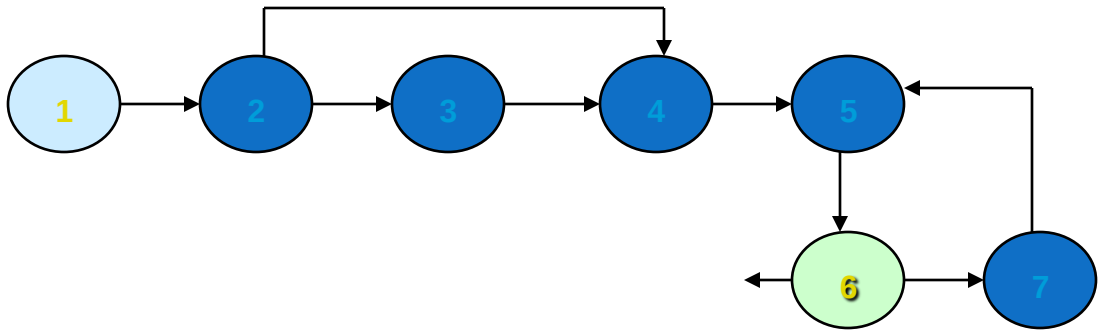


Simplified Flow Graph

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

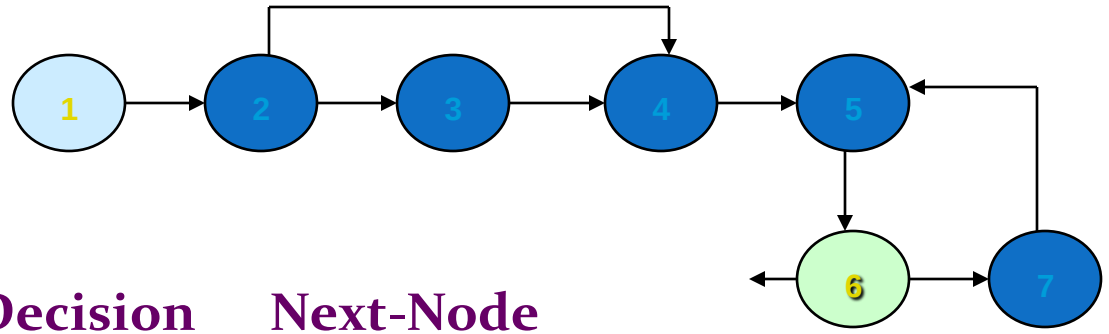
```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```



Simplified Flow Graph

Control Flow Graphs and Path Testing

Linked List Notation of a Control Flow Graph



Node	Processing, label, Decision	Next-Node
1	(BEGIN; INPUT X, Y; Z := X+Y ; V := X-Y)	: 2
2	(Z >= 0 ?)	: 4 (TRUE)
		: 3 (FALSE)
3	(JOE: Z := Z + V)	: 4
4	(SAM: Z := Z – V; N := 0)	: 5
5	(LOOP; Z := Z -1)	: 6
6	(N = V ?)	: 7 (FALSE)
		: END (TRUE)
7	(N := N + 1)	: 5

Path Testing Concepts

1. **Path** is a sequence of statements starting at an entry, junction or decision and ending at another, or possibly the same junction or decision or an exit point.

   Link is a single process (*block*) in between two nodes.

   Node is a junction or decision.

   Segment is a sequence of links. A path consists of many segments.

   Path segment is a succession of consecutive links that belongs to the same path. (3,4,5)

   Length of a path is measured by # of links in the path or # of nodes traversed.

   Name of a path is the set of the names of the nodes along the path. (1,2,3 4,5, 6)
   (1,2,3,4, 5,6,7, 5,6,7, 5,6)

   Path-Testing Path is an “entry to exit” path through a processing block.

Path Testing Concepts..

2. Entry / Exit for a routines, process blocks and nodes.

Single entry and single exit routines are preferable.

Called well-formed routines.

Formal basis for testing exists.

Tools could generate test cases.

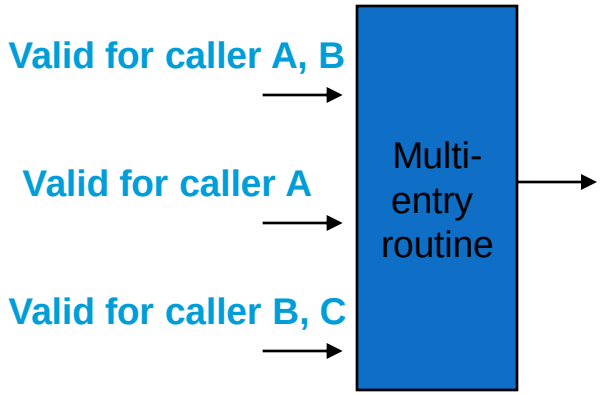
Control Flow Graphs and Path Testing

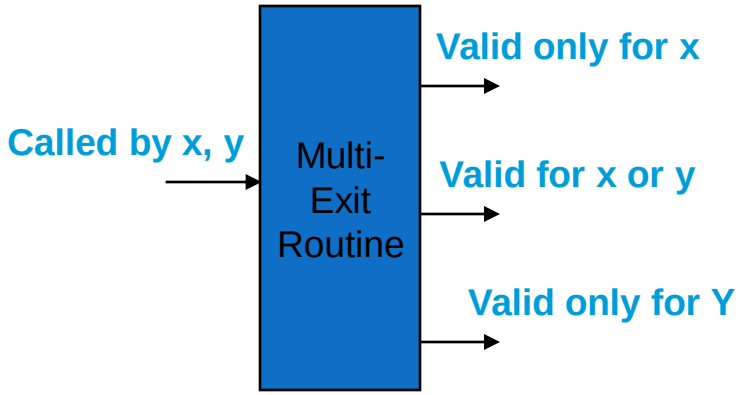
Path Testing Concepts..

Multi-entry / Multi-exit routines: (ill-formed)

- A Weak approach: Hence, convert it to single-entry / single-exit routine.
- Integration issues:

  Large # of inter-process interfaces. Creates problem in Integration.
  More # test cases and also a formal treatment is more difficult.





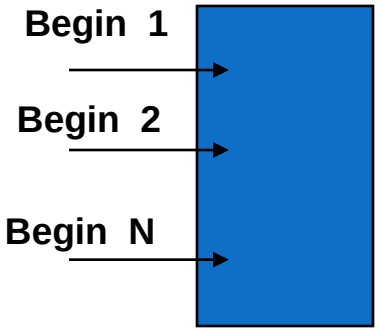
- Theoretical and tools based issues
  - A good formal basis does not exist.
  - Tools may fail to generate important test cases.

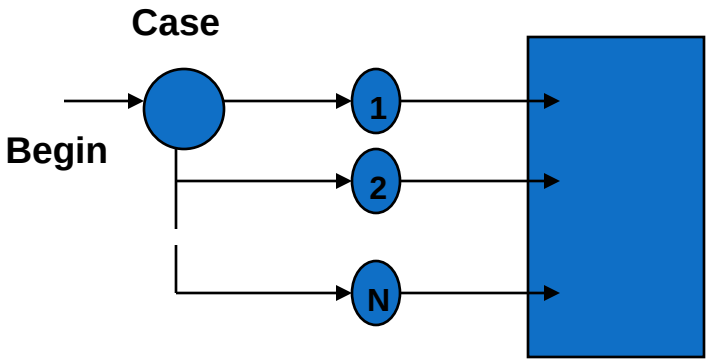
Control Flow Graphs and Path Testing

Path Testing Concepts contd..

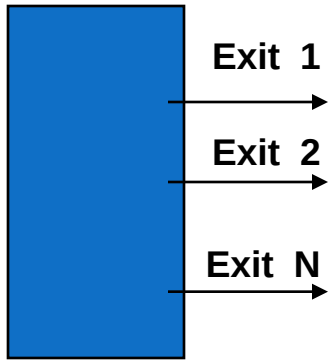
Convert a multi-entry / exit routine to a single entry / exit routine:

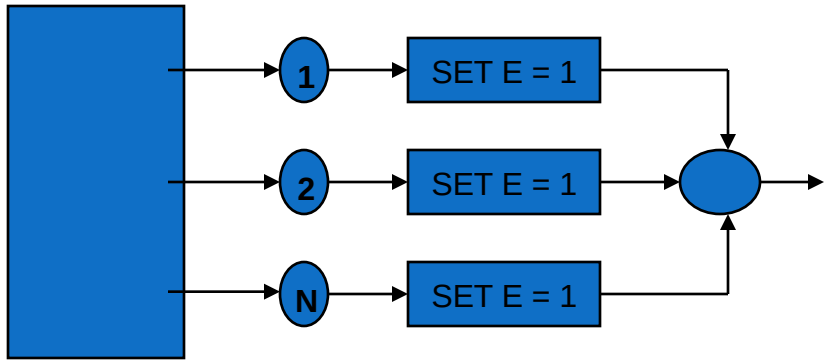
- Use an entry parameter and a case statement at the entry => single-entry





- Merge all exits to Single-exit point after setting one exit parameter to a value.





Path Testing Concepts contd..

Test Strategy for Multi-entry / exit routines

1. Get rid of them.
2. Control those you cannot get rid of.

3. Convert to single entry / exit routines.
4. Do unit testing by treating each entry/exit combination as if it were a completely different routine.

5. Recognize that integration testing is heavier
6. Understand the strategies & assumptions in the automatic test generators and confirm that they do (or do not) work for multi-entry/multi exit routines.

Path Testing Concepts

3. Fundamental Path Selection Criteria

A minimal set of paths to be able to do complete testing.

- Each pass through a routine from entry to exit, as one traces through it, is a **potential** path.
- The above includes the tracing of 1**..**n times tracing of an interactive block each separately.
- **Note**: A bug could make a mandatory path not executable or could create new paths not related to processing.

Complete Path Testing prescriptions:

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

- ♦ Point 1 => point 2 and 3.
- ♦ Point 1 is impractical.
- ♦ Point 2 & 3 are not the same
- ♦ For a structured language, Point 3 => Point 2

Path Testing Concepts

Path Testing Criteria :

1. **Path Testing (P_∞):**

   Execute all possible control flow paths thru the program; but typically restricted to entry-exit paths.

   Implies 100% path coverage. Impossible to achieve.

2. **Statement Testing (P_1)** :

   Execute all statements in the program at least once under the some test.
   100% statement coverage => 100% node coverage.
   Denoted by **C1**
   C1 is a minimum testing requirement in the IEEE unit test standard: ANSI 87B.

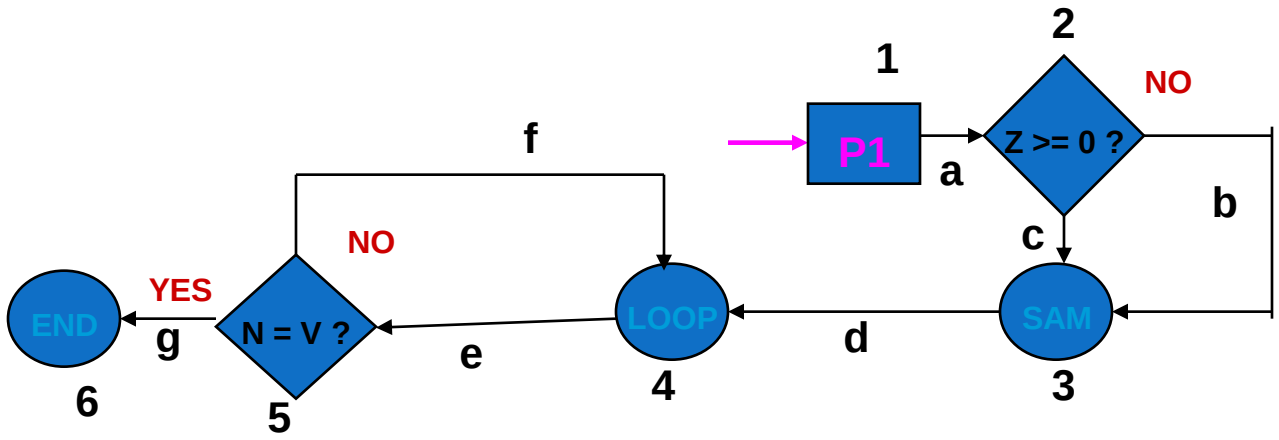
3. **Branch Testing (P_2)** :

   Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   Denoted by **C2**
   Objective: 100% branch coverage and 100% Link coverage.

   For well structured software, branch testing & coverage include statement coverage

Control Flow Graphs and Path Testing

Picking enough (the fewest) paths for achieving C1+C2



1. Does every decision have Y & N (C2)?
2. Are call cases of case statement marked (C2)?
3. Is every three way branch covered (C2)?
4. Is every link covered at least once (C1)?

Make small changes in the path changing only 1 link or node at a time.

Control Flow Graphs and Path Testing

Revised path selection Rules

1. Pick the simplest and functionally sensible entry/exit path
2. Pick additional paths as small variations from previous paths. (pick those with no loops, shorter paths, simple and meaningful)
3. Pick additional paths but without an obvious functional meaning (only to achieve C1+C2 coverage).
4. Be comfortable with the chosen paths. play hunches, use intuition to achieve C1+C2
5. Don’t follow rules slavishly – except for coverage

4. Testing of Paths involving loops

Bugs in iterative statements apparently are not discovered by C1+C2.
But by testing at the boundaries of loop variable.

Types of Iterative statements

1. **Single loop statement.**
2. **Nested loops.**
3. **Concatenated Loops.**
4. **Horrible Loops**

Let us denote the Minimum # of iterations by n_{min}
the Maximum # of iterations by n_{max}
the value of loop control variable by V
the #of test cases by T
the # of iterations carried out by n

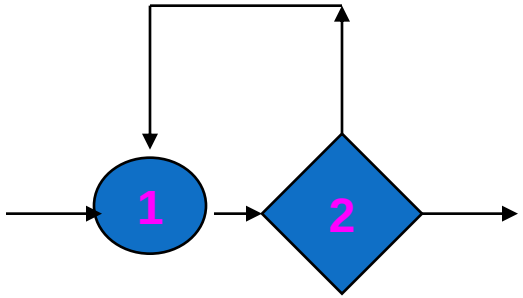
- Later, we analyze the Loop-Testing times

Control Flow Graphs and Path Testing

Testing of path involving loops…

1. Testing a Single Loop Statement (three cases)

Case 1. $n_{min} = 0$, $n_{max} = N$, no excluded values



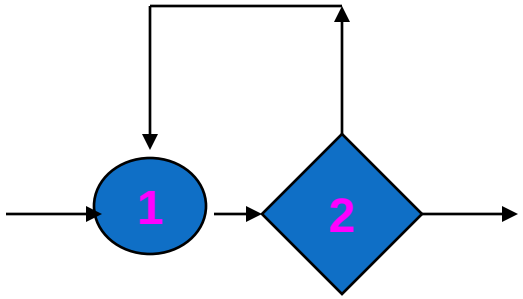
1. Bypass the loop.
   If you can't, there is a bug, $n_{min} \neq 0$ or a wrong case.
2. Could the value of loop (control) variable V be negative?
   could it appear to specify a –ve n ?
3. Try one pass through the loop statement: n = 1
4. Try two passes through the loop statement: n = 2
   To detect initialization data flow anomalies:
   Variable defined & not used in the loop, or
   Initialized in the loop & used outside the loop.
5. Try n = typical number of iterations : $n_{min} < n < n_{max}$
6. Try n = n_{max} -1

7. Try n = n_{max}
8. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Control Flow Graphs and Path Testing

Testing of path involving loops…

Case 2. n_{min} = +ve, n_{max} = N, no excluded values



1. Try n_{min} - 1
   Could the value of loop (control) variable V be < n_{min}?
   What prevents that ?
2. Try n_{min}
3. Try n_{min} + 1

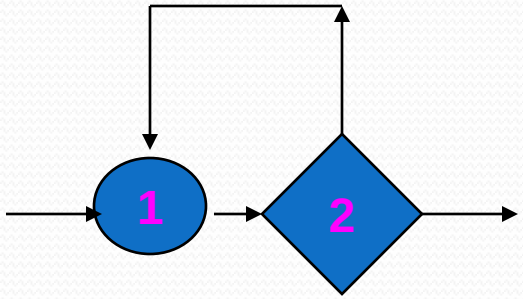
4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.

4. Try n = typical number of iterations : n_{min} < n < n_{max}
5. Try n = n_{max} -1

6. Try n = n_{max}
7. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Note: only a case of no iterations, n = 0 is not there.

Case 3. Single loop with excluded values



1. Treat this as single loops with excluded values as two sets.

2. Example:

V = 1 to 20 excluding 7,8,9 and10

Test cases to attempt are for:

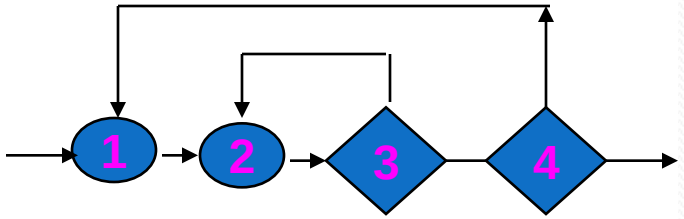
V = 0,1,2,4,6,7 and V = 10,11,15,19,20,21

(underlined cased are not supposed to work)

Control Flow Graphs and Path Testing

Testing of path involving loops…

2. Testing a Nested Loop Statement



- Multiplying # of tests for each nested loop => very large # of tests
- A test selection technique:

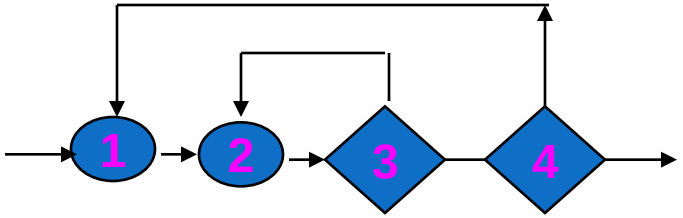
  1. Start at the inner-most loop. Set all outer-loops to Min iteration parameter values: **V_{min}**.
  2. Test the **V_{min}, V_{min} + 1, typical V, V_{max} - 1, V_{max}** for the inner-most loop. Hold the outer-loops to **V_{min}**. Expand tests are required for out-of-range & excluded values.

  3. If you have done with outer most loop, Go To step 5. Else, move out one loop and do step 2 with all other loops set to **typical values**.
  4. Do the five cases for all loops in the nest simultaneously.

- Assignment: check # test cases = 12 for nesting = 2, 16 for 3, 19 for 4.

Control Flow Graphs and Path Testing

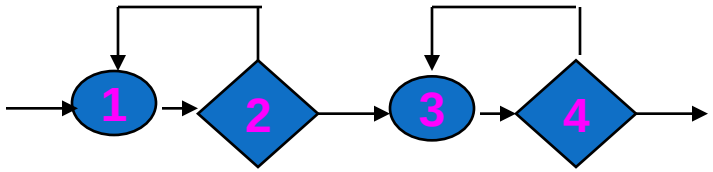
Testing of path involving loops…



- Compromise on # test cases for processing time.
- Expand tests for solving potential problems associated with initialization of variables and with excluded combinations and ranges.
- Apply Huang's twice thorough theorem to catch data initialization problems.

Testing of path involving loops…

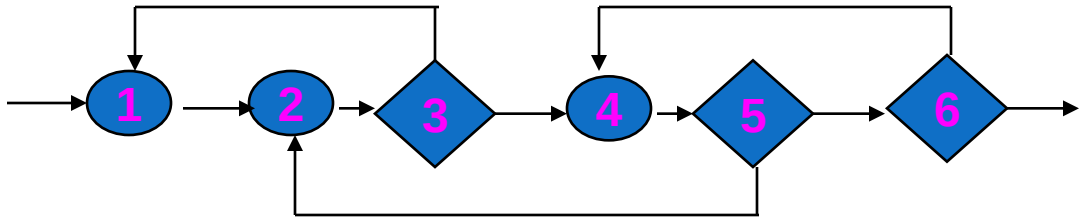
3. Testing Concatenated Loop Statements



- Two loops are concatenated if it's possible to reach one after exiting the other while still on the path from entrance to exit.
- If these are independent of each other, treat them as independent loops.
- If their iteration values are inter-dependent & these are same path, treat these like a nested loop.
- Processing times are additive.

Testing of path involving loops…

4. Testing Horrible Loops



- Avoid these.
- Even after applying some techniques of paths, resulting test cases not definitive.
- Too many test cases.
- Thinking required to check end points etc. is unique for each program.
- Jumps in & out of loops and intersecting loops etc, makes test case selection an ugly task.
- etc. etc.

Testing of path involving loops…

Loop Testing Times

- Longer testing time for all loops if all the extreme cases are to be tested.
- Unreasonably long test execution times indicate bugs in the s/w or specs.

  Case: Testing nested loops with combination of extreme values leads to long test times.

  - Show that it's due to incorrect specs and fix the specs.
  - Prove that combined extreme cases cannot occur in the real world. Cut-off those tests.

  - Put in limits and checks to prevent the combined extreme cases.
  - Test with the extreme-value combinations, but use different numbers.

- The test time problem is solved by rescaling the test limit values.

  - Can be achieved through a separate compile, by patching, by setting parameter values etc..

Effectiveness of Path Testing

- Path testing (with mainly P1 & P2) catches ~65% of Unit Test Bugs ie., ~35% of all bugs.
- More effective for unstructured than structured software.

- **Limitations**
  - Path testing may not do expected coverage if bugs occur.
  - Path testing may not reveal totally wrong or missing functions.
  - Unit-level path testing may not catch interface errors among routines.
  - Data base and data flow errors may not be caught.
  - Unit-level path testing cannot reveal bugs in a routine due to another.
  - Not all initialization errors are caught by path testing.
  - Specification errors cannot be caught.

Effectiveness of Path Testing

- **A lot of work**
  - Creating flow graph, selecting paths for coverage, finding input data values to force these paths, setting up loop cases & combinations.
- Careful, systematic, test design will catch as many bugs as the act of testing.
  Test design process at all levels at least as effective at catching bugs as is running the test designed by that process.
- More complicated path testing techniques than P1 & P2
  - Between P_2 & P_α
    - Complicated & impractical
  - Weaker than P1 or P2.
    - For regression (incremental) testing, it's cost-effective

Control Flow Graphs and Path Testing

Predicates, Predicate Expressions

Path

- A sequence of process links (& nodes)

Predicate

- The logical function evaluated at a decision : True or False. *(Binary , boolean)*

Compound Predicate

- Two or more predicates combined with AND, OR etc.

Path Predicate

- Every path corresponds to a succession of True/False values for the predicates traversed on that path.
- A predicate associated with a path.
  " X > 0 is True " AND "W is either negative or equal to 122" is True
- Multi-valued Logic / Multi-way branching

Predicates, Predicate Expressions…

Predicate Interpretation

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

  An input vector is a set of inputs to a routine arranged as a one dimensional array.

- Example:

```
        INPUT X, Y
        ON X GOTO A, B, C
A:  Z := 7  @  GOTO H
B:  Z := -7  @ GOTO H
C:  Z := 0  @ GOTO H

H:  DO SOMETHING

K:  IF X + Z > 0  GOTO  GOOD  ELSE  GOTO  BETTER
```

```
INPUT X
IF X < 0 THEN Y:= 2
ELSE  Y := 1
IF X + Y*Y > 0 THEN …
```

- Predicate interpretation may or may not depend on the path. IF -7 > 3 ..

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

Predicates, Predicate Expressions…

Process Dependency

- An **input variable** is **independent** of the processing if its value does not change as a result of processing.
- An **input variable** is **process dependent** if its value changes as a result of processing.
- A **predicate** is **process dependent** if its truth value can change as a result of processing.
- A **predicate** is **process independent** if its truth value does not change as a result of processing.
- Process dependence of a predicate doesn't follow from process dependence of variables
- Examples:

X + Y = 10	Increment X & Decrement Y.
X is odd	Add an even # to X

- If all the input variables (on which a predicate is based) are process independent, then **predicate is process independent**.

Predicates, Predicate Expressions…

Correlation

- Two **input variables** are **correlated** if every combination of their values cannot be specified independently.
- **Variables** whose values can be specified independently without restriction are **uncorrelated**.
- A pair of predicates whose outcomes depend on one or more variables in common are **correlated predicates**.
- Every path through a routine is **achievable** only if all predicates in that routine are **uncorrelated**.
- If a routine has a loop, then at least one decision's predicate must be process dependent. Otherwise, there is an input value which the routine loops indefinitely.

Predicates, Predicate Expressions…

Path Predicate Expression

- Every selected path leads to an associated boolean expression, called the **path predicate expression**, which characterizes the input values (if any) that will cause that path to be traversed.
- Select an entry/exit path. Write down un-interpreted predicates for the decisions along the path. If there are iterations, note also the value of loop-control variable for that pass. Converting these into predicates that contain only input variables, we get a set of boolean expressions called path predicate expression.
- Example (inputs being numerical values):

If $X_5 > 0$ .OR. $X_6 < 0$ then

$X_1 + 3X_2 + 17 >= 0$
$X_3 = 17$
$X_4 – X_1 >= 14\ X_2$

Predicates, Predicate Expressions…

A: $X_5 > 0$
B: $X_1 + 3X_2 + 17 >= 0$
C: $X_3 = 17$
D: $X_4 – X_1 >= 14\ X_2$

E: $X_6 < 0$
F: $X_1 + 3X_2 + 17 >= 0$
G: $X_3 = 17$
H: $X_4 – X_1 >= 14\ X_2$

Converting into the predicate expression form:

$$A\ B\ C\ D + E\ B\ C\ D \Rightarrow (A + E)\ B\ C\ D$$

If we take the alternative path for the expression: D then

$$(A + E)\ B\ C\ \overline{D}$$

Predicates, Predicate Expressions…

Predicate Coverage:

- Look at **examples** & possibility of bugs: A B C D A + B + C + D
  - Due to semantics of the evaluation of logic expressions in the languages, the entire expression may not be always evaluated.
  - A bug may not be detected.
  - A wrong path may be taken if there is a bug.
- Realize that on our achieving C2, the program could still hide some control flow bugs.
- **Predicate coverage:**
  - If all possible combinations of truth values corresponding to selected path have been explored under some test, we say predicate coverage has been achieved.
  - Stronger than branch coverage.
  - If all possible combinations of all predicates under all interpretations are covered, we have the equivalent of total path testing.

Testing blindness

- **coming to the right path** – even thru a wrong decision (at a predicate). Due to the interaction of some statements makes a buggy predicate work, and the bug is not detected by the selected input values.

- **calculating wrong number of tests** at a predicate by ignoring the # of paths to arrive at it.

- **Cannot be detected by path testing and need other strategies**

Testing blindness

- **Assignment blinding**: A buggy Predicate seems to work correctly as the specific value chosen in an assignment statement works with both the correct & buggy predicate.

Correct	Buggy	
X := 7	X := 7	
IF Y > 0 THEN …	IF X + Y > 0 THEN …	**(check for Y=1)**

- **Equality blinding**:
  - When the path selected by a prior predicate results in a value that works both for the correct & buggy predicate.

Correct	Buggy	
IF Y = 2 THEN …	IF Y = 2 THEN ..	
IF X + Y > 3 THEN …	IF X > 1 THEN …	**(check for any X>1)**

- **Self-blinding**
  - When a buggy predicate is a multiple of the correct one and the result is indistinguishable along that path.

Correct	Buggy	
X := A	X := A	
IF X - 1 > 0 THEN …	IF X + A -2 > 0 THEN …	**(check for any X,A)**

Achievable Paths

1. Objective is to select & test just enough paths to achieve a satisfactory notion of test completeness such as C1 + C2.

2. Extract the program's control flow graph & select a set of tentative covering paths.

3. For a path in that set, interpret the predicates.

4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression. Example: (A + BC) (D + E)

5. Multiply & obtain **sum-of-products** form of the **path predicate expression**:
   AD + AE + BCD + BCE

6. Each product term denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the selected path.

7. A set of input values for that path is found when any of the inequality sets is solved.

   A solution found => **path is achievable**. Otherwise the path is **unachievable.**

Path Sensitization

It's the act of **finding a set of solutions to the path predicate expression**.

In practice, for a selected path finding the required input vector is not difficult. If there is difficulty, it may be due to some bugs.

Heuristic procedures:

Choose an easily sensitizable path set, & pick hard-to-sensitize paths to achieve more coverage.

Identify all the variables that affect the decisions. For process dependent variables, express the nature of the process dependency as an equation, function, or whatever is convenient and clear. For correlated variables, express the logical, arithmetic, or functional relation defining the correlation.

1. Identify correlated predicates and document the nature of the correlation as for variables. If the same predicate appears at more than one decision, the decisions are obviously correlated.
2. Start path selection with uncorrelated & independent predicates. If coverage is achieved, but the path had dependent predicates, something is wrong.

Path Sensitization… Heuristic procedures: contd..

4. If the coverage is not achieved yet with independent uncorrelated predicates, extend the path set by using correlated predicates; preferably process independent (not needing interpretation)

5. If the coverage is not achieved, extend the path set by using dependent predicates (typically required to cover loops), preferably uncorrelated.

6. Last, use correlated and dependent predicates.

7. For each of the path selected above, list the corresponding input variables. If the variable is independent, list its value. For dependent variables, interpret the predicate ie., list the relation. For correlated variables, state the nature of the correlation to other variables. Determine the mechanism (relation) to express the forbidden combinations of variable values, if any.

8. Each selected path yields a set of inequalities, which must be simultaneously satisfied to force the path.

Examples for Path Sensitization

1. Simple Independent Uncorrelated Predicates

2. Independent Correlated Predicates

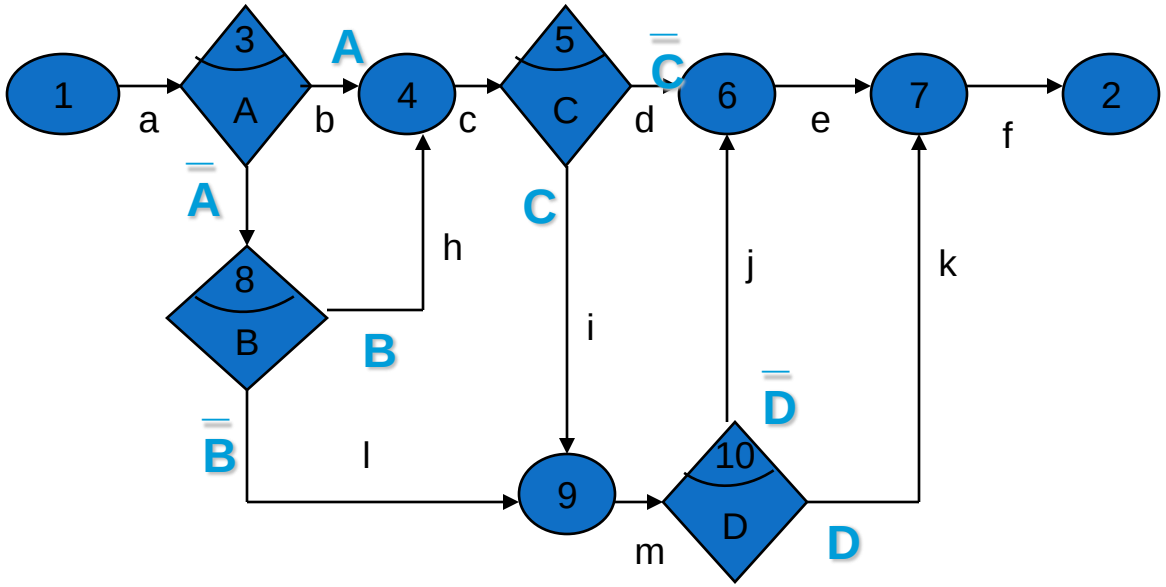
3. Dependent Predicates

4. Generic

Control Flow Graphs and Path Testing

Examples for Path Sensitization..

1. Simple Independent Uncorrelated Predicates



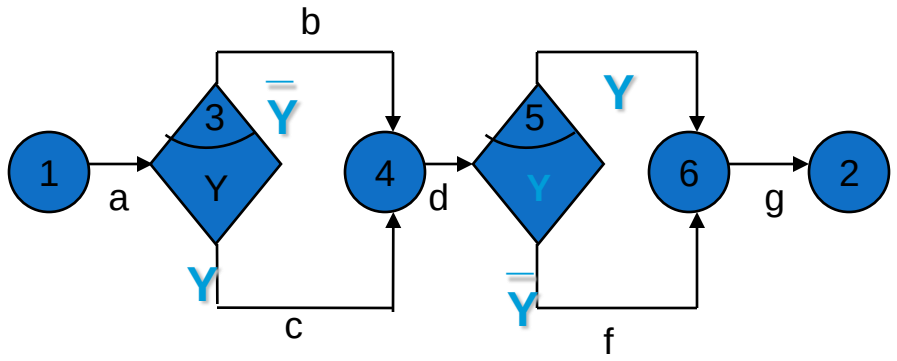
4 predicates => 16 combinations
Set of possible paths = 8

Path	Predicate Values
abcdef	A, $\overline{C}$
aghcimkf	$\overline{A}$, B, C, D
aglmjef	$\overline{A}$, $\overline{B}$, $\overline{D}$

Path	Predicate Values
abcdef	A, $\overline{C}$
abcimjef	A, C, $\overline{D}$
abcimkf	A, C, D
aghcdef	$\overline{A}$, B, $\overline{C}$
aglmkf	$\overline{A}$, $\overline{B}$, $\overline{D}$

A Simple case of solving inequalities. (obtained by the procedure for finding a covering set of paths)

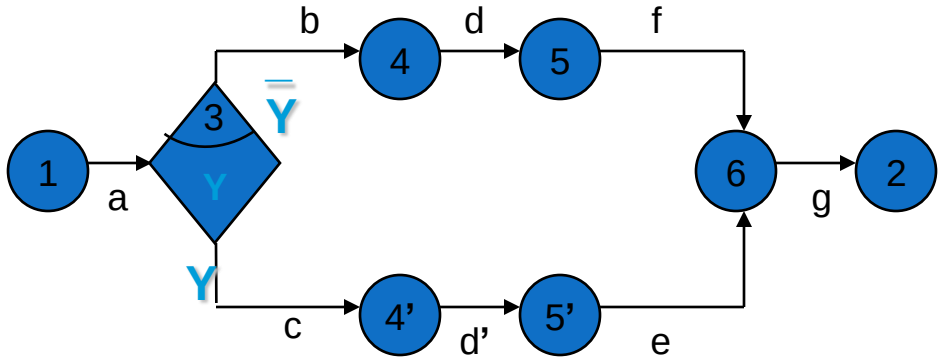
2. Correlated Independent Predicates



Correlated paths => **some** paths are unachievable
ie., redundant paths.
ie., n decisions but # paths are < 2^n

Due to practice of saving code which makes the code very difficult to maintain.

Eliminate the correlated decisions.
Reproduce common code.



If a chosen sensible path is not achievable,

- there's a bug.
- design can be simplified.
- get better understanding of correlated decisions

Correlated decision removed & CFG simplified

3. Dependent Predicates

Usually most of the processing does not affect the control flow.

Use computer simulation for sensitization in a simplified way.

Dependent predicates contain iterative loop statements usually.

For Loop statements:

Determine the value of loop control variable for a certain # of iterations, and then work backward to determine the value of input variables (input vector).

4. The General Case

No simple procedure to solve for values of input vector for a selected path.

1. Select cases to provide coverage on the basis of **functionally sensible paths**.

Well structured routines allow easy sensitization.

Intractable paths may have a bug.

2. Tackle the path with the fewest decisions first. Select paths with least # of loops.

3. Start at the end of the path and list the predicates while tracing the path in **reverse**.

Each predicate imposes restrictions on the subsequent (in reverse order) predicate.

4. Continue tracing along the path. Pick the broadest range of values for variables affected and consistent with values that were so far determined.

5. Continue until the entrance & therefore have established a set of input conditions for the path.

If the solution is not found, path is not achievable, *it could be a bug.*

4. The General Case contd..

Alternately:

1. In the **forward** direction, list the decisions to be traversed.
   For each decision list the broadest range of input values.

2. Pick a path & adjust all input values. These restricted values are used for next decision.

3. Continue. Some decisions may be dependent on and/or correlated with earlier ones.

4. The path is unachievable if the input values become contradictory, or, impossible.

   If the path is achieved, try a new path for additional coverage.

Advantages & Disadvantages of the two approaches:

The forward method is usually less work.

you do not know where you are going as you are tracing the graph.

PATH INSTRUMENTATION

Output of a test:

Results observed. But, there may not be any expected output for a test.

Outcome:

Any change or the lack of change at the output.

Expected Outcome:

Any **expected** change or the lack of change at the output (predicted as part of design).

Actual Outcome:

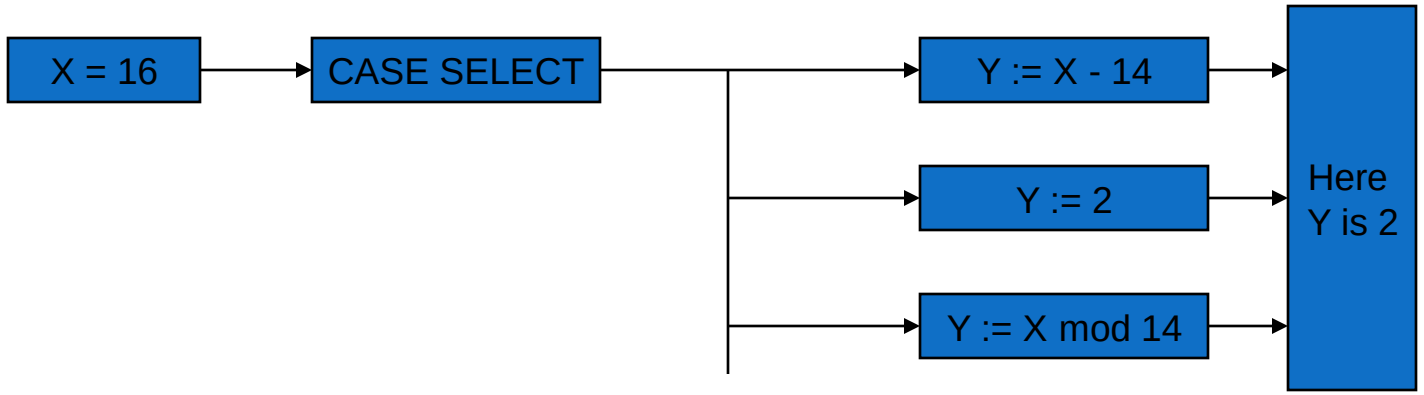
Observed outcome

PATH INSTRUMENTATION

Coincidental Correctness:

When expected & actual outcomes match,

- Necessary conditions for test to pass are met.
- Conditions met are probably not sufficient.
  (the expected outcome may be achieved due to a wrong reason)



Path Instrumentation is what we have to do confirm that the outcome was achieved by the intended path.

PATH INSTRUMENTATION METHODS

1. General strategy:

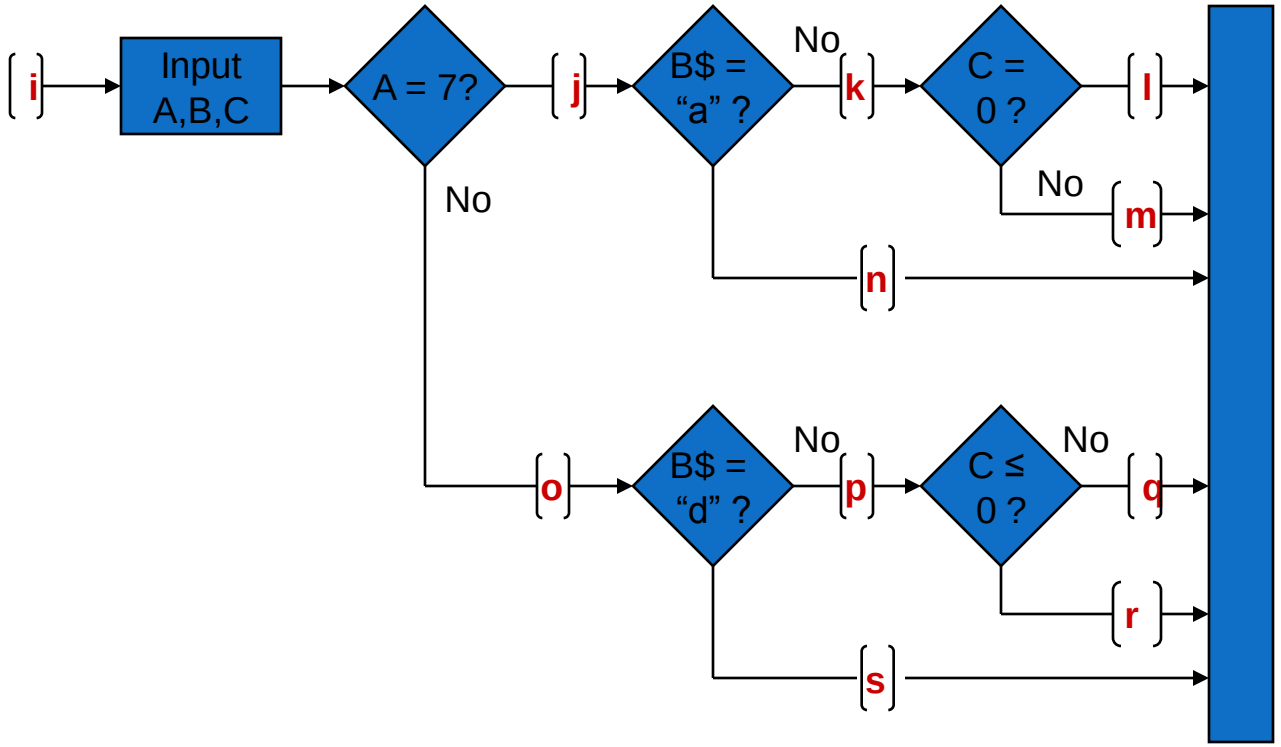
1. Based on Interpretive tracing & use interpreting trace program.
2. A trace confirms the expected outcome is or isn't obtained along the intended path.
3. Computer trace may be too massive. Hand tracing may be simpler.

2. Traversal or Link Makers:

Simple and effective

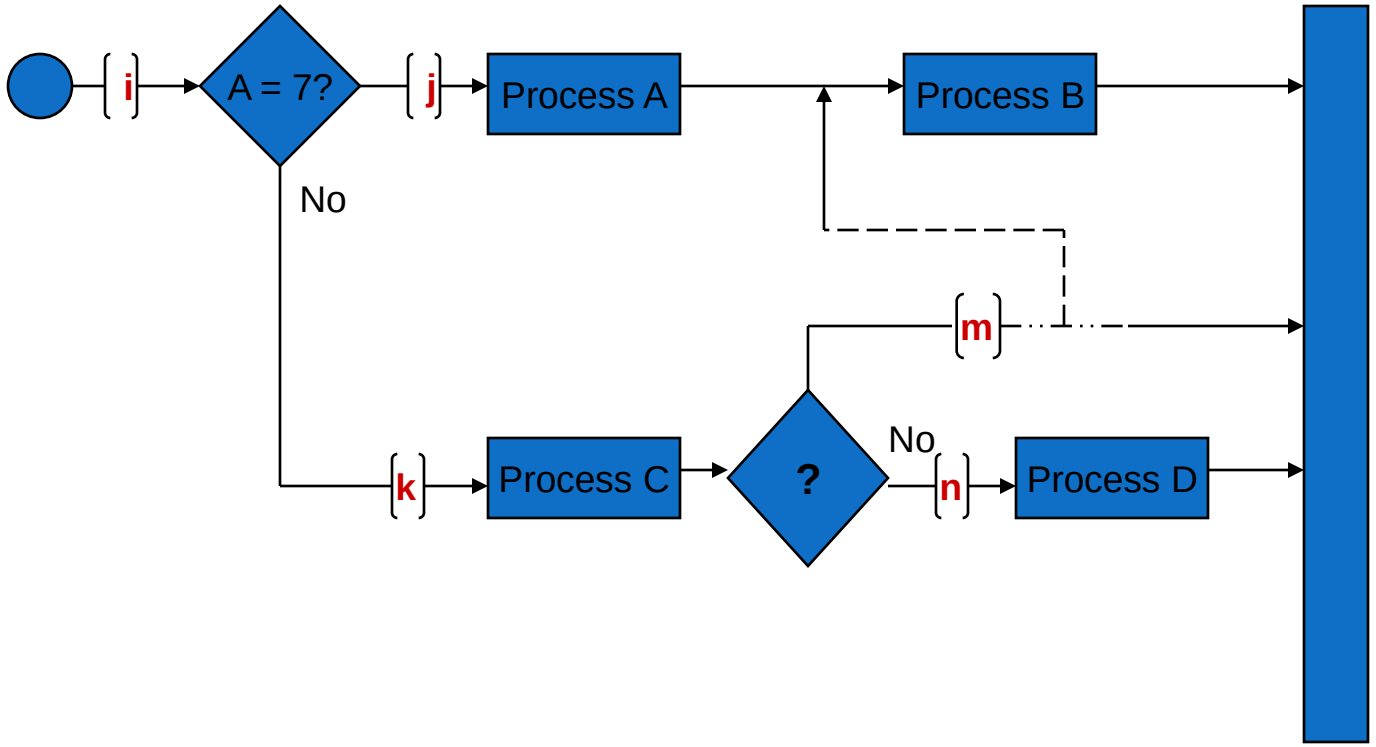
1. Name every link.
2. Instrument the links so that the link is recorded when it is executed (during the test)
3. The succession of letters from a routine's entry to exit corresponds to the pathname.

Single Link Marker Instrumentation: An example



Sample path: i j n

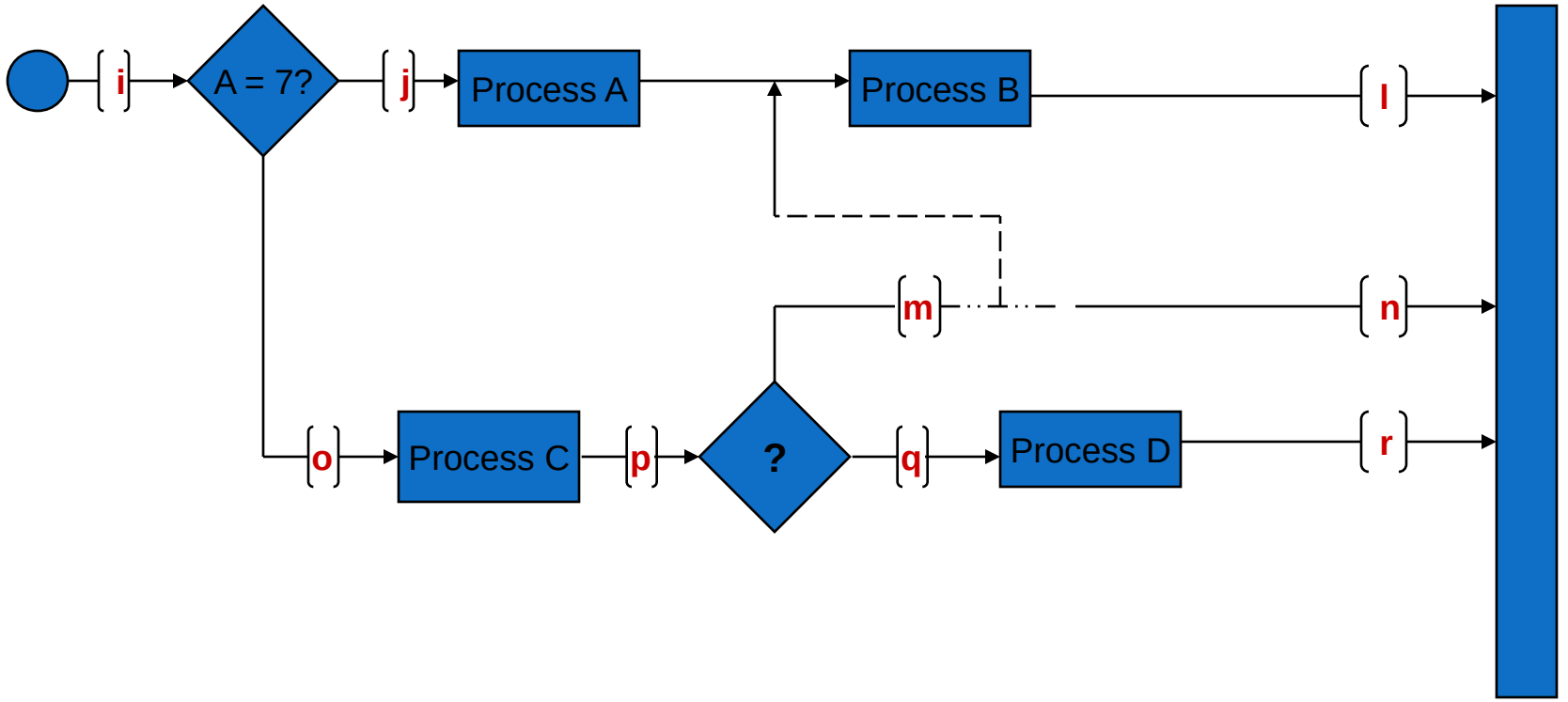
Single Link Marker Instrumentation: Not good enough



Problem:

Processing in the links may be chewed open by bugs. Possibly due to GOTO statements, control takes a different path, yet resulting in the intended path again.

Double Link Marker Instrumentation:



The problem is solved.

Two link markers specify the path name and both the beginning & end of the link.

3. Link Counters Technique:

- Less disruptive and less informative.
- Increment a link counter each time a link is traversed. **Path length could confirm the intended path**.
- For avoiding the same problem as with markers, use double link counters.
  Expect an even count = double the length.
- Now, put a link counter on every link. *(earlier it was only between decisions)*
  If there are no loops, the link counts are = 1.
- Sum the link counts over a series of tests, say, a covering set. Confirm the total link counts with the expected.
- Using double link counters avoids the same & earlier mentioned problem.

3. Link Counters Technique: contd..

Check list for the procedure:

- Do begin-link counter values equal the end-link counter values?
- Does the input-link count of every decision equal to the sum of the link counts of the output links from that decision?
- Do the sum of the input-link counts for a junction equal the output-link count for that junction?
- Do the total counts match the values you predicted when you designed the covering test set?

This procedure and the checklist could solve the problem of Instrumentation.

PATH INTRUMENTATION techniques...

Limitations

- Instrumentation probe (marker, counter) **may disturb the timing relations** & hide racing condition bugs.

- Instrumentation probe (marker, counter) **may not detect location dependent bugs**.

  If the presence or absence of probes modifies things (for example in the data base) in a faulty way, then the **probes hide the bug** in the program.

PATH INTRUMENTATION - IMPLEMENTATION

For Unit testing : Implementation may be provided by a comprehensive test tool.

For higher level testing or for testing an unsupported language:

- Introduction of probes could introduce bugs.
- Instrumentation is more important for higher level of program structure like transaction flow
- At higher levels, the discrepancies in the structure are more possible & overhead of instrumentation may be less.

For Languages supporting conditional assembly or **compilation**:

- Probes are written in the source code & tagged into categories.
- Counters & traversal markers can be implemented.
- Can selectively activate the desired probes.

For language not supporting conditional assembly / compilation:

- Use macros or function calls for each category of probes. This may have less bugs.
- A general purpose routine may be written.

In general:

- Plan instrumentation with probes in levels of increasing detail.

Implementation & Application of Path Testing

1. Integration, Coverage, and Paths in Called Components

- Mainly used in Unit testing, especially new software.

- **In an Idealistic bottom-up integration test process** – integrating one component at a time. Use stubs for lower level component (sub-routines), test interfaces and then replace stubs by real subroutines.

- **In reality,** integration proceeds in associated blocks of components. Stubs may be avoided. Need to think about paths inside the subroutine.

  To achieve C1 or C2 coverage:
  - Predicate interpretation may require us to treat a subroutine as an in-line-code.
  - Sensitization becomes more difficult.
  - Selected path may be unachievable as the called components' processing may block it.

Weaknesses of Path testing:

- It assumes that effective testing can be done one level at a time without bothering what happens at lower levels.
- predicate coverage problems & blinding.

Implementation & Application of Path Testing

2. Application of path testing to **New Code**

- Do Path Tests for C1 + C2 coverage
- Use the procedure similar to the idealistic bottom-up integration testing, using a mechanized test suite.
- A path blocked or not achievable could mean a bug.
- When a bug occurs the path may be blocked.

Implementation & Application of Path Testing

3. Application of path testing to **Maintenance**

- Path testing is applied first to the modified component.
- Use the procedure similar to the idealistic bottom-up integration testing, but without using stubs.
- Select paths to achieve C2 over the changed code.
- Newer and more effective strategies could emerge to provide coverage in maintenance phase.

Implementation & Application of Path Testing

4. Application of path testing to **Rehosting**

- Path testing with C1 + C2 coverage is a powerful tool for rehosting old software.
- Software is rehosted as it's no more cost effective to support the application environment.
- Use path testing in conjunction with automatic or semiautomatic structural test generators.

Implementation & Application of Path Testing

Application of path testing to **Rehosting..**

Process of path testing during rehosting

- A translator from the old to the new environment is created & tested. Rehosting process is to catch bugs in the translator software.
- A complete C1 + C2 coverage path test suite is created for the old software. Tests are run in the old environment. The outcomes become the specifications for the rehosted software.
- Another translator may be needed to adapt the tests & outcomes to the new environment.
- The cost of the process is high, but it avoids risks associated with rewriting the code.
- Once it runs on new environment, it can be optimized or enhanced for new functionalities (which were not possible in the old environment.)

UNIT-II

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

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

Definitions of Transaction,Transaction-flow, TFG

Transaction-flow

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

Transaction-flow Graph

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

Transaction

- is a unit of work seen from system's user point of view.
- consists of a sequence of operations performed by a system, persons or external devices.
- is created (birth) due to an external act & up on its completion (closure), it remains in the form of historical records.

A Simple Transaction

Example: the sequence of steps in a transaction in an online information retrieval system

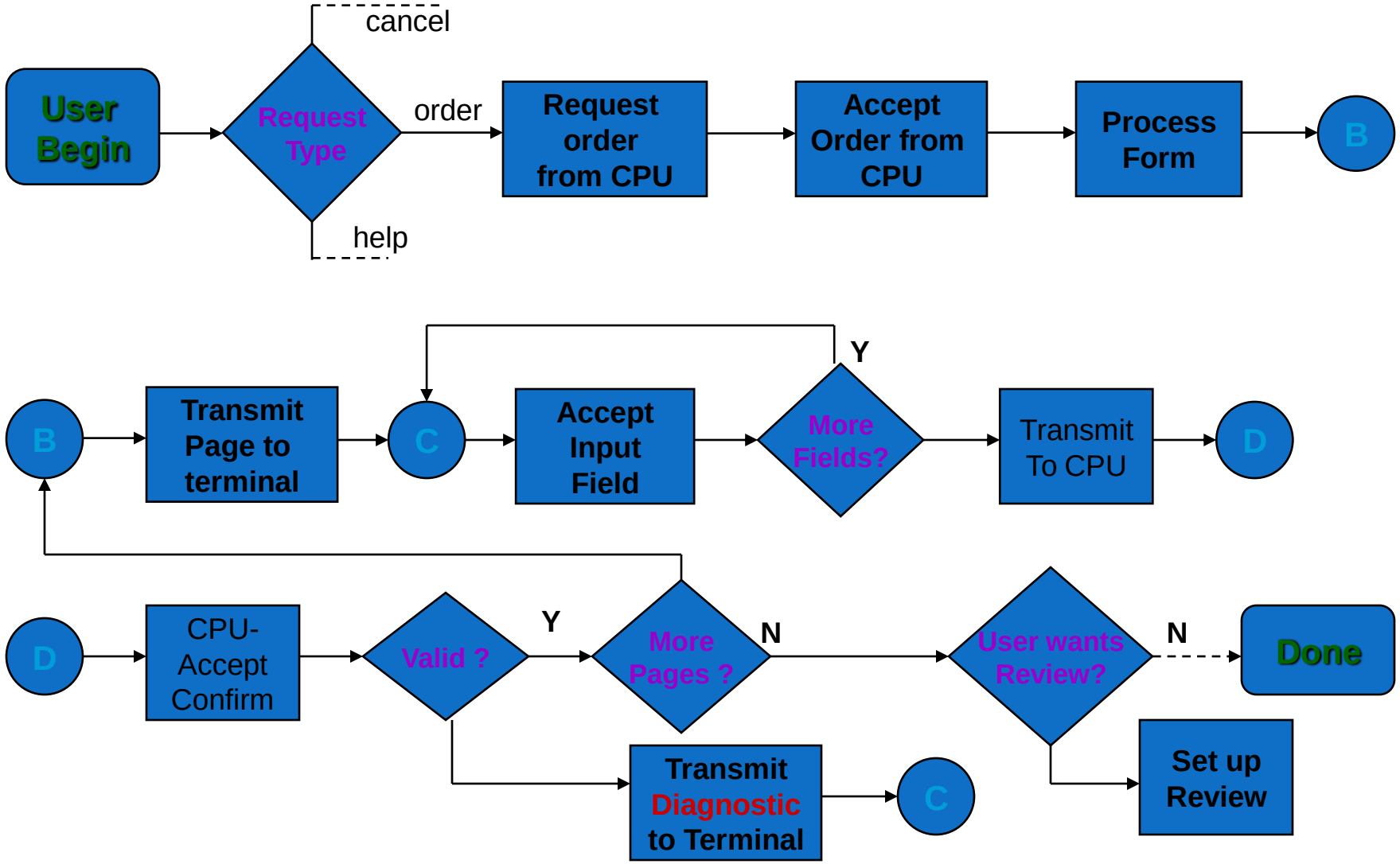
1. accept inputs
2. Validate inputs (Birth of tr.)
3. Transmit ack. to the user
4. Process input
5. Search file
6. Request direction from user
7. Accept Inputs
8. Validate inputs
9. Process the request
10. Update file
11. Transmit output
12. Record transaction in log & cleanup (Closure)

Users View of a transaction : Single step

Systems view : Sequence of many operations

Example of a Transaction flow (diagram)

User (terminal) ⟷ Terminal controller ⟷ CPU



Definitions

Transaction-flow Graph : a scenario between users & computer

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

Uses of Transaction-flow

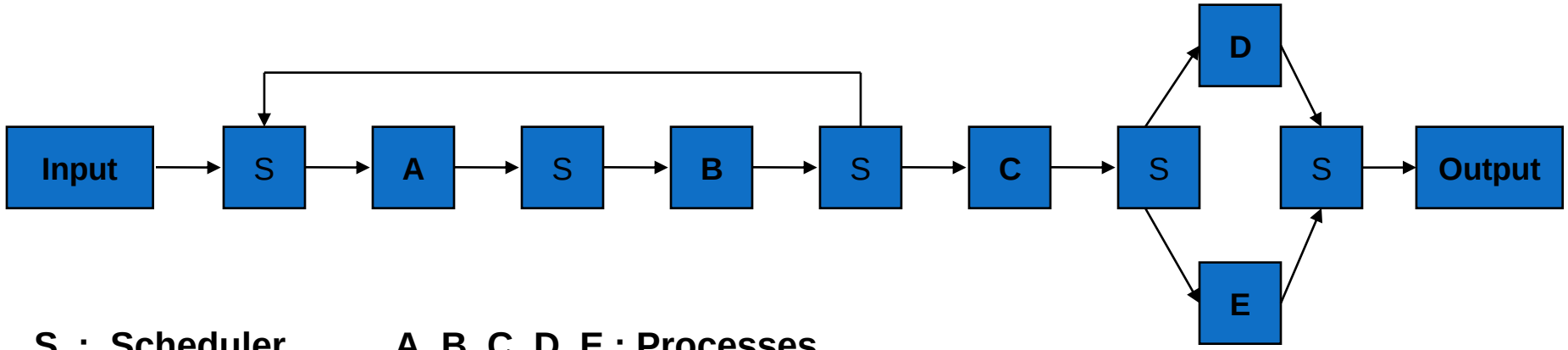
Specifying requirements of big, online and complicated systems.

Airline reservation systems, air-traffic control systems.

Loops are less as compared to CFG. Loops are used for user input error processing

Implementation of Transaction-Flow (in a system)

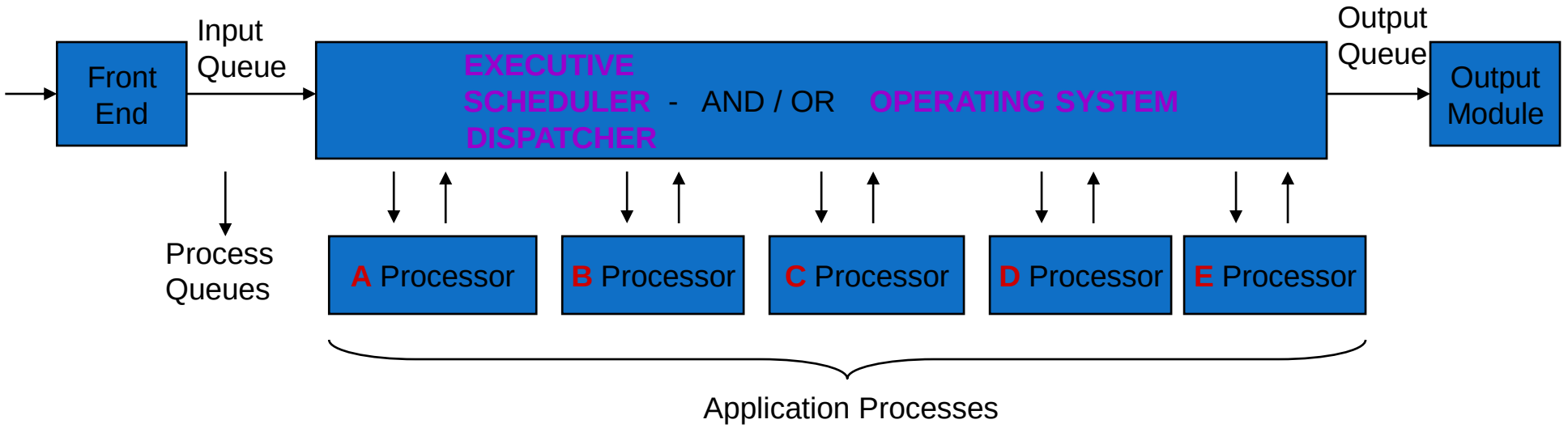
- Implicit in the design of system's control structure & associated database.
- No direct one-to-one correspondence between the "**processes**" and "**decisions**" of transaction-flow, and the corresponding program **component**.
- A **transaction-flow** is a path taken by the transaction through a succession of processing modules.
- A transaction is represented by a **token**.
- A **transaction-flow graph** is a pictorial representation of what happens to the tokens.



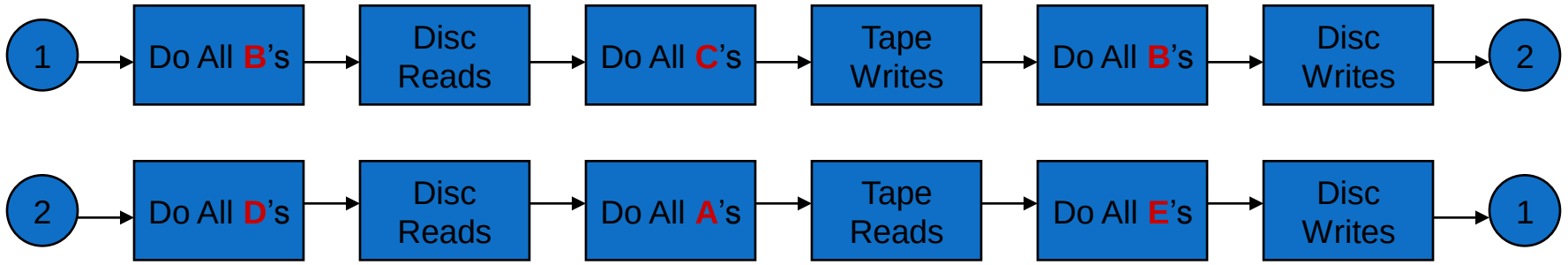
S : Scheduler **A, B, C, D, E : Processes**

Implementation of Transaction-Flow

System Control Structure (architecture of the implementation) :



Executive / Dispatcher Flowchart *(a sample sequence)*



System control structure

System is controlled by a scheduler …

A Transaction is created by filling in a **Transaction Control Block** (TCB) by user inputs and by placing that **token** on input Q of Scheduler.

Scheduler examines and places it on appropriate process Q such as **A**. When **A** finishes with the Token, it places the TCB back on the scheduler Q.

Scheduler routes it to the next process after examining the token :

1. It contains tables or code to route a token to the next process.
2. It may contain routing information only in tables.
3. Scheduler contains no code / data. Processing modules contain code for routing.

Implementation of Transaction-Flow

Transaction Processing System (simplified):

- There are many Tr. & Tr-flows in the system.
- Scheduler invokes processes A to E as well as disk & tape read & writes.
- The order of execution depends on priority & other reasons.

Cyclic structure like in this example is common in process control & communication systems.

♦ The criteria for implementation mechanism depends on performance and resource optimization.

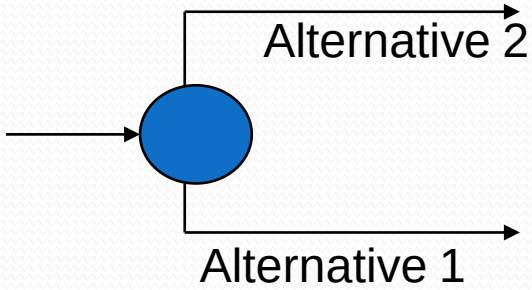
A perspective of Transaction-Flow

Transaction-flow testing is a block box technique. *(as we assumed nothing regarding computer, communications, environment, O.S., transaction identity or structure or state.)*

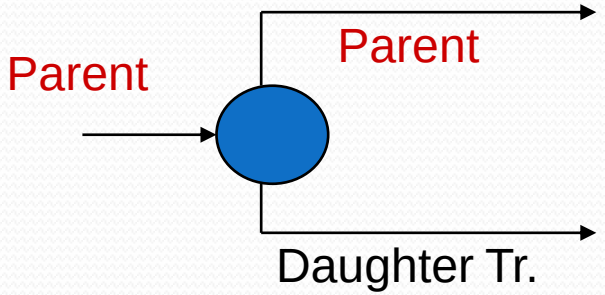
1. TFG is a kind of DFG.
   - TFG has tokens, & DFG has data objects with history of operations applied on them.
   - Many techniques of CFG apply to TFG & DFG
2. **Decision nodes** of TFG have exception exits to the central recovery process.
3. So, we ignore the **effect of interrupts** in a Transaction-flow.

Splits of transactions (Births)

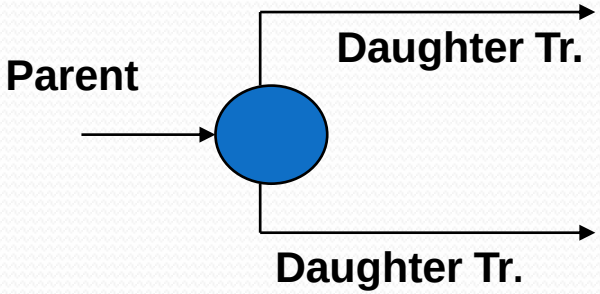
1. A decision point in TFG



2. Biosis

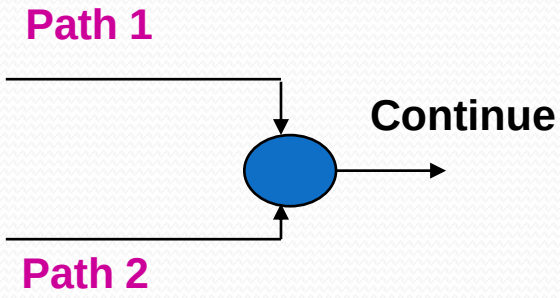


3. Mitosis

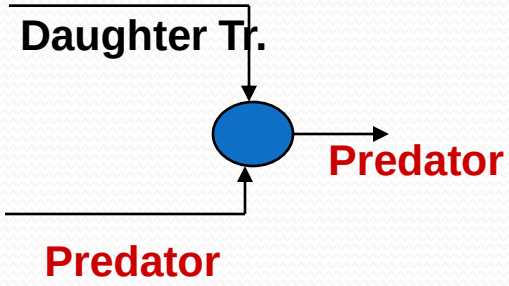


Mergers of transactions

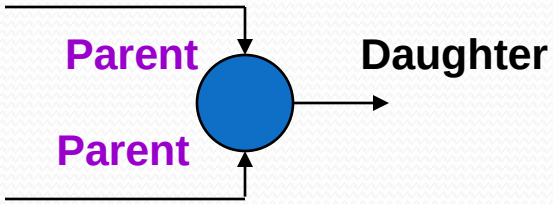
1. Junction



2. Absorption



3. Conjugation



TFG – splitting, merging Transactions

NOTES:

- Multiple meanings now for decision and junction symbols in a TFG.
- Simple TFG model is **not enough** to represent multi-processor systems & associated coordination systems.
- **Petrinet model** uses operations for all the above. But Petrinets are applied to H/W, N/W protocol testing – but not **Software**.

Simplify TFG model

- **Add New Tr-Flows for** Biosis, Mitosis, Absorption, Conjugation
- **Problems for programmer, designer & test designer.**
- Need to **design specific tests** – possibility of bugs.

Transaction-flow Structure

Reasons for Unstructuredness

1. Processes involve Human Users

1. Part of Flow from External Systems

1. Errors, Failures, Malfunctions & Recovery Actions

1. Transaction Count, Complexity. Customer & Environment

Transaction-flow Structure

Reasons for Unstructuredness …

5. **New Transactions, & Modifications**

6. **Approximation to Reality**

   - Attempt to Structure

Transaction - Flow Testing - Steps

First, Build / Obtain Transaction Flows

- **Represent Explicitly**
- **Design details the Main Tr-Flows**
- **Create From PDL**
- **HIPO charts & Petrinet Representations**

Objective – Trace the transaction

Transaction - Flow Testing - Steps

1. Inspections, Reviews & Walkthroughs

Start From Preliminary Design

1. Conducting Walkthroughs

- **Discuss enough Transaction Types (**98% Transactions**)**
- **User needs & Functional terms** *(Design independent)*
- **Traceability to Requirements**

Transaction - Flow Testing - Steps

1. **Inspections, Reviews & Walkthroughs …**

2. **Design Tests for C1 + C2 coverage**

3. **Additional Coverage (> C1+C2)**

   - Paths with loops, extreme values, domain boundaries
   - **Weird cases, long & potentially troublesome Tr.**

4. **Design Test cases for Tr. Splits & mergers**

5. **Publish Selected Test Paths early**

6. **Buyer's Acceptance – functional & acceptance tests**

Transaction - Flow Testing Techniques

2. Path Selection

1. **Covering Set (C1+C2) of Functionally Sensible Tr.**

2. **Add Difficult Paths**

   - Review with designers & implementers
   - Exposure of interface problems & duplicated processing
   - Very few Implementation bugs may remain

Transaction-flow Path Covering Set belongs in System Feature Tests

Transaction - Flow Testing Techniques

3. Sensitization

1. **Functionally Sensible Paths – Simple**
2. **Error, Exception, External Protocol Interface Paths - Difficult**

Testing Tr.–Flows with External Interfaces

- Use patches & break points, mistune, and break the rules,

4. Instrumentation

1. **Link Counters are not Useful.**
2. **Need**
   - Trace
   - Queues on which Tokens resided
   - Entries to & Exits from Dispatcher
   - A Running Log

Make Instrumentation as part of System Design

Transaction - Flow Testing Techniques

5. Test Data bases

1. **Design & Maintenance of a Test Data base - Effort**
2. Mistakes
   - Unawareness about design of a centrally administered test DB
   - Test DB design by Testers
   - Using one DB for all tests (need 4 to 5)

Need experienced System & Test Designers

Transaction - Flow Testing Techniques

6. Test Execution

1. **Use Test Execution Automation**

2. Have to do a large # of Tests for C1+C2 coverage

Transaction - Flow Testing - Implementation

1. **Transaction based systems**
   - **TCB**

1. **Centralized, Common Processing Queues**
   - **Just O(n) Queues for Links of $O(n^2)$**

2. **Transaction Dispatcher**
   - **Uses tables & Finite State Machines**

3. **Recovery & Other Logs**
   - **Key events in Tr – Flow**

4. **Self-Test Support**
   - **Privileged modes in Transaction control tables**

Anomaly

Unreasonable processing on data

- Use of data object before it is defined
- Defined data object is not used

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

- DFT Leads to testing strategies between P_{∞} and P1 / P2

Data - Flow Testing - Basics

Definition:

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

Example:

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

Motivation

- **Confidence in the program**
- **Data dominated design.** *Code migrates to data..*
- **Source Code for Data Declarations**

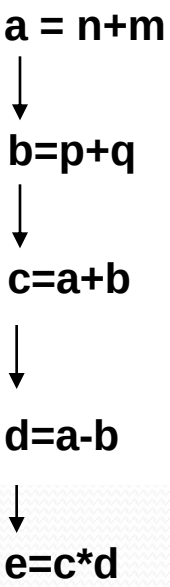
- **Data flow Machines vs Von Neumann's**
  - **Abstract M I M D**
  - **Language & compiler take care of parallel computations**

Program Control flow with Von Neumann's paradigm

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

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

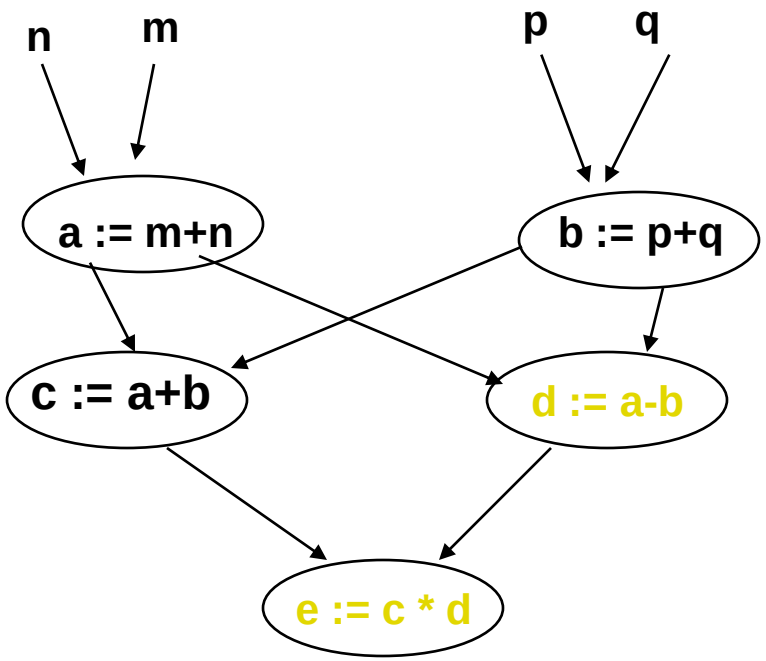
```
a := m + n
b := p + q
c := a + b
d := a - b

e := c * d
```



Multiple representations of control flow graphs possible.

Program Flow using Data Flow Machines paradigm

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



The interrelations among the data items remain same.

Data - Flow Testing - Basics - Motivation

- **Control flow graph**
  - **Multiple representations**

- **Data Flow Graph**

  A spec. for relations among the data objects.

  Covering DFG => Explore all relations under some test.

Assumptions

- **Problems in a control flow**
- **Problems with data objects**

Data Flow Graphs (DFG)

- It is a graph with nodes & directed links
- Test the Von Neumann way -

  Convert to a CFG

  Annotate : program actions (weights)

Data Object State & Usage

Program Actions (d, k, u):

Defined (created)	-	explicitly or implicitly	(d)
Killed (released)	-	directly or indirectly	(k)
Used	-	(u)	
• In a calculation	-	(c)	
• In a predicate	-	directly or indirectly	(p)

Data Flow Anomalies

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

dd : harmless, suspicious

dk : probably a bug.

du : normal

kd : normal

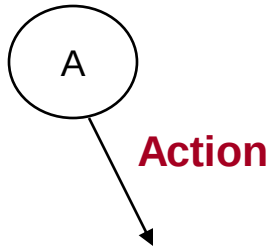
kk : harmless, but probably a bug

ku : a bug

ud : normal. Redefinition.

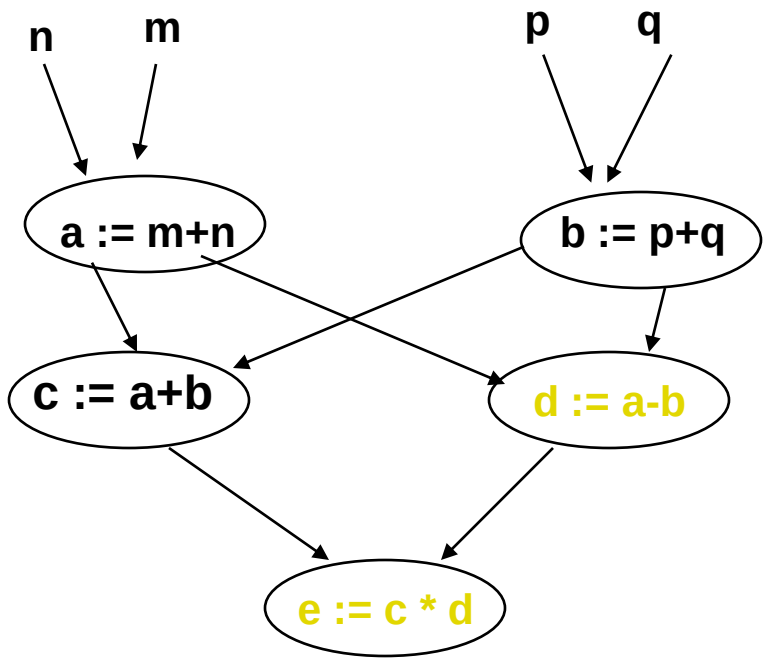
uk : normal

uu : normal



Program Flow using Data Flow Machines paradigm

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



The interrelations among the data items remain same.

Actions on data objects

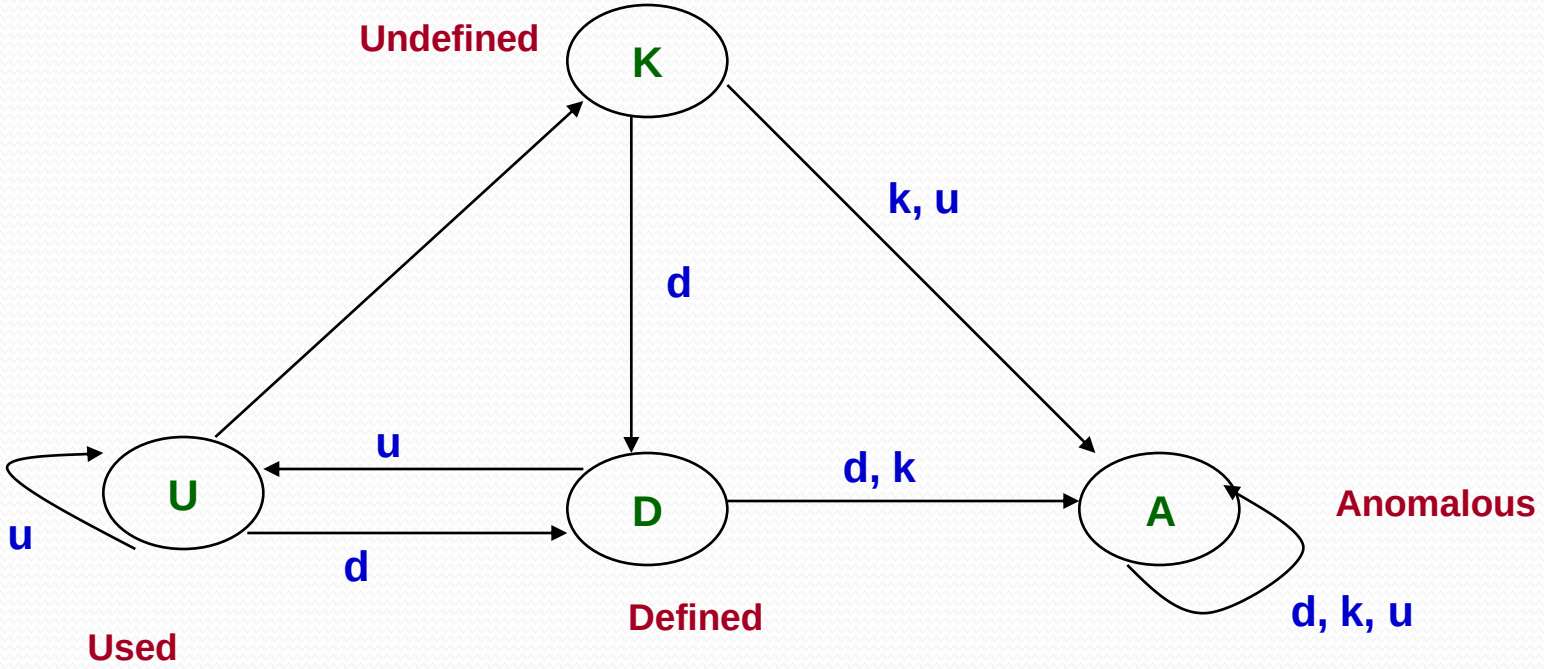
-	no action from START to this point From this point till the EXIT
- d	normal
- u	anomaly
- k	anomaly
k-	normal
u -	normal - possibly an anomaly
d -	possibly anomalous

Data Flow Anomaly State graph

- **Data Object State**
  - **K, D, U, A**
- **Processing Step**
  - **k, d, u**

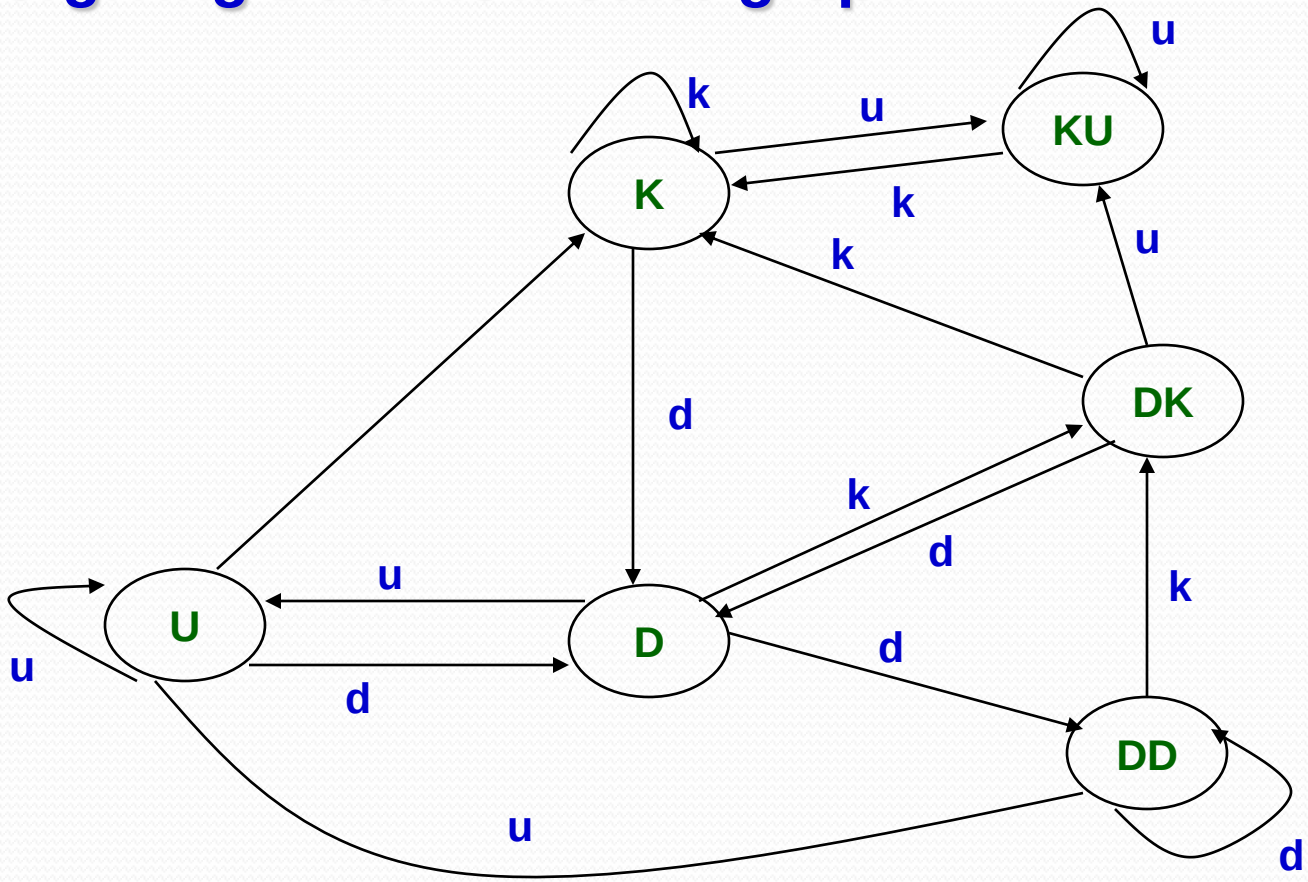
Data Flow Anomaly State graph

- **Object state**
- **Unforgiving Data flow state graph**



Data Flow Anomaly State graph

Forgiving Data flow state graph



A ⇒ DD, DK, KU

Data Flow State Graphs

- Differ in processing of anomalies
- Choice depends on Application, language, context

Static vs Dynamic Anomaly Detection

- Static analysis of data flows
- Dynamic analysis
  Intermediate data values

Insufficiency of Static Analysis (for Data flow)

1. Validation of Dead Variables
2. Validation of pointers in Arrays
3. Validation of pointers for Records & pointers
1. Dynamic addresses for dynamic subroutine calls
2. Identifying False anomaly on an unachievable path
1. Recoverable anomalies & Alternate state graph
2. Concurrency, Interrupts, System Issues

Data Flow Model

- Based on CFG
- CFG annotated with program actions
- link weights : dk, dp, du etc..
- Not same as DFG
- For each variable and data object

Procedure to Build:

1. Entry & Exit nodes

1. Unique node identification

1. Weights on out link

2. Predicated nodes

3. Sequence of links

    1. Join

    2. Concatenate weights

    3. The converse

Data - Flow Testing - Basics : Data Flow Model

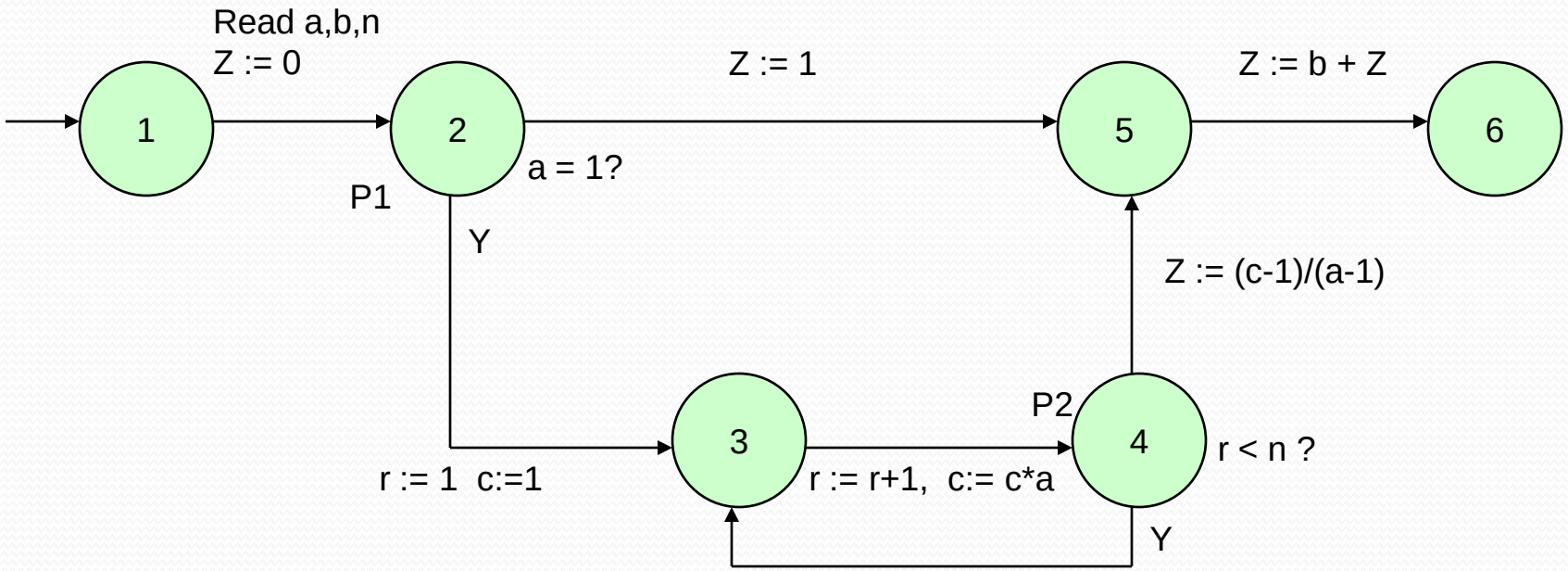
Example:

$$Z = b + \frac{a^n - 1}{a - 1}$$

```
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     GO TO POWER
    Z := (c – 1) / (a – 1)
DONE1:
Z := b + Z

END
```

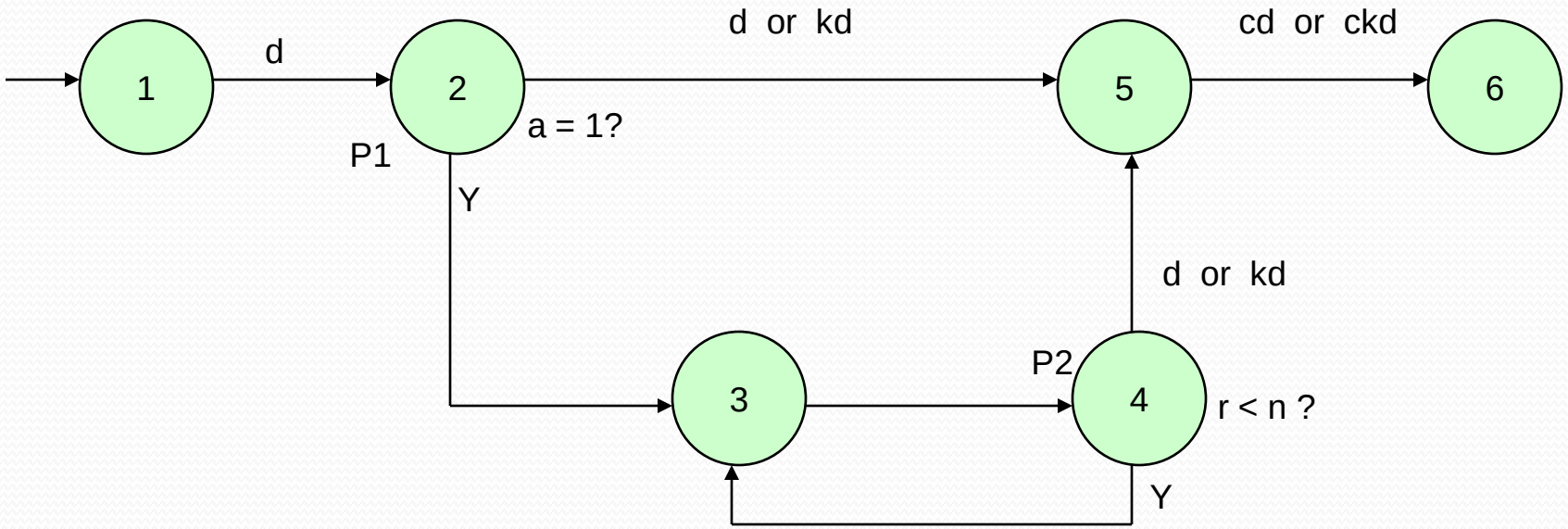
Data - Flow Testing - Basics – Data Flow model

CFG for the Example



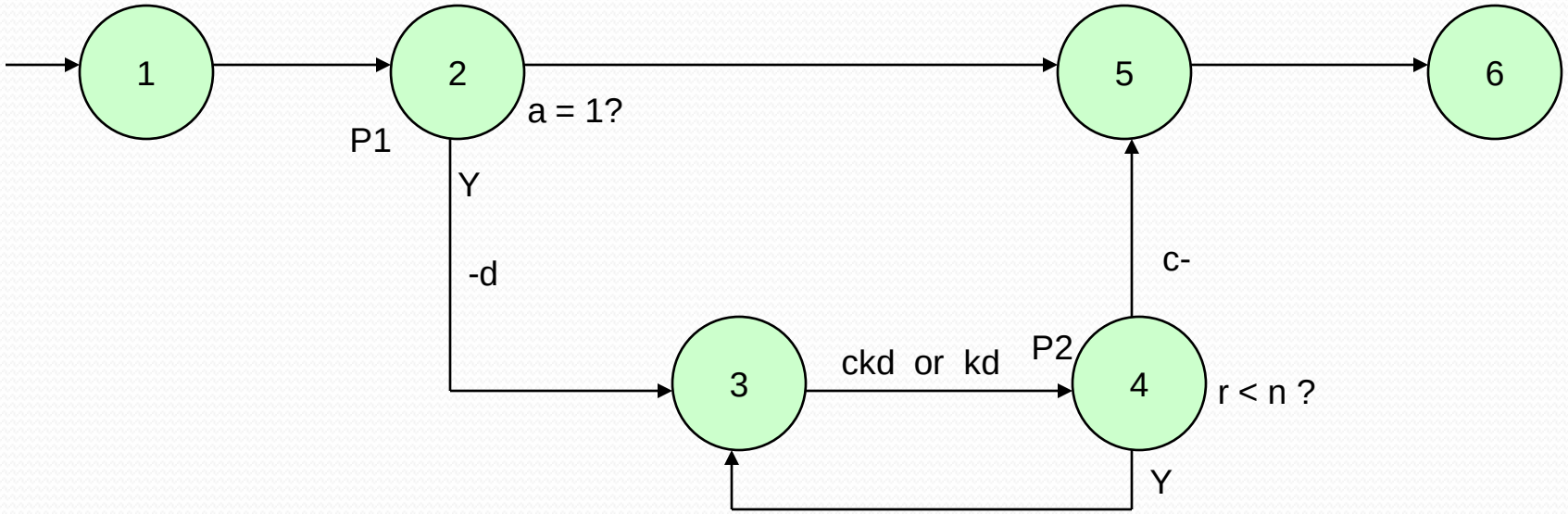
Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for Z



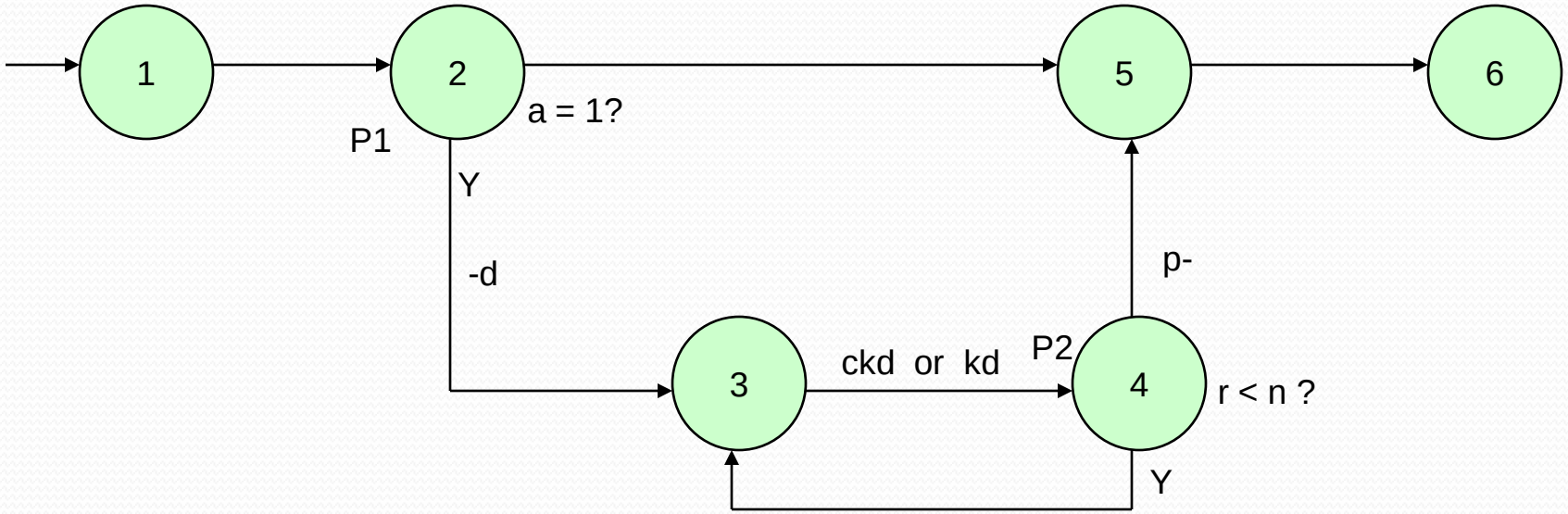
Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for c



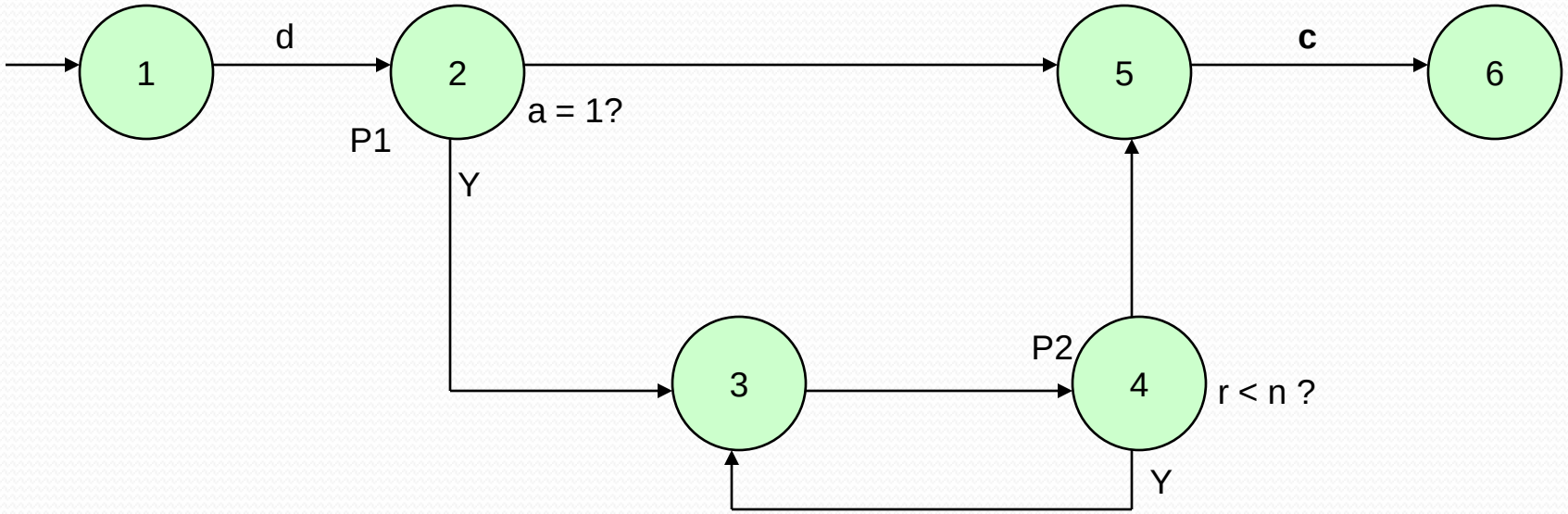
Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for r



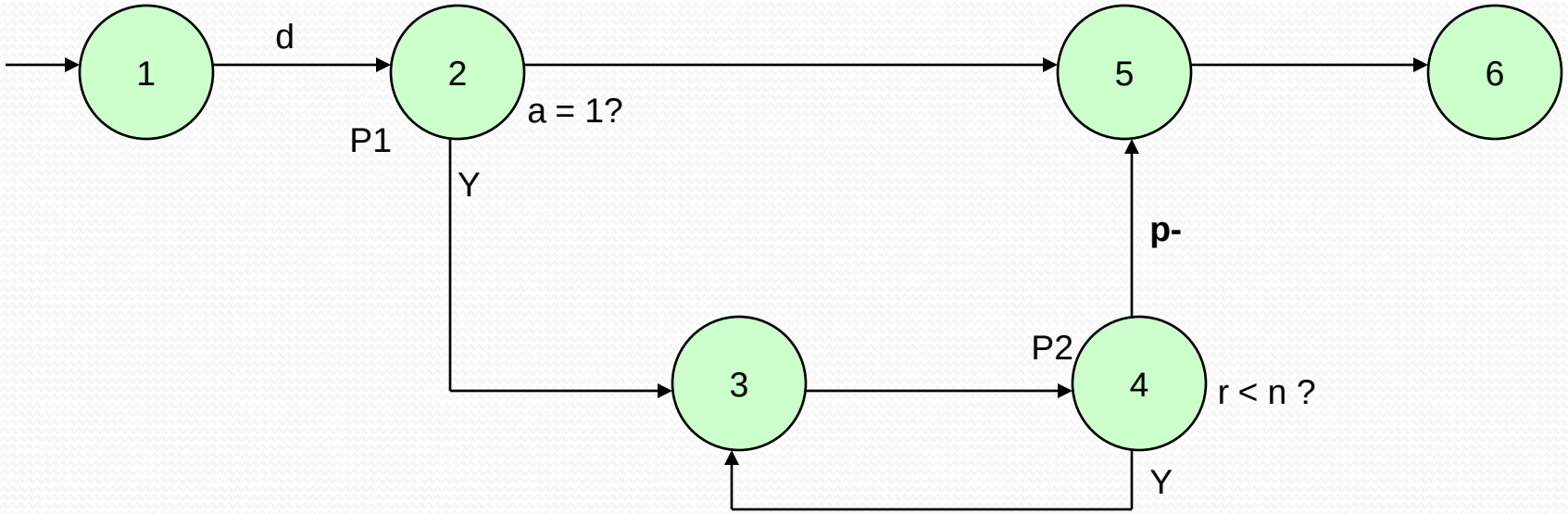
Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for **b**



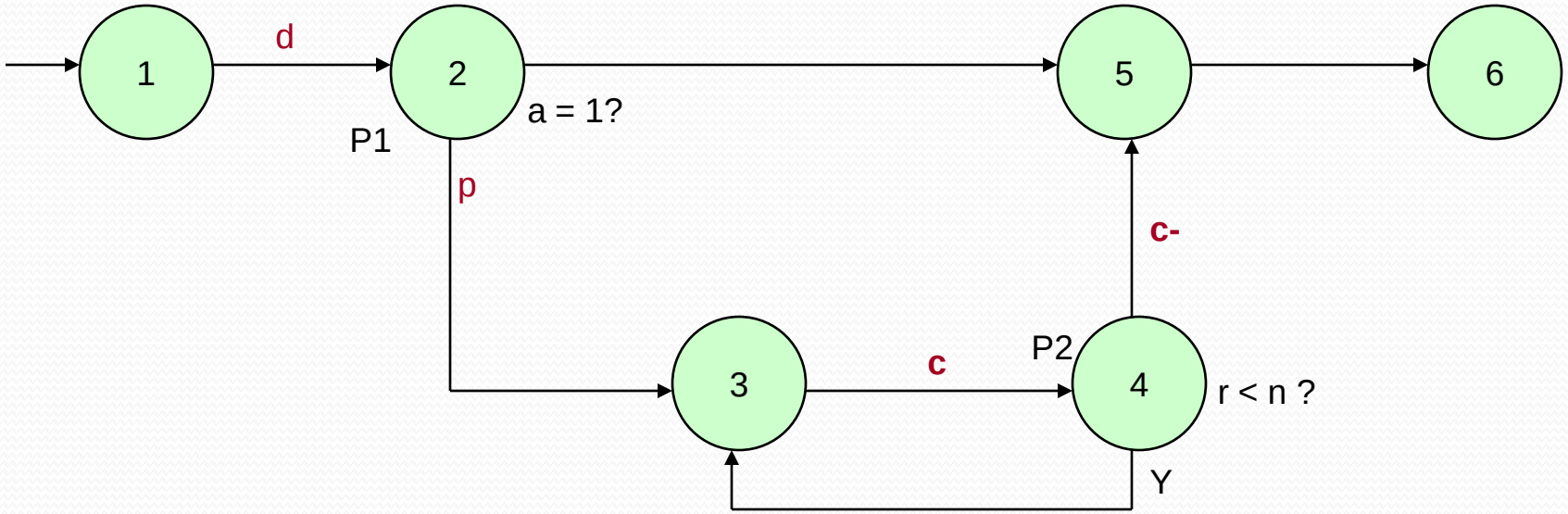
Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for **n**



Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for **a**



Data - Flow Testing - Basics – Data Flow model

A DFM for each variable

Single DFM for multiple variables

Use weights subscripted with variables

Data - Flow Testing – Data Flow Testing Strategies

- A structural testing strategy (**path testing**)
- Add, data flow strategies **with link weights**
- Test path segments to have a **‘d’** (or **u, k, du, dk**)

Data - Flow Testing – Data Flow Testing Strategies

DEFINITIONS

- **w.r.t. a variable or data object ‘v’**
- **Assume all DF paths are achievable**

1. **Definition-clear path segment**
   no k, kd

2. **Loop-free path segment**

2. **Simple path segment**

3. **du path from node i to k**

   - **definition-clear & simple c**
   - **definition-clear & loop-free p**

DFT Strategies

1. All-du paths (ADUP)
2. All uses (AU) strategy
3. All p-uses/some c-uses and All c-uses/some p-uses
1. All Definitions Strategy
1. All p-uses, All c-uses Strategy

Purpose:

Test Design, Develop Test Cases

Data - Flow Testing – Data Flow Testing Strategies

1. All-du paths (ADUP)

- Strongest DFT
- Every du path for every variable for every definition to every use

2. All uses (AU) strategy

- At least one definition clear path segment from every definition of every variable to every use of that definition be exercised under some test.
- At least one path segment from every definition to every use that can be reached from that definition.

3. All p-uses/some c-uses and All c-uses/some p-uses

- APU + c
  - Stronger than P2
- ACU + p
  - Weaker than P2

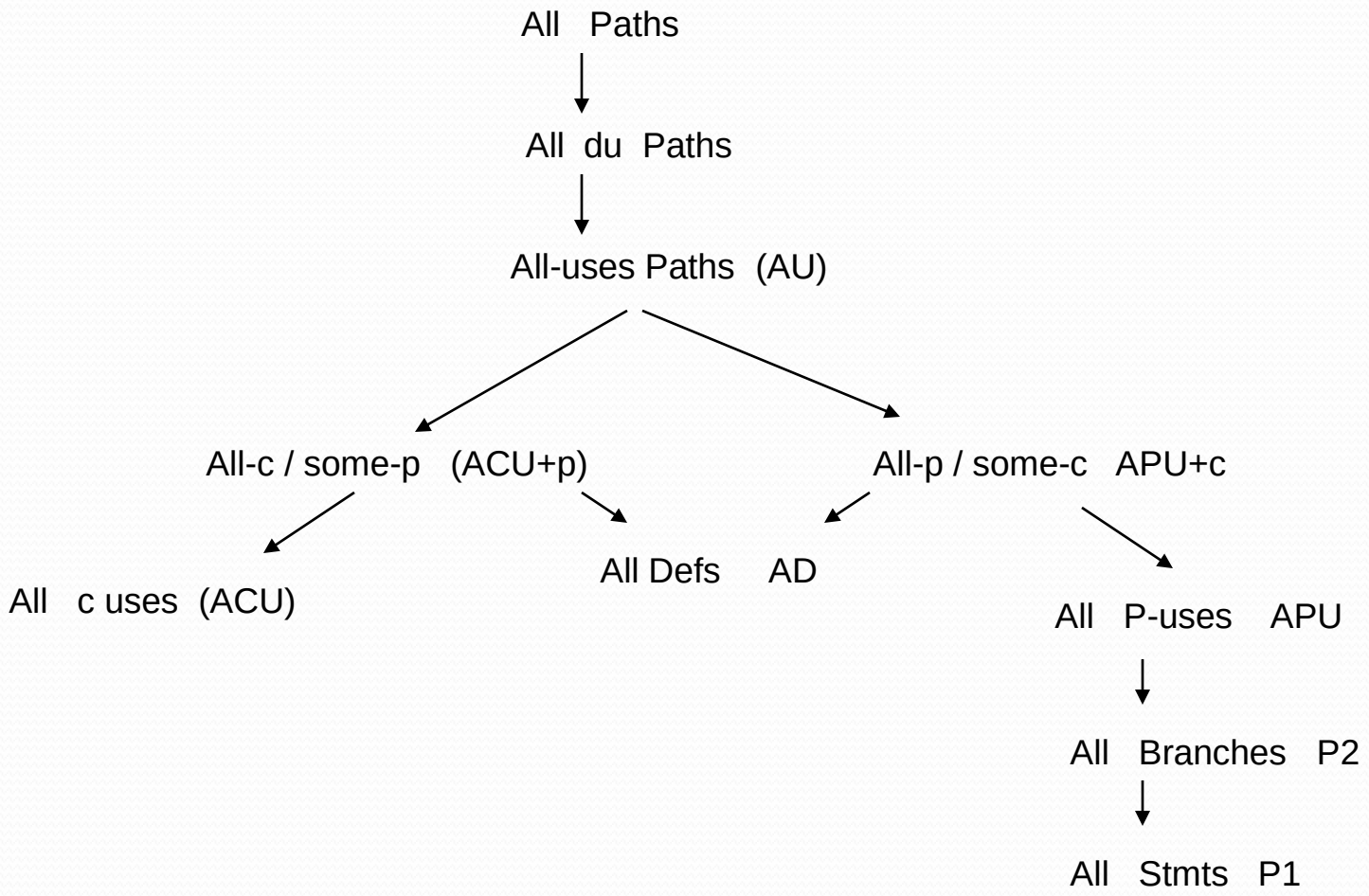
4. All Definitions Strategy (AD)

- Cover every definition by **at least one** p or c
- Weaker than ACU + p and APU + c

5. All-Predicate Uses, All-Computational Uses Strategy

- **APU :**
  - **Include definition-free path for every definition of every variable from the definition to predicate use.**
- **ACU :**
  - **Include for every definition of every variable include at least one definition-free path from the definition to every computational use.**

Ordering the strategies



Testing, Maintenance & Debugging in the Data Flow context

Slicing:

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

Stmt1 var v
stmt2
Stmt3 var v
Stmt4 var v

Stmt s var v

Testing, Maintenance & Debugging in the Data Flow context

Dicing:

- A program dice is a part of slice in which all stmts. which are known to be correct have been removed.
- Obtained from ‘slice’ by incorporating correctness information from testing / debugging.

Testing, Maintenance & Debugging in the Data Flow context

Debugging:

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

Testing, Maintenance & Debugging in the Data Flow context

Dynamic Slicing:

- Refinement of static slicing
- Only achievable paths to the stmt ‘s’ in question are included.

Slicing methods bring together testing, maintenance & debugging.

Application of DFT

- **Comparison Random Testing, P2, AU - by Ntafos**
  - **AU detects more bugs than**
    - **P2 with more test cases**
    - **RT with less # of test cases**
- **Comparison of P2, AU - by Sneed**
  - **AU detects more bugs with 90% Data Coverage Requirement.**

Application of DFT

- **Comparison of # test cases for ACU, APU, AU & ADUP**
  - **by Weyuker using ASSET testing system**
  - **Test Cases Normalized. t = a + b * d** *d = # binary decisions*
  - **At most d+1 Test Cases for P2** *loop-free*
  - **# Test Cases / Decision**

    ADUP > AU > APU > ACU > revised-APU

Application of DFT

Comparison of # test cases for ACU, APU, AU & ADUP by Shimeall & Levenson

Test Cases Normalized. t = a + b * d *(d = # binary decisions)*

At most d+1 Test Cases for P2 *loop-free*

 # Test Cases / Decision

ADUP ~ ½ APU*

AP ~ AC

Application of DFT

DFT vs P1, P2

- **DFT is Effective**
- **Effort for Covering Path Set ~ Same**
- **DFT Tracks the Coverage of Variables**
- **Test Design is similar**

Data - Flow Testing - – Data Flow Testing Strategies

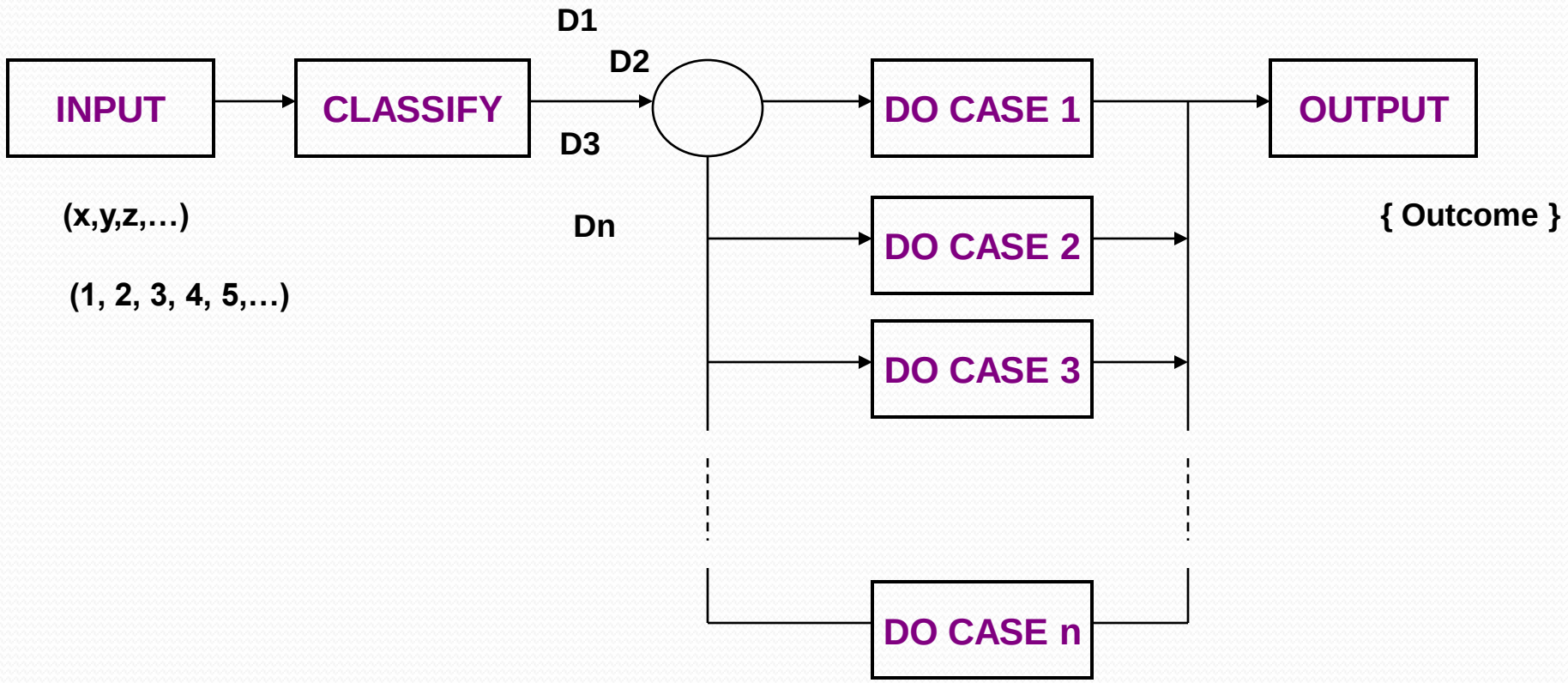
DFT - TOOLS

- **Cost-effective development**
- **Commercial tools :**
- **Can possibly do Better than Commercial Tools**
  - **Easier Integration into a Compiler**
  - **Efficient Testing**

UNIT-III

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

Domain Testing Model



Domain Testing

Views Programs as input data classifiers

Verifies if the classification is correct

Domain Testing Model

Two Views

- Based on Specs
  - Functional Testing
- Based on Implementation information
  - Structural Technique

Domain Testing Model

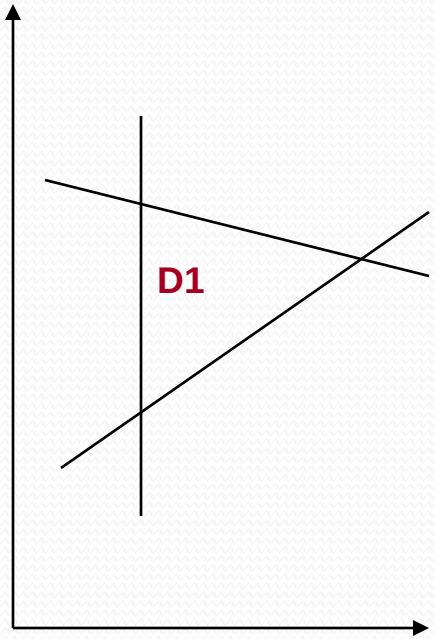
- Variables
  - Simple combinations of two variables
  - Numbers from $-\infty$ to $+\infty$
- Input variables as numbers
- Loop-free Programs

Domain Testing Model

- Structural Knowledge is not needed. Only Specs.
- For each Input Case,
  - A Hypothetical path for Functional testing
  - An Actual path for Structural testing

Domain Testing - Domains & Paths

Domain (simplified)



- **A Single Connected Set of numbers**
- **No arbitrary discrete sets**
- **Defined by the boundaries**
  - One or more boundaries
  - Specified by predicates
  - Bugs likely at the boundaries
- One or more variables

Predicates

- Interpretation
  - Structural Testing - CFG
  - Functional Testing - DFG
- Specifies the Domain Boundary
- n Predicates in Sequence $\Rightarrow$ 2^n domains
  - Or, just two domains

A .AND. B .AND. C

Paths

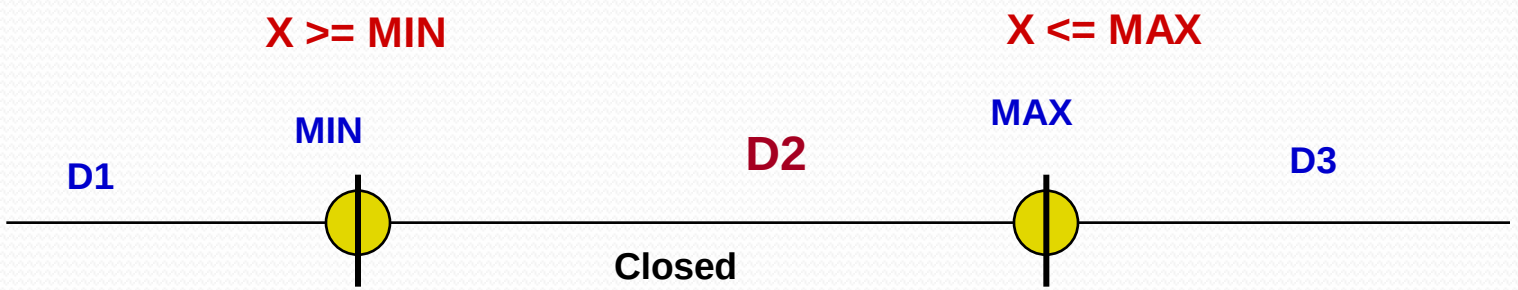
- At least One PATH through the Program
- More paths for disconnected domains

Summary:

- Domain for a loop-free program corresponds to a set of numbers defined over the input vector
- For every domain, there is at least one path thru the routine, along which that domain's processing is done
- The set of interpreted predicates traversed on that path (ie., the path's predicate expression) defines the domain's boundaries.

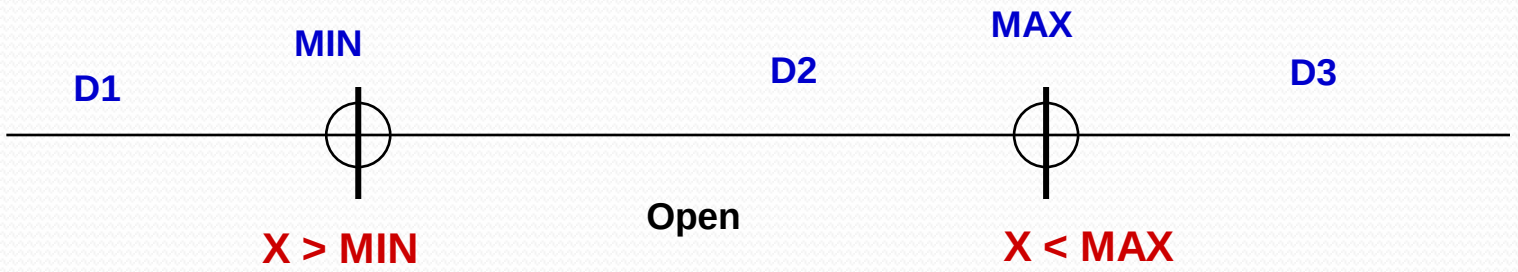
Domain Testing - Domains & Paths - Domain Closure

Boundary: Closed / Open



Domain Testing - Domains & Paths - Domain Closure

Boundary: Closed / Open



Domain Testing - Domains & Paths - Domain Dimensionality

- **One dimension per variable**
- **At least one predicate**
  - Slices thru previously defined Domain
- **Slicing Boundary**
- **N-spaces are cut by Hyperplanes**

Domain Testing - Domains & Paths – Bug Assumptions

- **Processing is OK. Domain definition may be wrong.**

  ⇒ Boundaries are wrong.

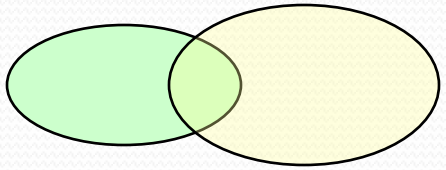
  ⇒ Predicates are wrong.

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

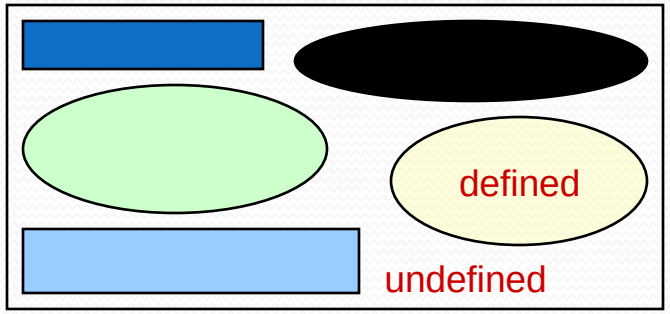
- **More bugs causing domain errors …**

Domain Testing - Domains & Paths – Bug Assumptions..

- **Double-zero representation**
- **Floating point zero check**
- **Contradictory domains**



- **Ambiguous domains**



Domain Testing - Domains & Paths – Bug Assumptions..

- **Over-specified domains**

  X > 3 .AND. X < 2 .AND. Y > 3

- **Boundary errors**

  Boundary closure

  Shifted

Domain Testing - Domains & Paths - Bug Assumptions

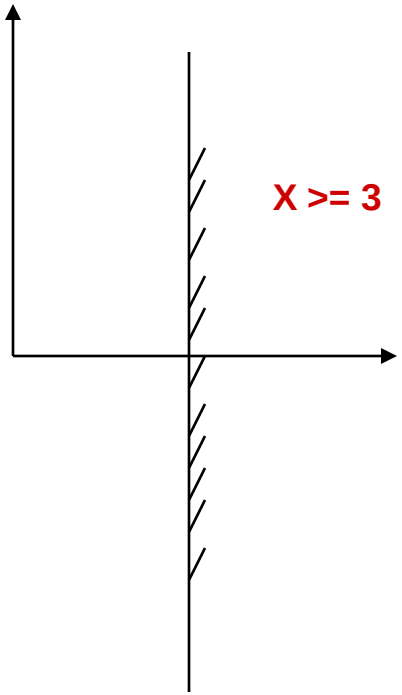
- **Boundary errors….**

Tilted

Missing

Extra boundary

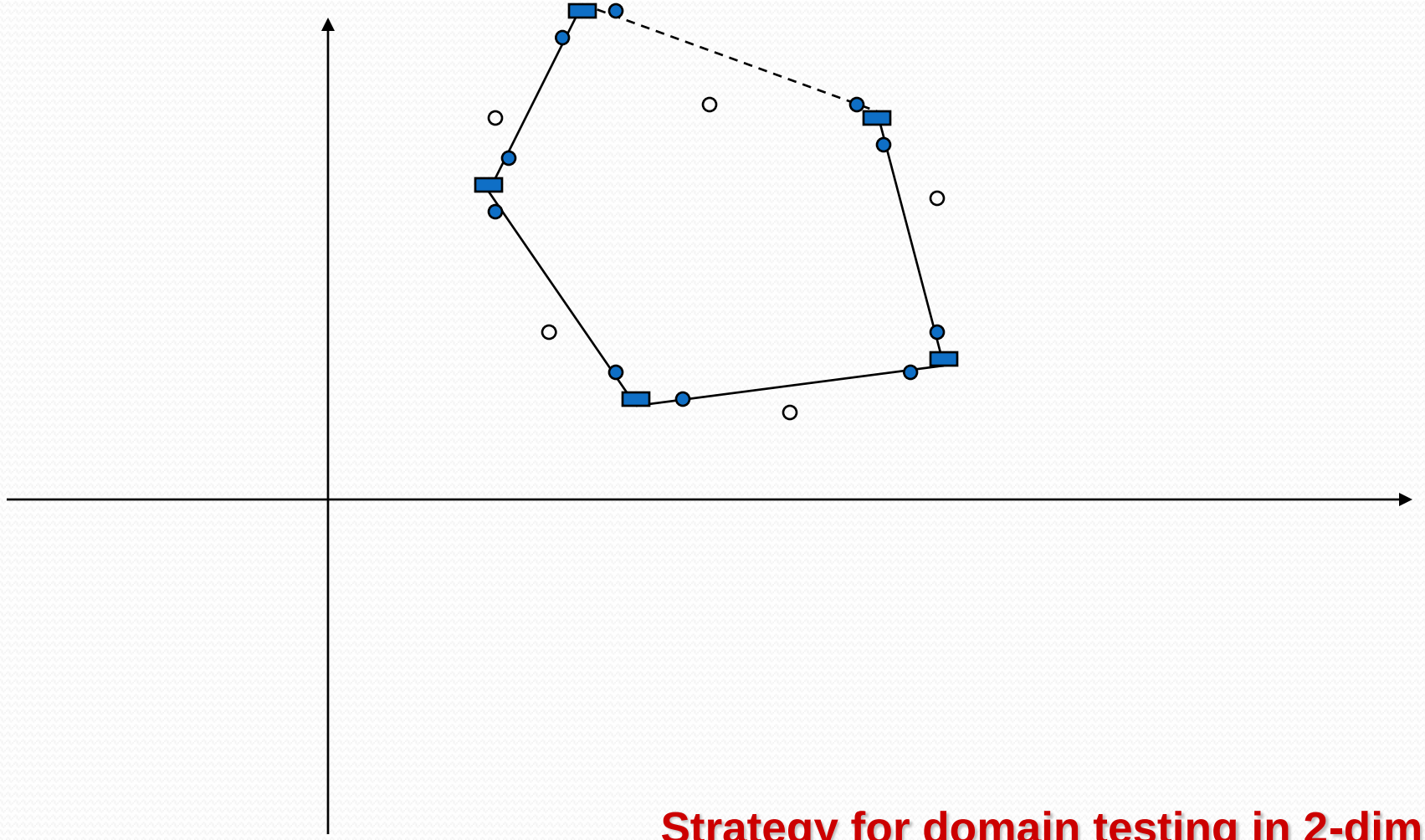
- **Closure Reversal**



- **Faulty Logic**

Simplification of compound predicates

Domain Testing - Domains & Paths - Bug Assumptions



Strategy for domain testing in 2-dim

In General…

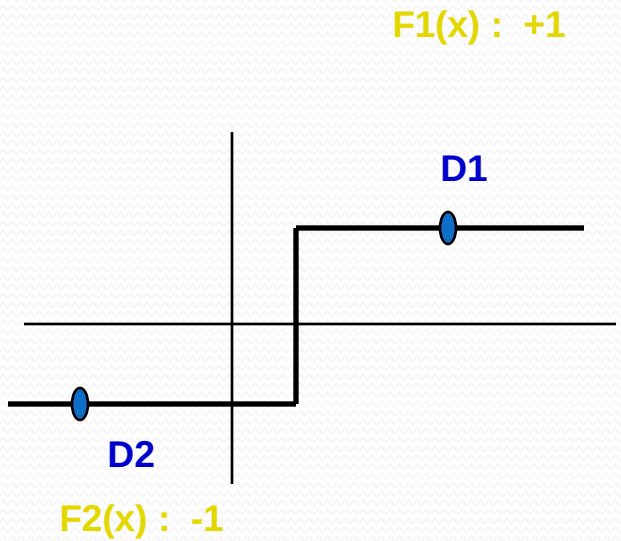
1. Coincidental Correctness

DT cannot detect
Example

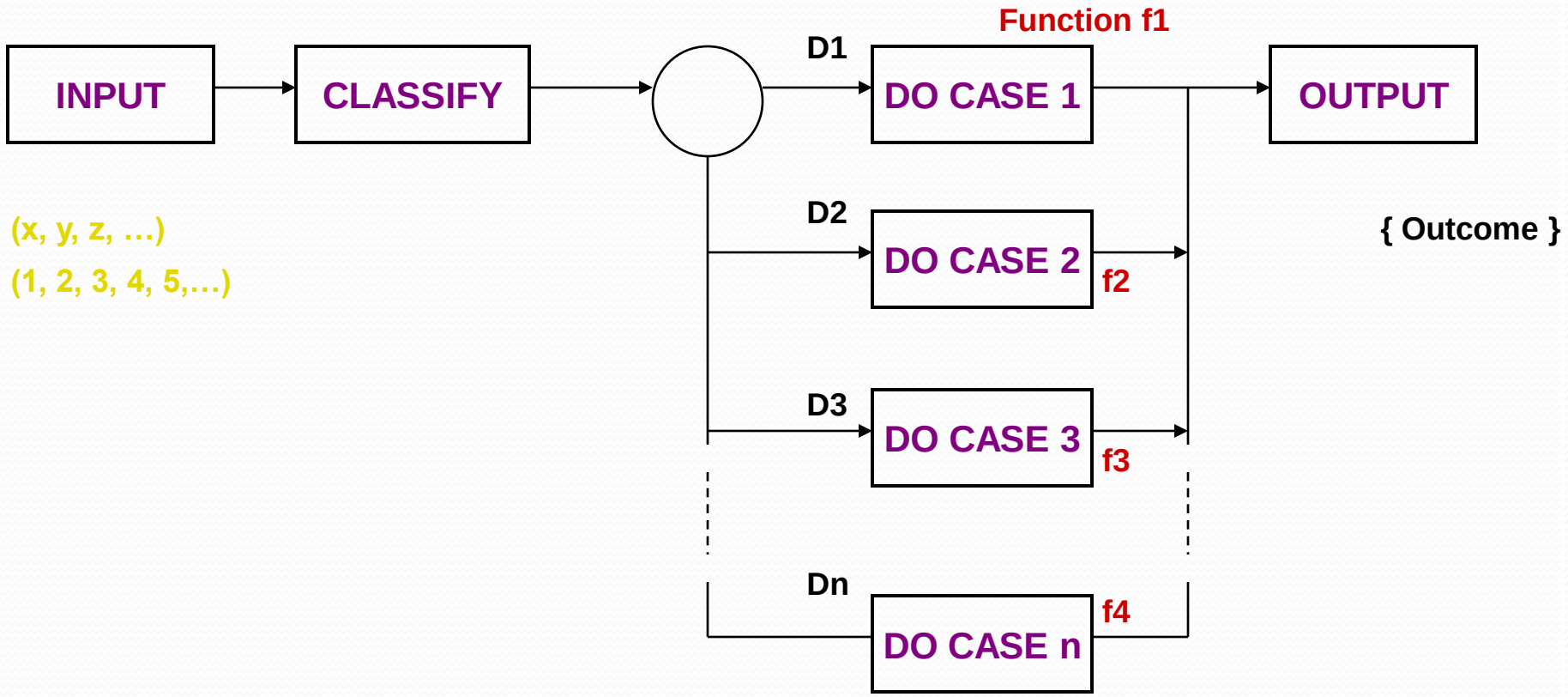
- **Representative outcome**

Partition testing

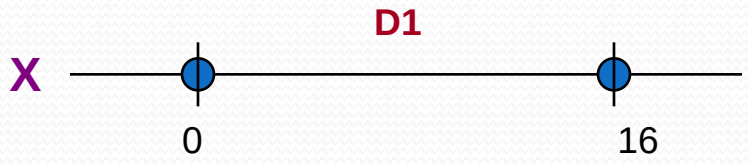
Input Equivalence



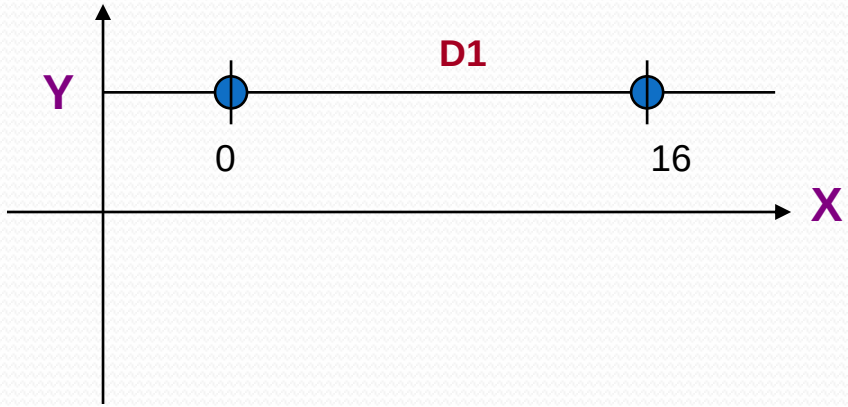
Domain Testing Model



3. Simple boundaries & Compound predicates



X >=0 .AND. X <= 16



Y = 1 .AND. X >=0 .AND. X <= 16

Compound predicates….

- **Impact of .OR.**
  - Concave, Disconnected
  - Adjacent domains with same function
  - Example

A B C + D E F

- **Eliminate compound predicates**

4. Functional Homogeneity of Bugs

- **Functional form still Retained**

  a x + b y >= c

- **Bugs are only in a, b, c**

5. Linear Vector Space

- Linear boundary predicate, Interpreted
- Simple relational operators

- Conversion to linear vector space
  - 2-d Polar co-ordinates
  - Polynomials

- Problems with Non-linear Boundaries

6. Loop-free Software

Predicate for each iteration

Loop over the entire transaction

Definite loop

Nice Domains

Requirements

- Bugs $\Rightarrow$ ill-defined domains

Before DT

- Analyze specs
- Make the Boundary Specs Consistent & Complete

Implemented Domains

- Complete, Consistent & Process all inputs

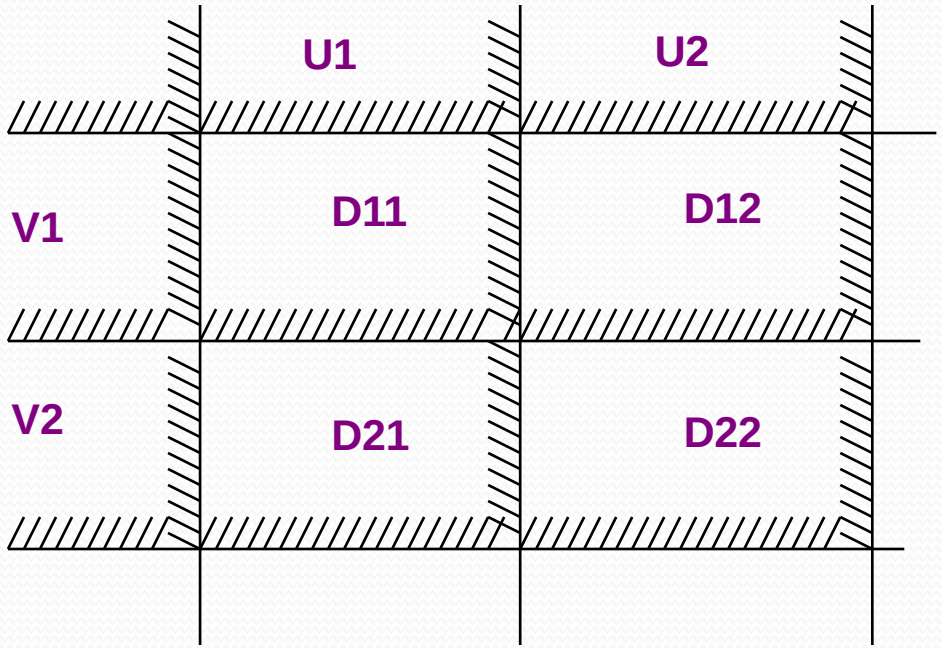
Specified Domains

- Incomplete, Inconsistent

Programmer's / Designer's Effort

Nice Domains

- Linear, Complete, Systematic, Orthogonal, Consistently Closed, & Convex



Nice Domains

Advantages

- **Ease of DT**
- **Fewer Bug Occurrences**

Nice Domains

Boundaries are

1. Linear
2. Complete
3. Systematic
4. Orthogonal
5. Closure consistency
6. Convex
7. Simply connected

Domain Testing - Domains & Paths – Nice domains

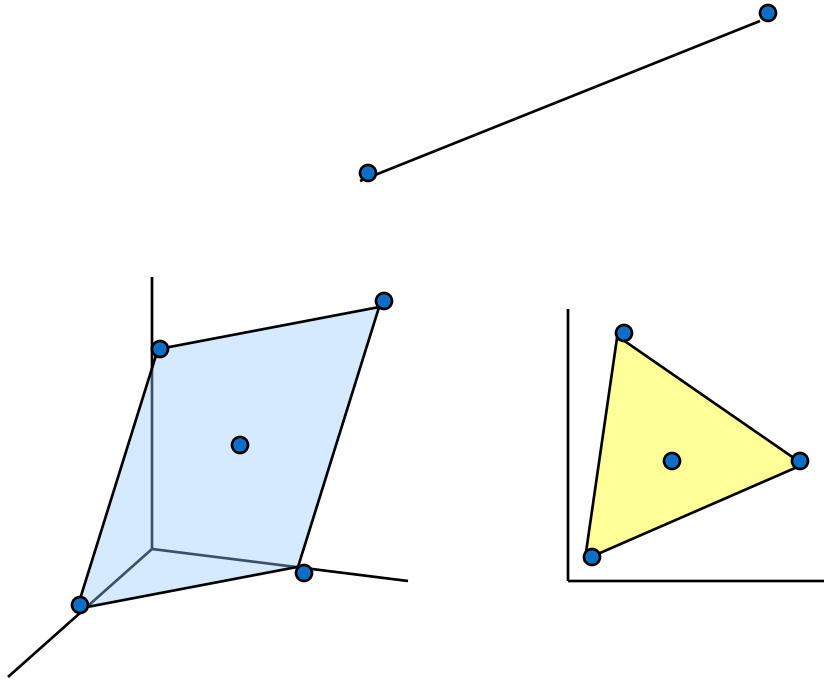
1. Linear Boundaries

Interpreted linear inequalities

n-dim Hyperplane:

n+1 Points

n+1 + 1 Test Cases



Non-Linear

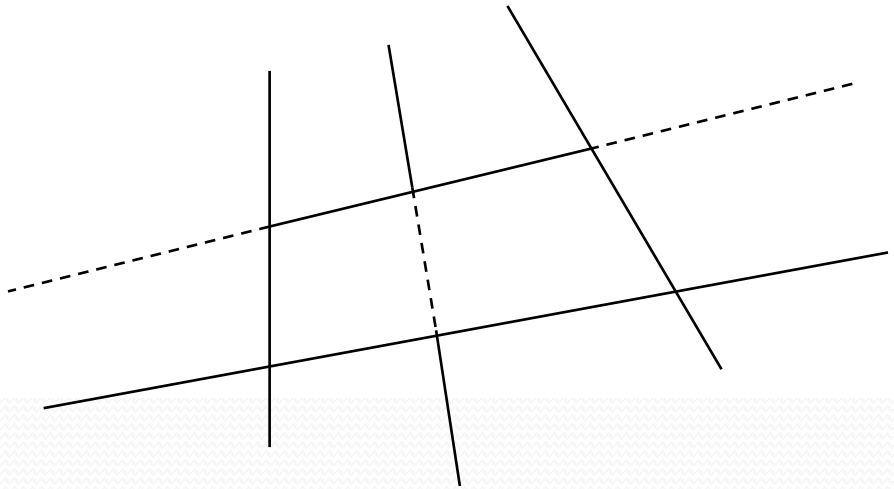
- **Transform**

2. Complete Boundaries

- Span the total number space $(-\infty, +\infty)$
  - One set of Tests

Incomplete…

- Reasons
- Tests

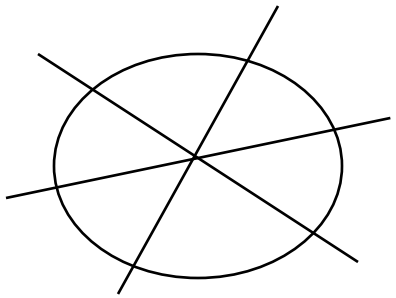


3. Systematic Boundaries

- Linear Inequalities differing by a constant

$f_j(X) \geq k_j$ or, $f_j(X) \geq g(j, c)$ $g(j, c) = j + k * c$

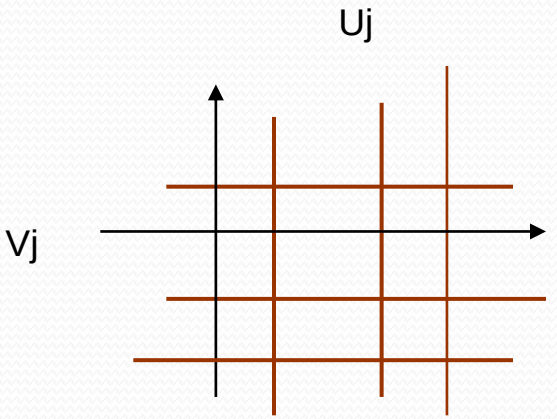
- Parallel lines
- Identical Sectors in a Circle



- **DT**
  - Test a domain $\Rightarrow$ Tests for other Domains

4. Orthogonal Boundaries

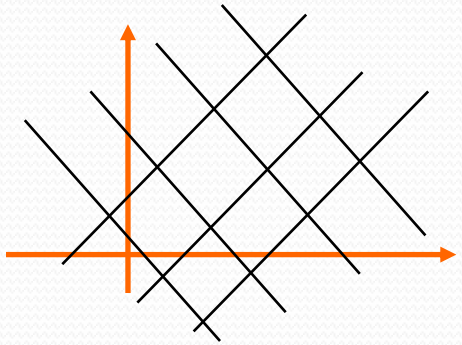
- Two boundaries or, boundary sets
- Parallel to axes



- **DT**
  - Each Set Independently
  - # Tests α O (n)

Orthogonal Boundaries

- Tilted sets
  - transformation
  - Test cases: O(n)

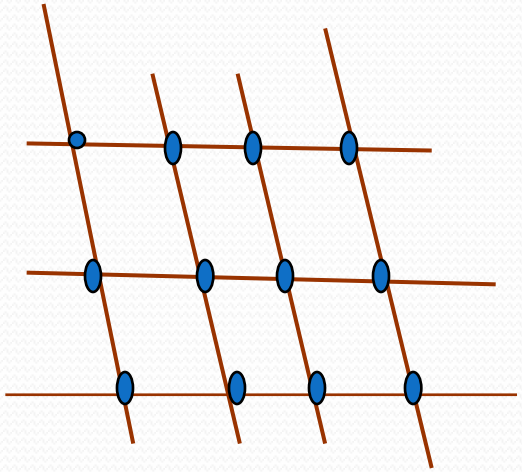


- Concentric circles with radial lines
  - Rectangular coordinates
  - Polar

$r \geq a_j$ .AND. $r < a_{j+1}$ .AND.
$\theta \geq \theta_j$ .AND. $\theta < \theta_{j+1}$

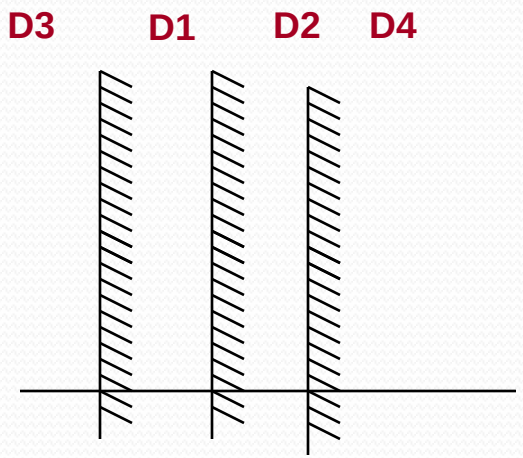
Non-Orthogonal Boundaries

- Test Intersections $\Rightarrow$ $O(n^2) + O(n)$



5. Closure Consistency

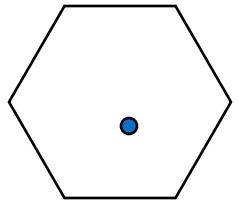
- A Simple pattern in all boundary closures
- Example
  - Same relational operator for systematic boundaries



6. Convex Domain

- Line joining any two points lies with in the domain

- DT
  - n on-points & 1 off-pt

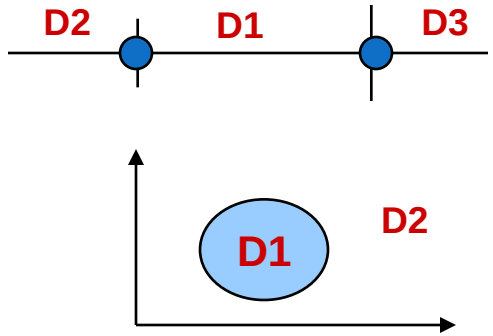


Concave

- " But, However, Except, Or … " in Specs
- Handle with special care

7. Simply Connected Domain

- In a single piece



- 2 complementary domains
  - D1: Convex ⇒ D2: Concave, not-Connected

- Programmer / Designer
  - Convex part First

Domain Testing - Domains & Paths – Ugly Domains

Generally,

From Bad Specs

Programmer / Designer Simplifies => Ugly to Good

possibility of Introduction of more bugs ?!?

Causes

1. Non-linear Boundaries

1. Ambiguities & Contradictions

1. Simplifying the topology

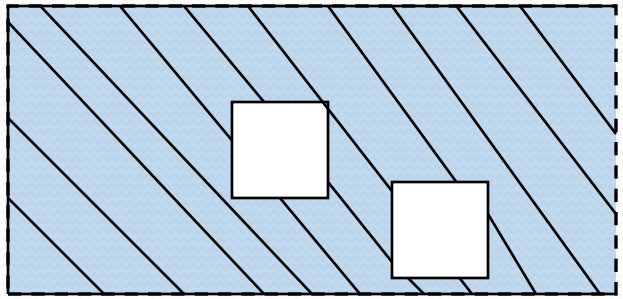
1. Rectifying boundary closures

1. Non-linear Boundaries

- Transform

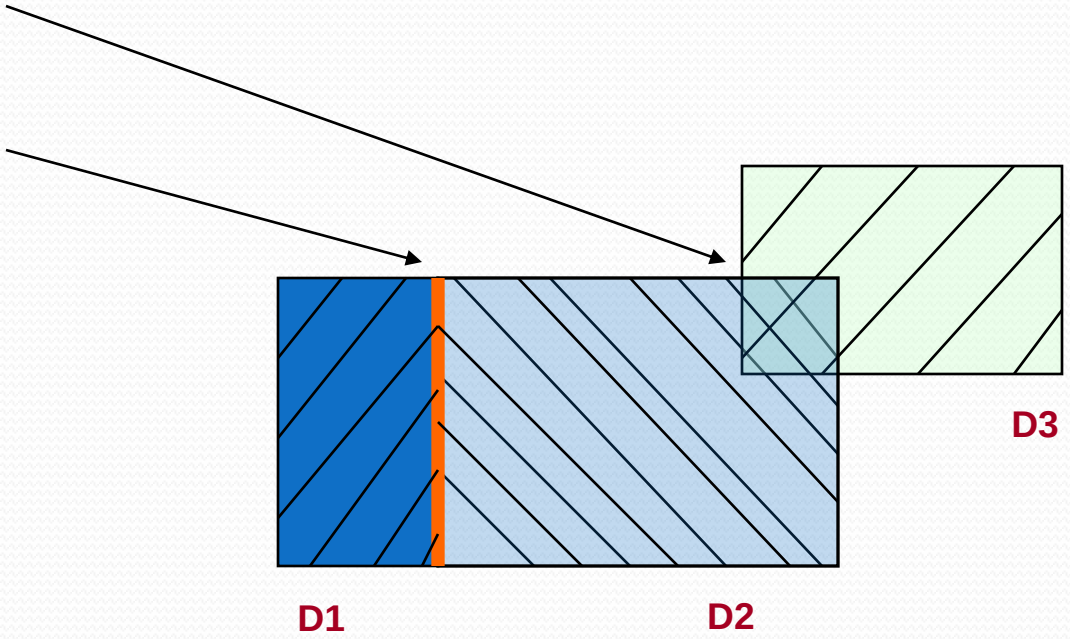
2. Ambiguities

- Holes in input vector space.
- Missing boundary
- Detected by Specification languages & tools.

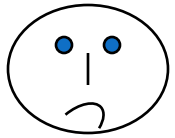


2. Contradictions..

- Overlapping of
  - Domain Specs
  - Closure Specs



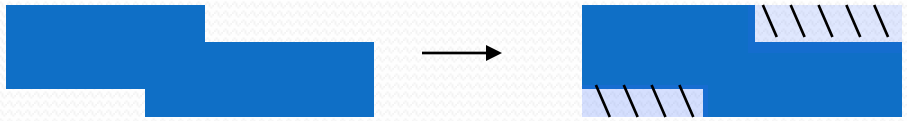
3. Simplifying the Topology….

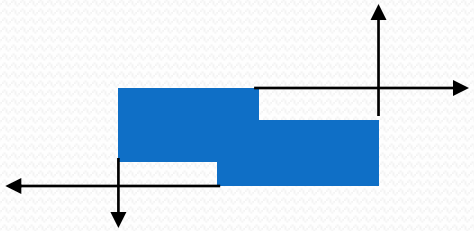


- Complexity !
  - Concavity, Holes, Disconnectedness

3. Simplifying the Topology….

- Smoothing out concavity

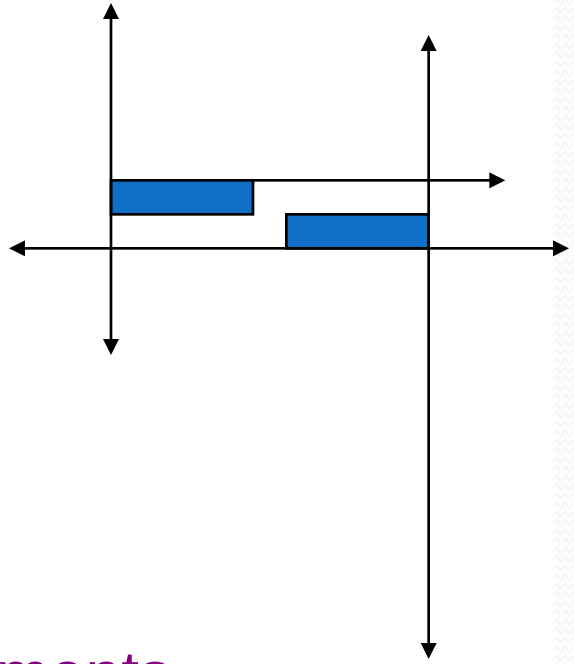




- Filling in Holes

3. Simplifying the Topology….

- Joining the pieces

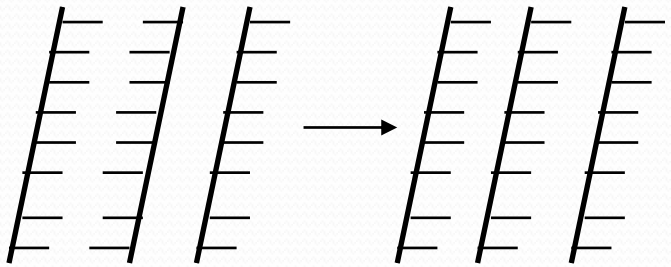


Correct:

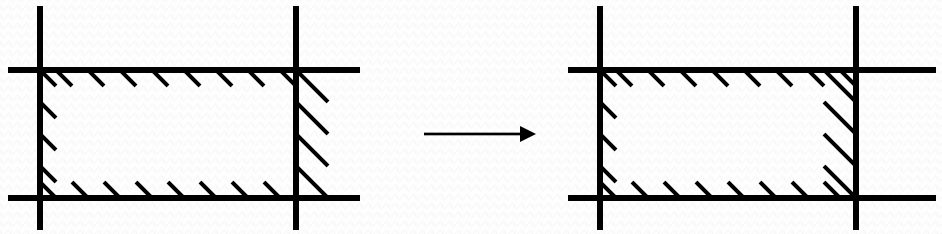
- Connect disconnected boundary segments
- Extend boundaries to infinity

4. Rectifying Boundary Closures.

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



Consistent Direction



Inclusion / Exclusion Consistency

General DT Strategy

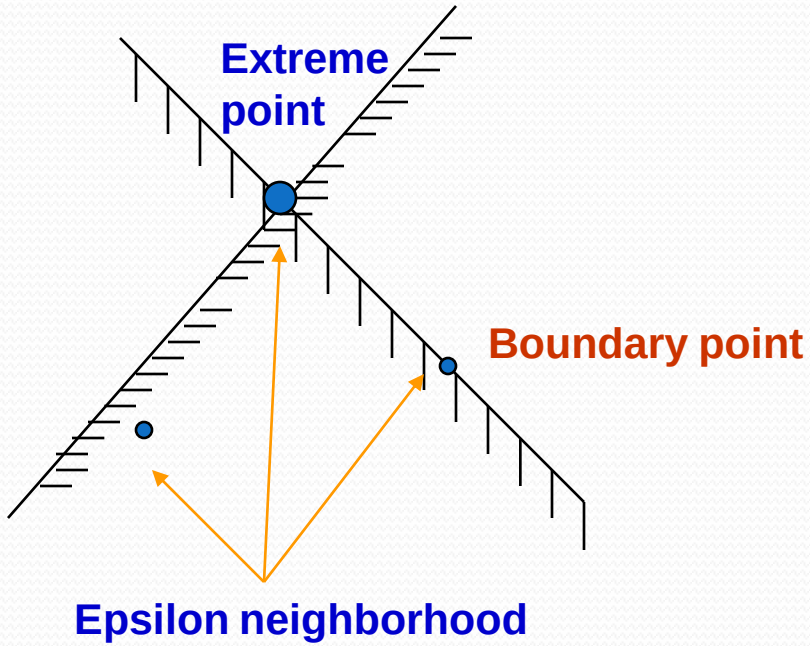
1. Select test points near the boundaries.
2. Define test strategy for each possible bug related to boundary
3. Test points for a domain useful to test its adjacent domain.
4. Run the tests. By post test analysis determine if any boundaries are faulty & if so how?
5. Run enough tests to verify every boundary of every domain

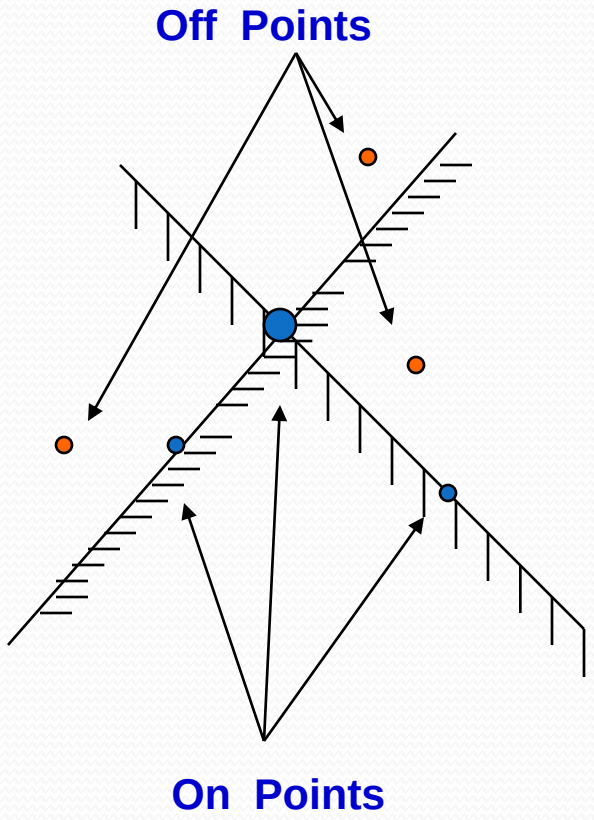
DT for Specific Domain Bugs

Generally,

- Interior point
- Exterior point
- Epsilon neighborhood
- Extreme point
- On point

DT for Specific Domain Bugs



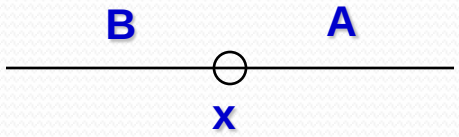


Domain D1

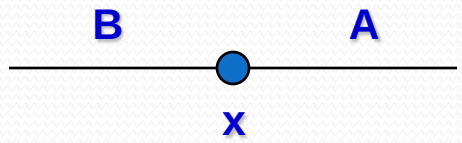
Domain Testing - Domains & Paths – Domain Testing

1. 1-d Domains
2. 2-d Domains
3. Equality & inequality Predicates
4. Random Testing
5. Testing n-dimensional Domains

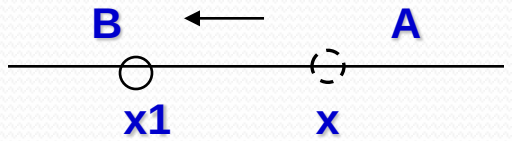
Testing 1-d Domains : Bugs with open boundaries



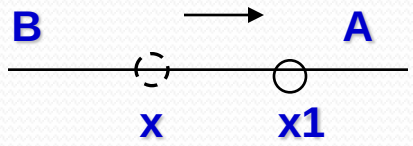
Closure Bug



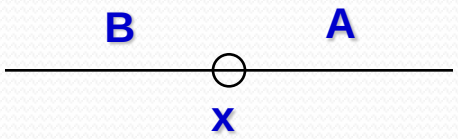
Shift left Bug



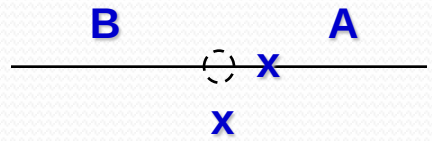
Shift Right Bug



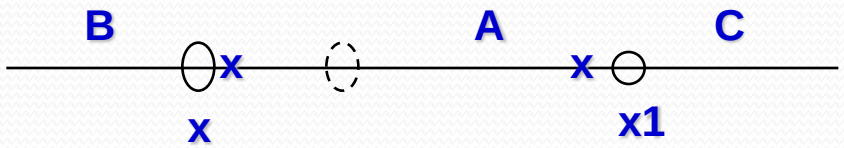
Testing 1-d Domains : Bugs with open boundaries



Missing Boundary



Extra Boundary

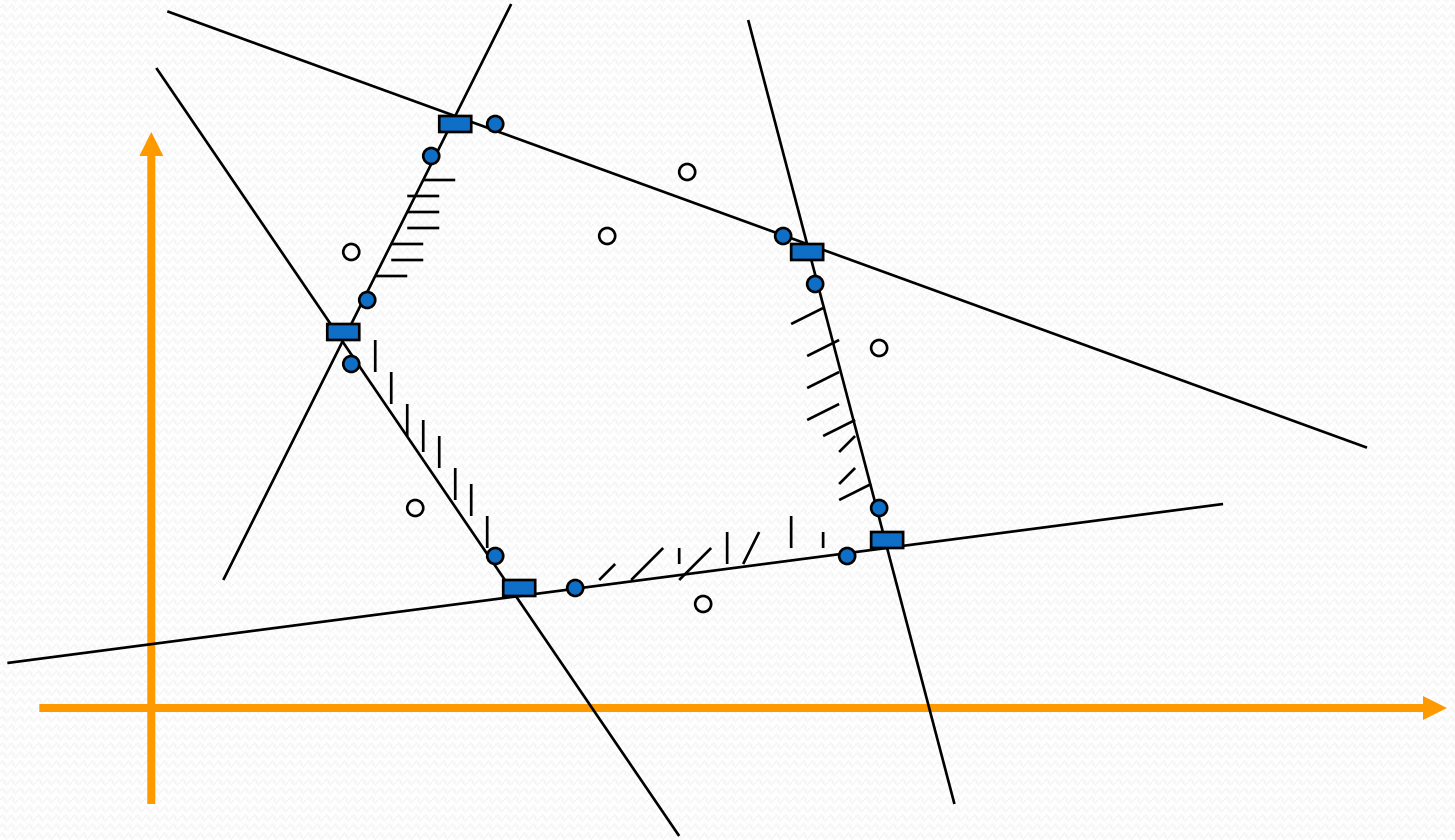


Bugs with Closed Boundaries : Similar to the above

2. Testing 2-d Domains

- Closure bug
- Boundary Shift : up / down
- Tilted Boundary
- Extra Boundary
- Missing Boundary

Domain Testing - Domains & Paths – Domain Testing

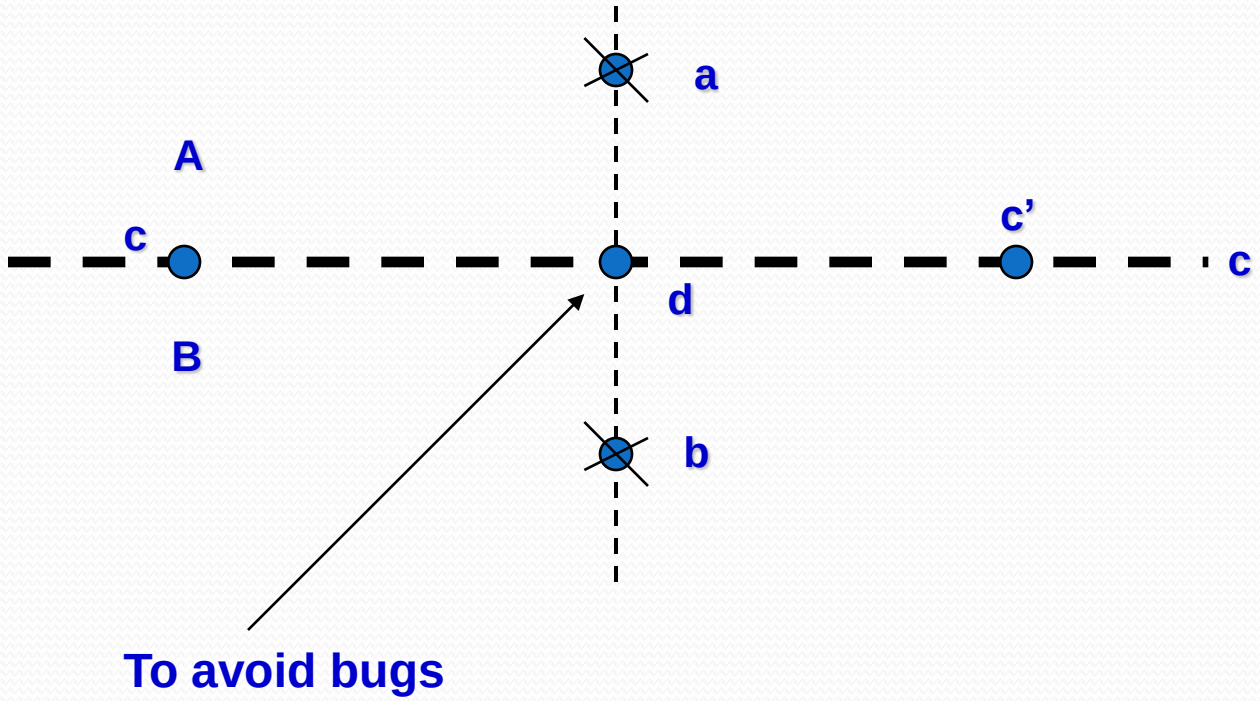


2 n : Domains share tests
3 n : no sharing of tests by domains

Strategy for domain testing in 2-dim

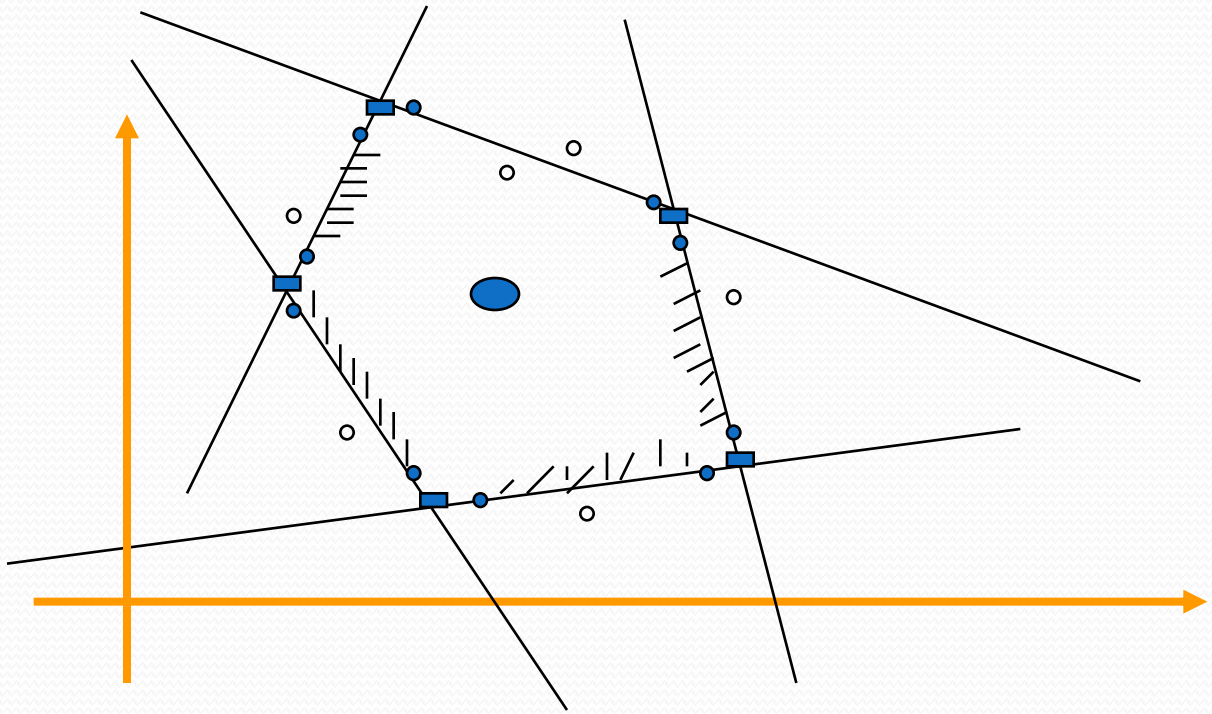
3. Equality & Inequality Predicates

- An Equality predicate defines a line in 2-d



4. Random Testing

- A Point in the center : verifies computation



5. Testing n-Dimensional Domains (strategy)

n-dimensions, p boundary segments

- (n+1)*p test cases : n on points & 1 off point
- Extreme pt shared : 2 * p points
- Equalities over m-dimensions create a subspace of n-m dimensions
- Orthogonal domains with consistent boundary closures, orthogonal to the axes & complete boundaries
    - Independent testing

Procedure for solving for values

Simple procedure. Need tools.

1. Identify input variables

2. Identify variables which appear in domain-defining predicates, such as control-flow predicates

Procedure

3. Interpret all domain predicates in terms of input variables:
   - Transform non-linear to linear
   - Find data flow path

4. Predicate expression with p # predicates.
   Find # domains : $< 2^p$

3. Solve inequalities for extreme points

4. Use extreme points to solve for nearby on points …

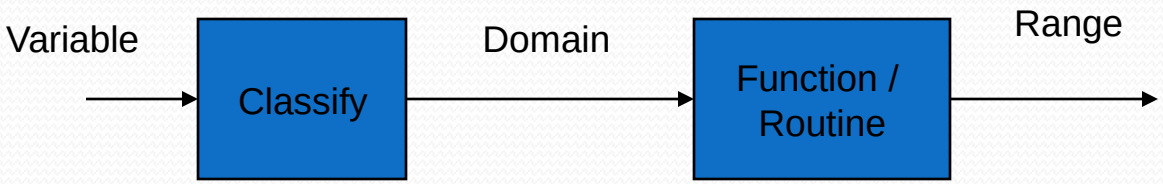
Domain Testing

Effectiveness

- Cost effective
- Bugs on boundaries, extreme points
- Hardware logic testing – tool intensive

Domains & Interface Testing

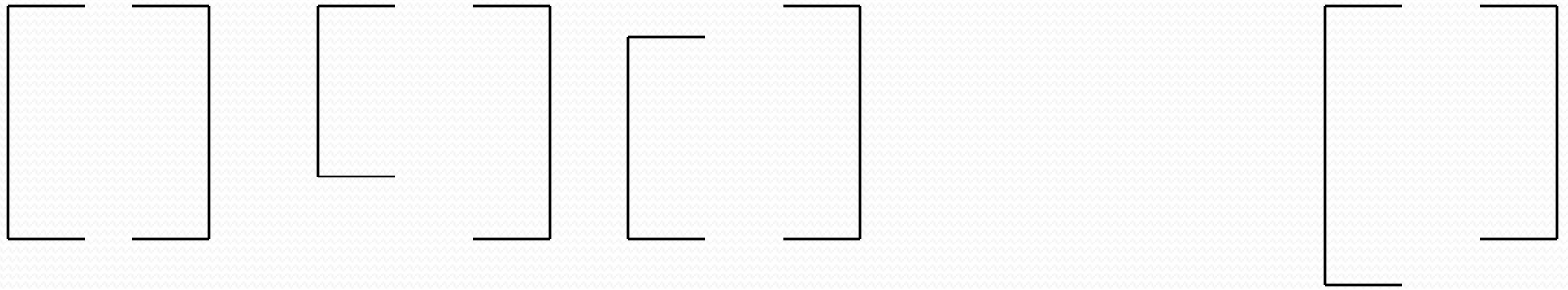
- Domain
- Range



Domain Testing

Domains & Interface Testing

- Span compatibility



Domain Testing

Interface Range/Domain Compatibility Testing

- Test each var independently
- Find an inverse function
- Build a super domain

Domain Testing

Domains and Testability

- **Linearizing transformations**
  - Polynomial
  - Rectangular to polar
  - Generic & rational => Taylor series

Domains and Testability

Perform transformations for better testing.

- **Co-ordinate transformations**

Domains and Testability

- Canonical Program Form

UNIT-IV

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

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

Path Products & expressions - Purpose

PURPOSE: APPLICATIONS

1. How many paths in a flow-graph?

   Maximum, minimum etc.

2. The probability of getting to a point in a program ?

   (to a node in a flow graph)

3. The mean processing time of a routine (a flow graph)

4. Effect of Routines involving complementary operations :

   (Push / Pop & Get / Return)

PURPOSE & APPLICATIONS

1. **Check for data flow anomalies.**

2. **Regular expressions are applied to problems in test design & debugging**

3. electronics engineers use flow graphs to design & analyze circuits & logic designers.

1. Software development, testing & debugging tools use flow graph analysis tools & techniques.

1. These are helpful for test tool builders.

Motivation

1. Flow graph is an abstract representation of a program.

2. A question on a program can be mapped on to an equivalent question on an appropriate flow graph.

3. It will be a foundation for syntax testing & state testing

Path Products & expressions - Definitions

Path Expression:

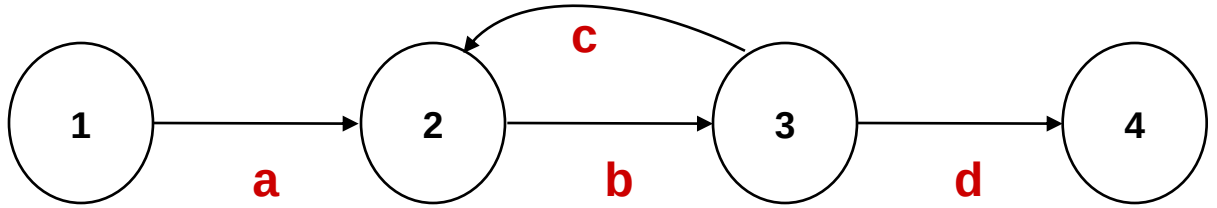
An algebraic representation of sets of paths in a flow graph.

Regular Expression:

Path expressions converted by using arithmetic laws & weights into an algebraic function.

Path Products & expressions – Path Product

- Annotate each link with a name.
- The pathname as you traverse a path (segment) expressed as concatenation of the link names is the path product.

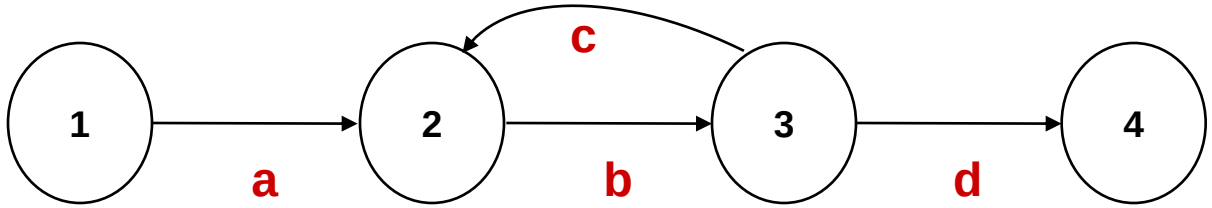


- Examples of path products between 1 & 4 are:

a b d a b c b d a b c b c b d ….

Path Expression

Simply: Derive using path products.



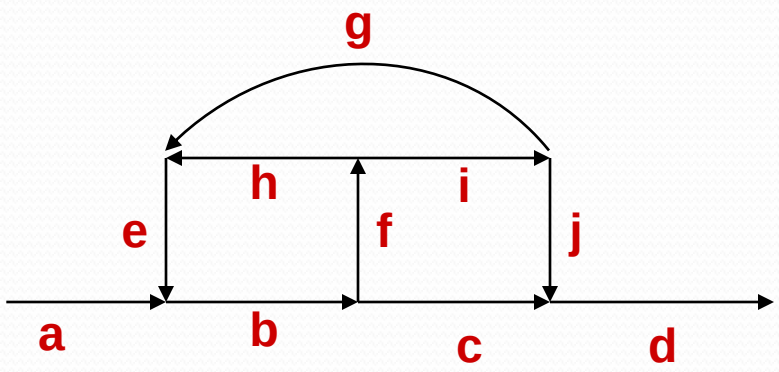
Example:

{ a b d , a b c b d, a b c b c b d , ….. }

abd + abcbd + abcbcbd + ….

Path Products & expressions – Path Expression

Example:



{ abcd , abfhebcd , abfigebcd , abfijd }

abcd + abfhebcd + abfigebcd + abfijd

Path Products & expressions – Path Expression

Path name for two successive path segments is the concatenation of their path products.

X = abc **Y = def** **XY = abcdef**

a X = aabc **X a = abca** **XaX = abcaabc**

X = ab + cd **Y = ef + gh** **XY = abef + abgh + cdef + cdgh**

Path Products & expressions – Path Segments & Products

- **Loops:**

$a^1 = a$ $a^2 = aa$ $a^3 = aaa$ $a^n = aaaaa \ldots$ n times

$X = abc$ $X^1 = abc$ $X^2 = (abc)^2 = abcabc$

- **Identity element**

$a^0 = 1$ $X^0 = 1$ (path of length 0)

Path Products & expressions – Path Product

Path Product

- **Not Commutative:**

  $XY \neq YX$ in general

- **Associative**

  $A(BC) = (AB)C = ABC$: Rule 1

Path Products & expressions – Path Product

Denotes a set of paths in parallel between two nodes.

- Commutative

$$X + Y = Y + X \quad : \textbf{Rule 2}$$

- Associative

$$(X + Y) + Z = X + (Y + Z) = X + Y + Z \quad : \textbf{Rule 3}$$

- Distributive

$$A(B + C) = AB + AC \quad : \textbf{Rule 4}$$
$$(A + B)C = AC + BC$$

Path Products & expressions – Path Products

- **Absorption**

X + X = X : **Rule 5**

X + any subset of X = X

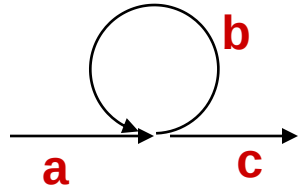
X = a + bc + abcd

X + a = X + bc + abcd = X

Path Products & expressions – Path Products

- **Loop:**

An infinite set of parallel paths.



$b^* = b^0 + b^1 + b^2 + b^3 + \ldots\ldots$

$X^* = X^0 + X^1 + X^2 + X^3 + \ldots\ldots$

$X^+ = X^1 + X^2 + X^3 + \ldots\ldots$

- $X\ X^* = X^*\ X = X^+$
- $a\ a^* = a^*\ a = a^+$

$X^{\underline{n}} = X^0 + X^1 + X^2 + X^3 + \ldots\ldots + X^n$

Path Products & expressions – Path products

More Rules…

$X^{\underline{m}} + X^{\underline{n}} = X^{\underline{n}}$ if $n \geq m$: **Rule 6**

$= X^{\underline{m}}$ if $n < m$

$X^{\underline{m}} X^{\underline{n}} = X^{\underline{m+n}}$: **Rule 7**

$X^{\underline{n}} X^{*} = X^{*} X^{\underline{n}} = X^{*}$: **Rule 8**

$X^{\underline{n}} X^{+} = X^{+} X^{\underline{n}} = X^{+}$: **Rule 9**

$X^{*} X^{+} = X^{+} X^{*} = X^{+}$: **Rule 10**

Identity Elements ..

1 : Path of Zero Length

$1 + 1 = 1$: **Rule 11**

$1X = X1 = X$: **Rule 12**

$1^{\underline{n}} = 1^{n} = 1^{*} = 1^{+} = 1$: **Rule 13**

$1^{+} + 1 = 1^{*} = 1$: **Rule 14**

Identity Elements ..

0 : empty set of paths

$X + 0 = 0 + X = X$: **Rule 15**

$X\,0 = 0\,X = 0$: **Rule 16**

$0^* = 1 + 0 + 0^2 + 0^3 + \ldots = 1$: **Rule 17**

Path Products & expressions – Reduction Procedure

- To convert a flow graph into a path expression that denotes the set of all entry/exit paths.

- Node by Node Reduction Procedure

Path Products & expressions – Reduction Procedure

Initialization Steps:

1. Combine all serial links by multiplying their path expressions.

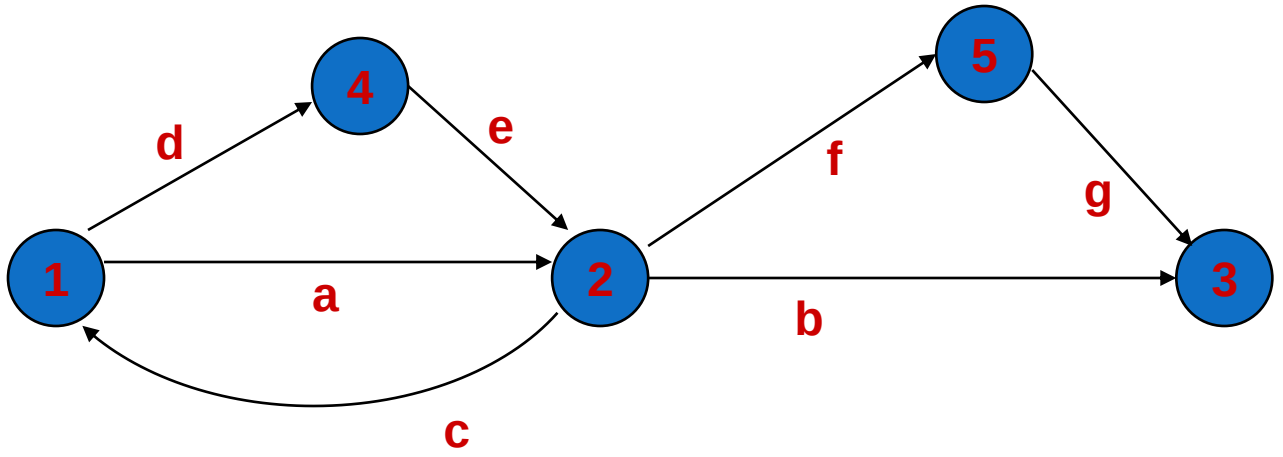
1. Combine all parallel links by adding their path expressions.

1. Remove all self-loops - replace with links of the form X*

Steps in the Algorithm's loop

4. Select a non-initial & non-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 in-links with the set of out-links of that node.



Path Products & expressions – Reduction Procedure

Steps in the Algorithm's loop:

5. Combine any serial links by multiplying their path expressions.
   (as in step 1)

6. Combine any parallel links by adding their path expressions.
   (as in step 2)

7. Remove all the self-loops.
   (as in step 3)

8. IF there's just one node between entry & exit nodes, path expression for the flow graph is the link's path expression.
   ELSE, return to step 4.

Path Products & expressions – Reduction Procedure

Path Expression for a Flow Graph

- is not unique
- depends on the order of node removal.

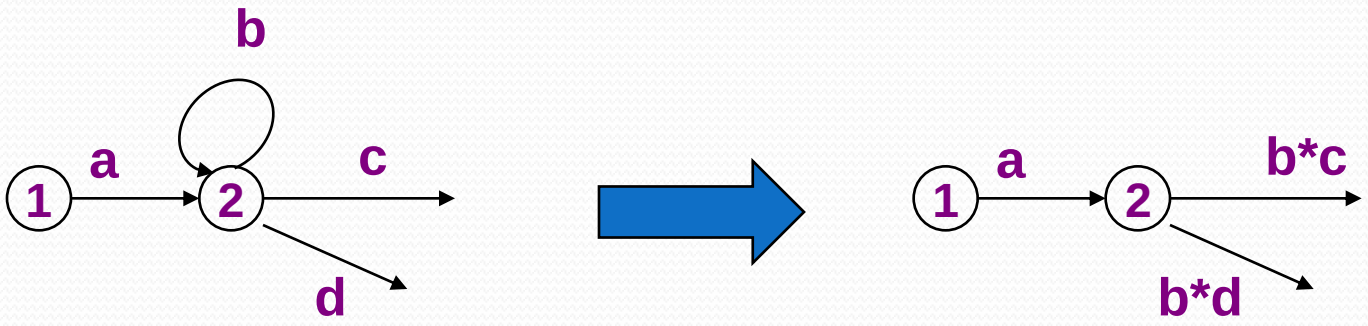
Path Products & expressions – Reduction Procedure Example

Cross-Term Step (Step 4 of the algorithm)

- Fundamental step.
- Removes nodes one by one till there's one entry & one exit node.
- Replace the node by path products of all in-links with all out-links and interconnecting its immediate neighbors.

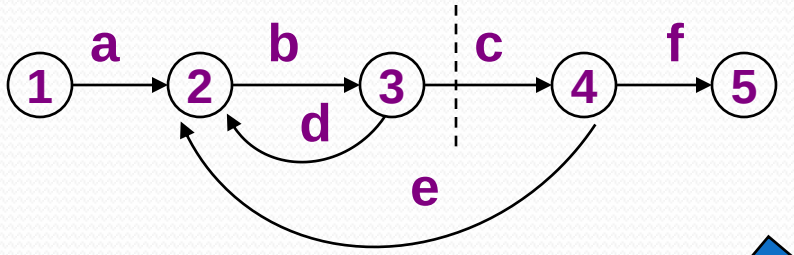
Path Products & expressions – Reduction Procedure Example

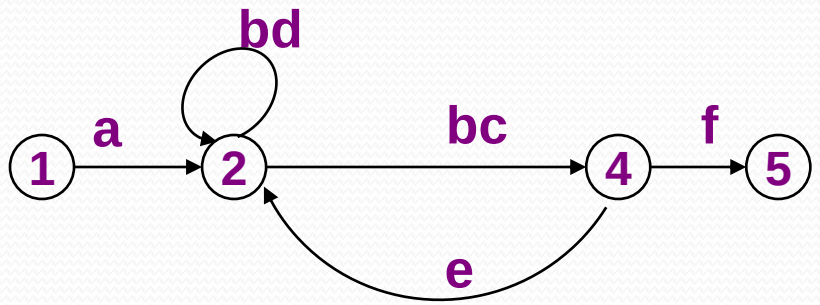
Processing of Loop Terms:

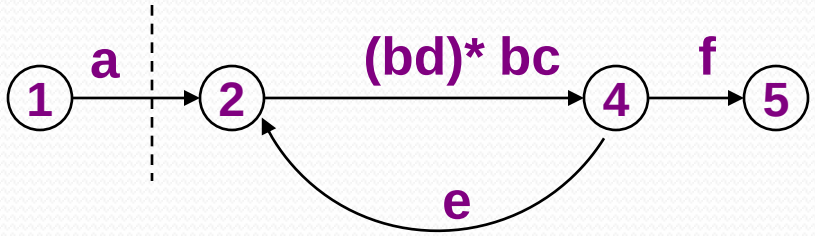


Path Products & expressions – Reduction Procedure Example

Processing of Loop Terms:

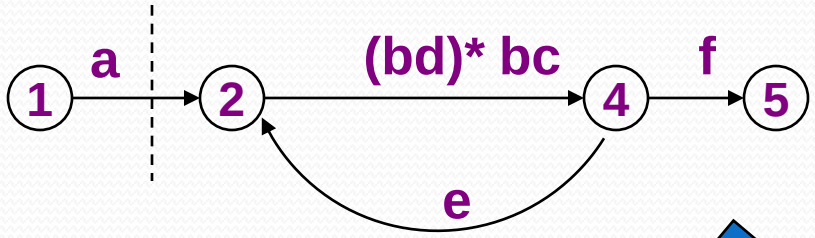


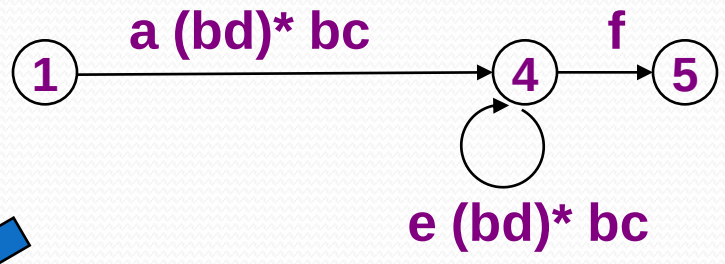


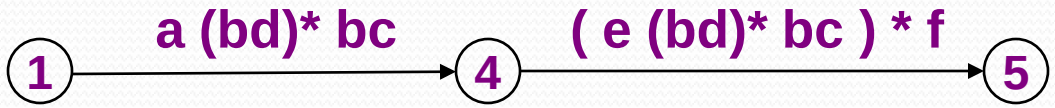


Path Products & expressions – Reduction Procedure Example

Processing of Loop Terms:



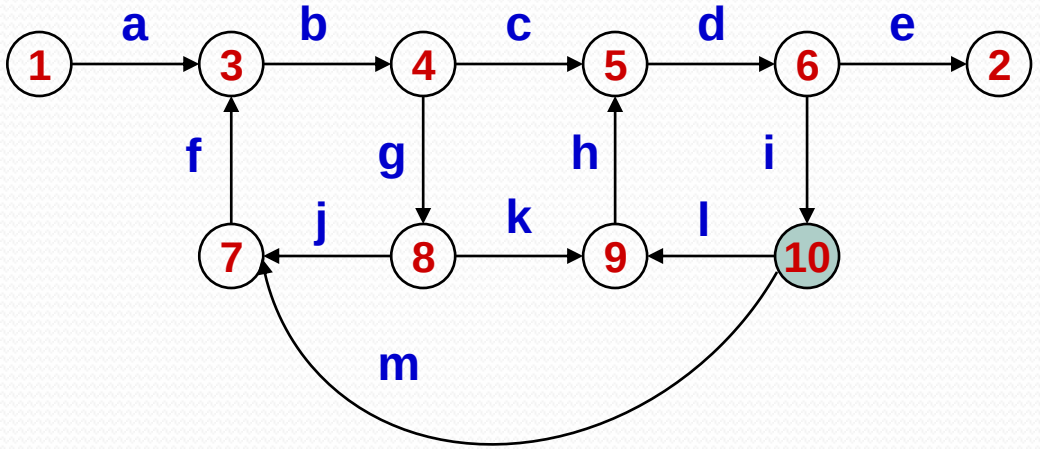


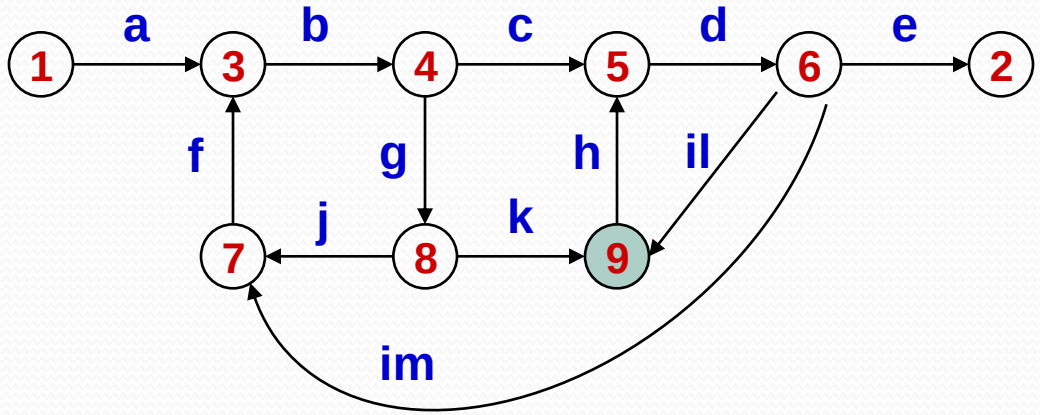


a (bd)* bc (e (bd)* bc)* f

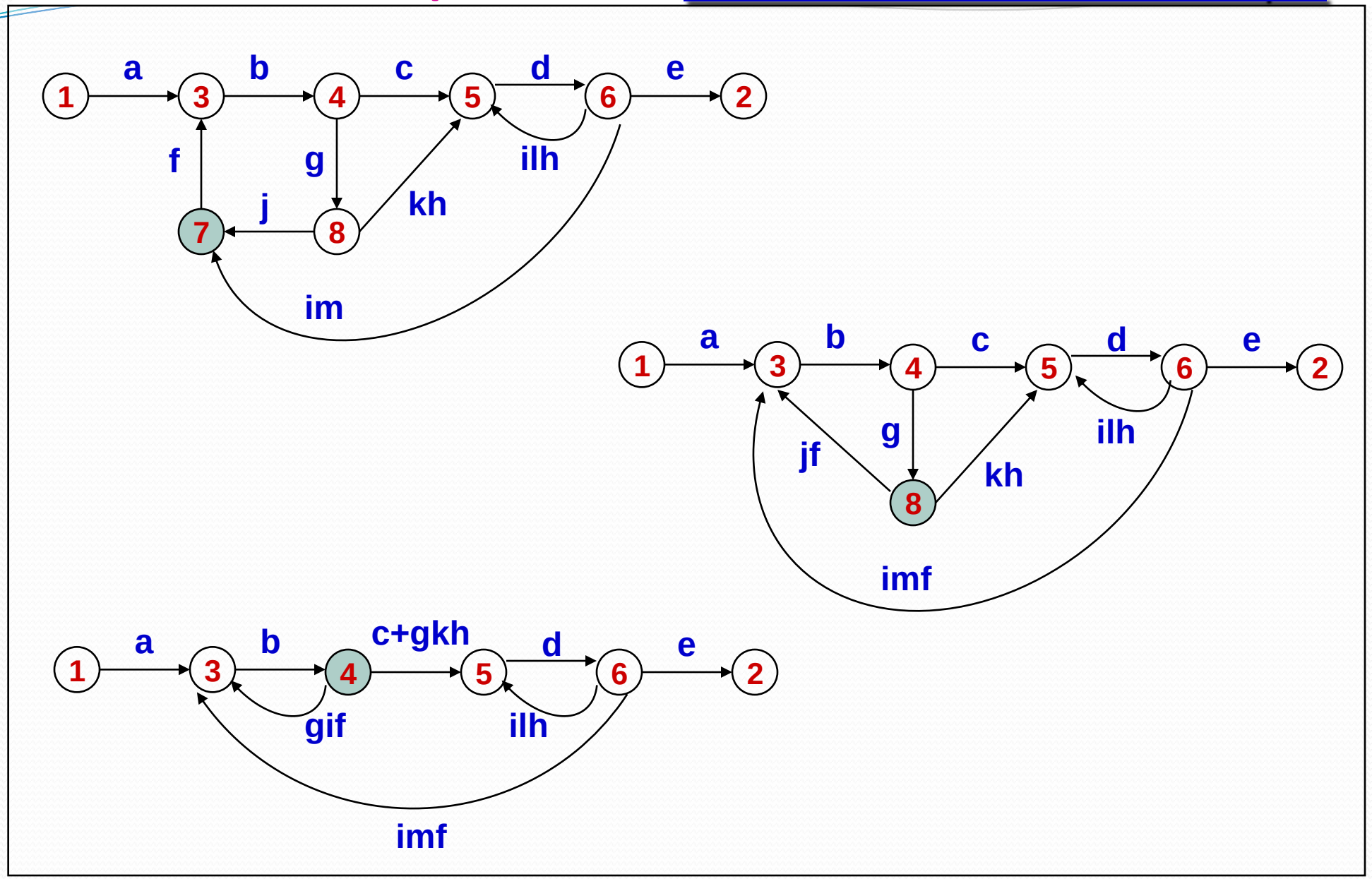
Path Products & expressions – Reduction Procedure Example

Example:

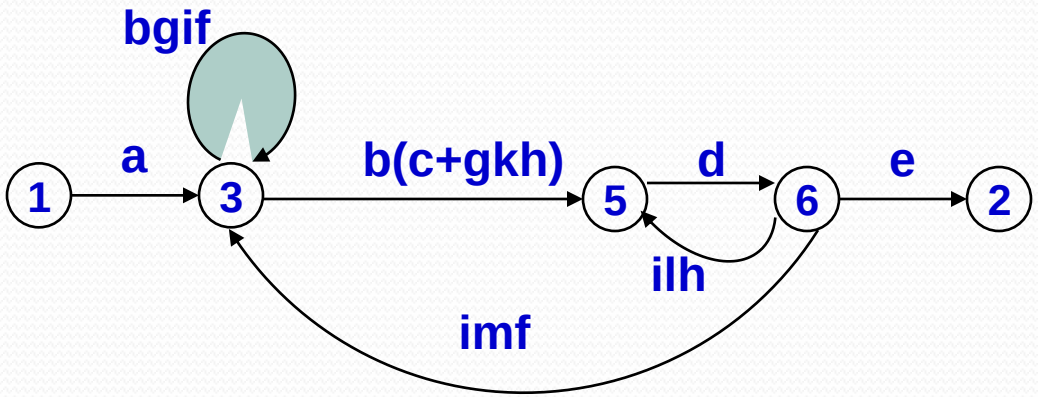


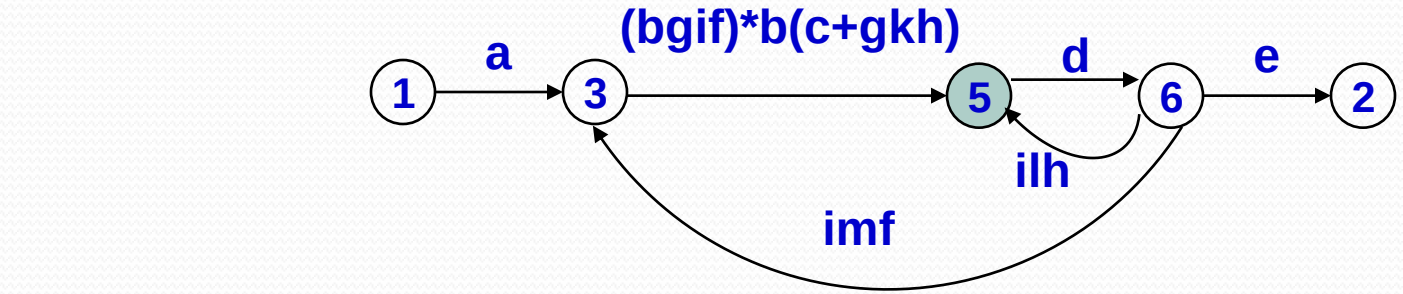


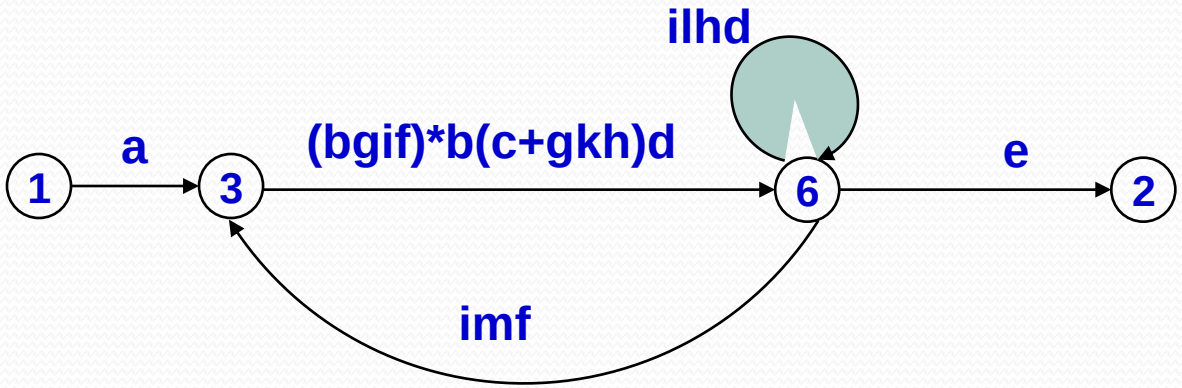
Path Products & expressions – Reduction Procedure Example



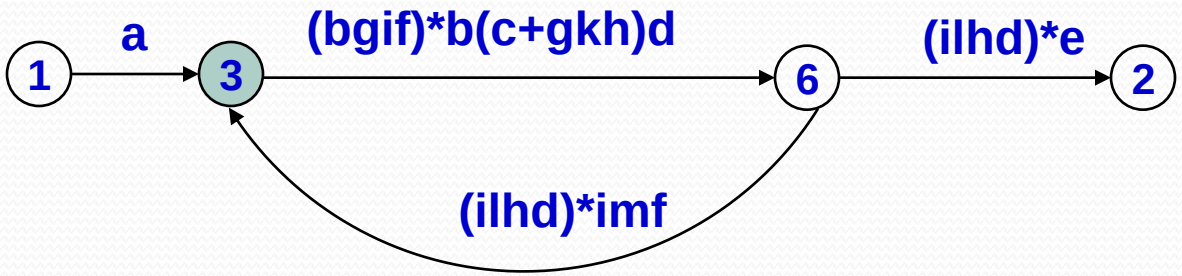
Path Products & expressions – Reduction Procedure Example

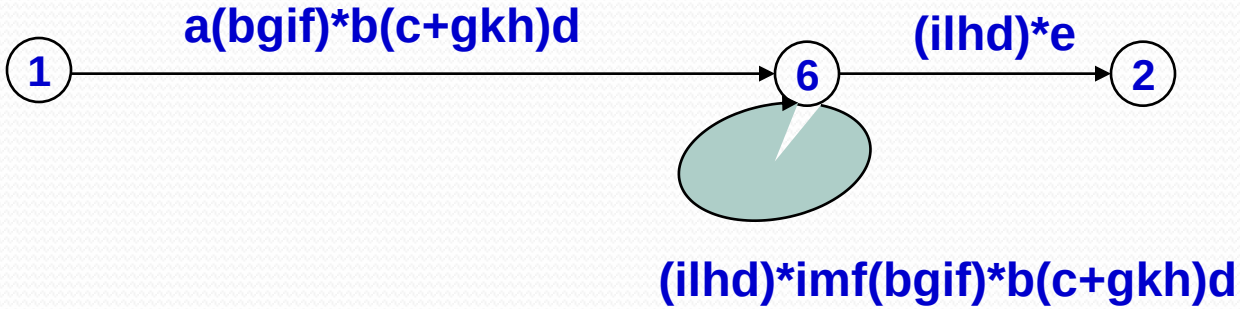




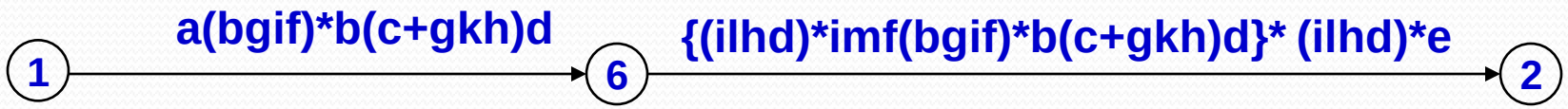


Path Products & expressions – Reduction Procedure Example





Path Products & expressions – Reduction Procedure Example



Flow Graph Path Expression :

a(bgif)*b(c+gkh)d {(ilhd)*imf(bgif)*b(c+gkh)d}* (ilhd)*e

Path Products & expressions – Before going into Applications

Before that, we learn:

- Identities
- Structured Flow Graphs (code/routines)
- Unstructured Flow graphs (routines)

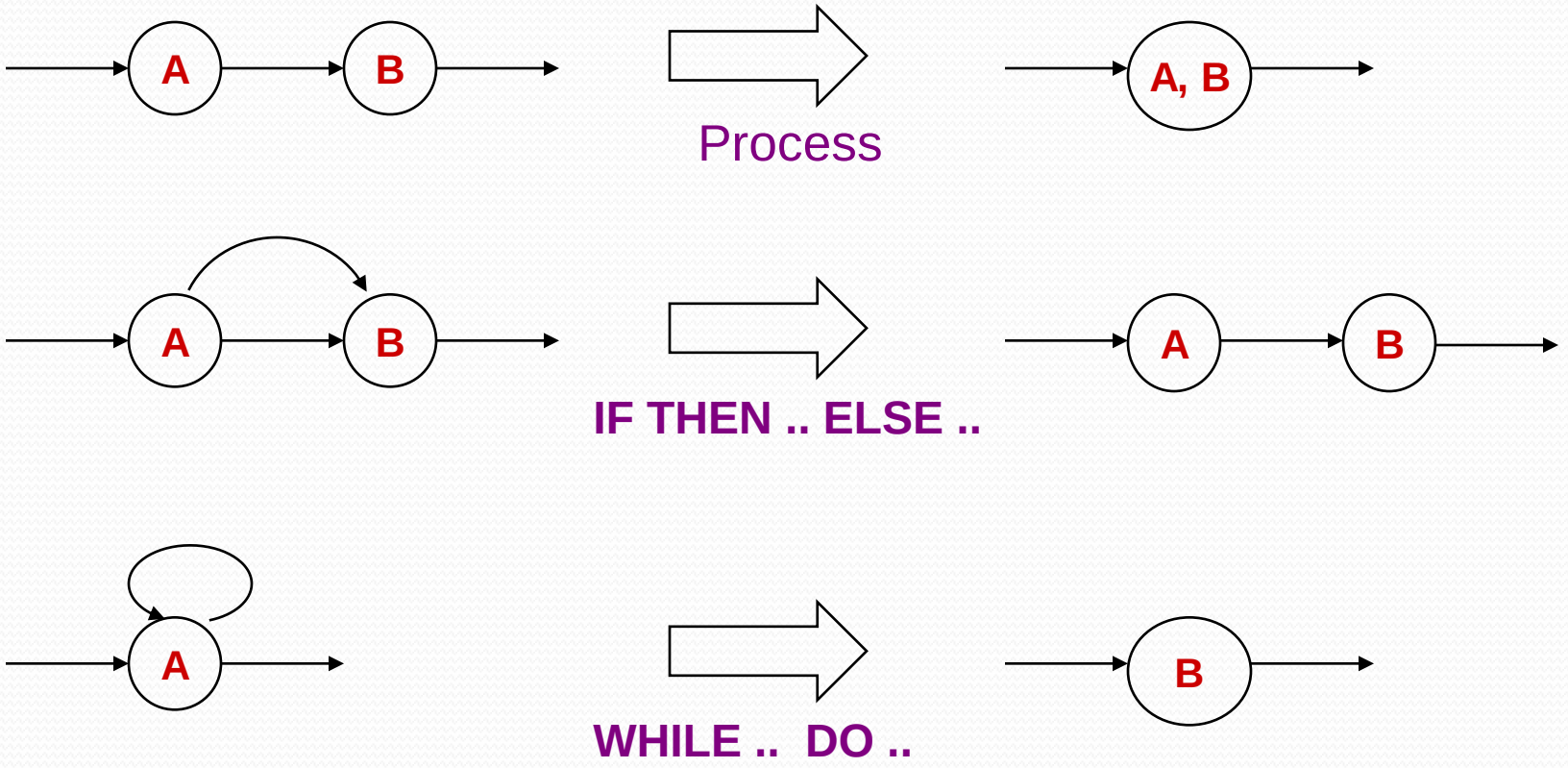
Path Products & expressions – Identities / Rules

(A + B) *	=	(A* + B*) *	: I1
	=	(A* B*)*	: I2
	=	(A* B)* A*	: I3
	=	(B* A)* B*	: I4
	=	(A* B + A)*	: I5
	=	(B* A + B)*	: I6
(A + B + C + . . .) *	=	(A* + B* + C* + . . .)*	: I7
	=	(A* B* C* . . .)*	: I8

Derived by removing nodes in different orders & applying the series-parallel-loop rules.

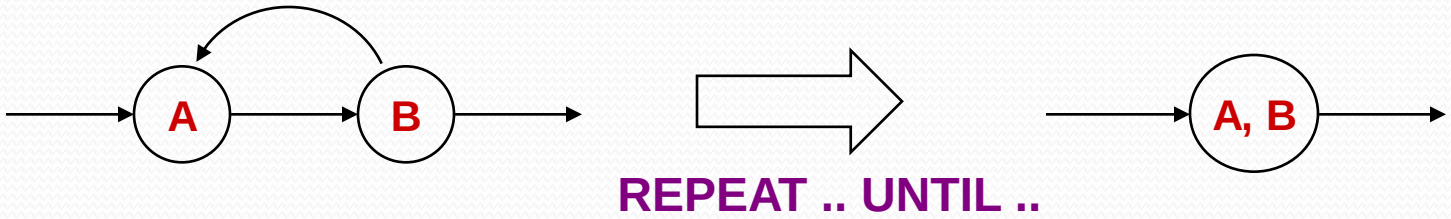
Path Products & expressions – Structured Flow Graphs

Reducible to a single link by successive application of the transformations shown below.



Path Products & expressions – Structured Flow Graphs

Structured flow graph transformations

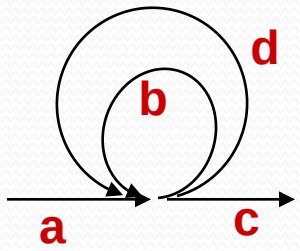


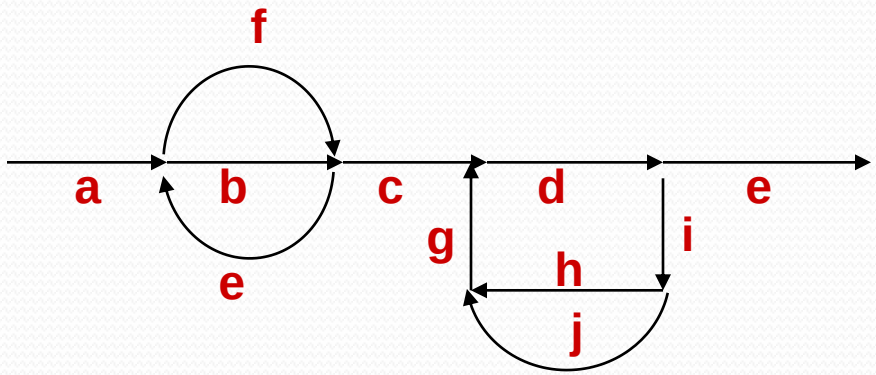
Properties:

- No cross-term transformation.
- No GOTOs.
- No entry into or exit from the middle of a loop.

Path Products & expressions – Structured Flow Graphs

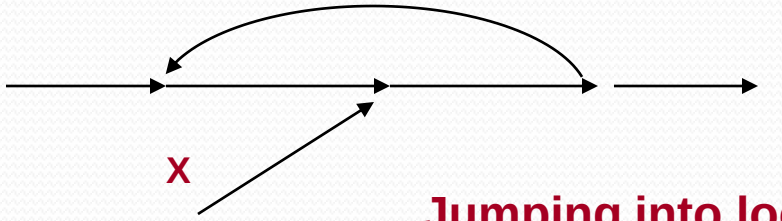
Some examples:



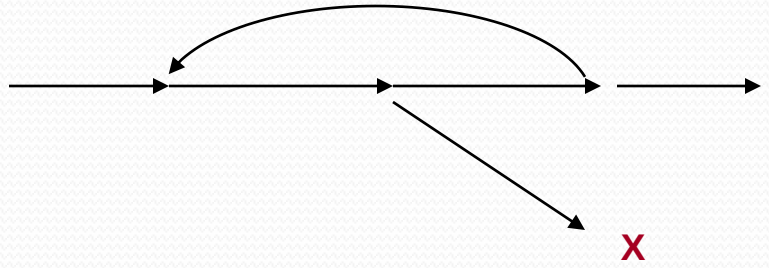


Path Products & expressions – UNstructured Flow Graphs

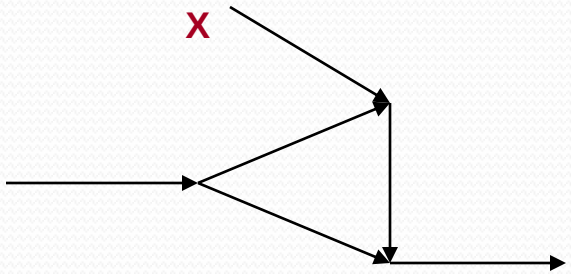
Some examples – unstructured flow graphs/code:



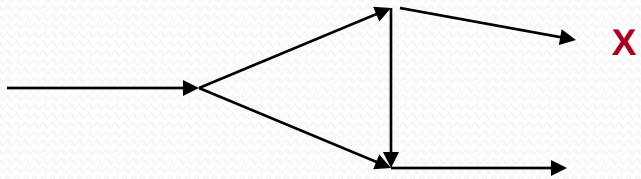
Jumping into loops



Jumping out of loops



Branching into Decisions



Branching out of Decisions

Logic based testing: Overview

- Logic is used in a program by programmers. Boolean algebra is the way to work with logic – simplification & calculation.
- **Hardware logic testing** – hardware logic test design tools and methods use logic & Boolean algebra. Hardware design language compilers/translators use logic & Boolean algebra.
- Impact of errors in specifications of a software is high as these are first in and last out. So, higher level language for specs is desired to reduce the number of errors. Higher order logic systems are used for formal specifications. The tools to simplify, transform and check specs use Boolean algebra.

Logic based testing: Overview

Knowledge based systems:

- Knowledge based systems and artificial intelligence systems use high level logic languages which are based on rule bases consisting of rules. Rules are predicate expressions containing domain knowledge related elements combined with logical connectives. The answers to queries (problems) are derived based on Boolean algebraic operations performed on the rule bases. Such programs are called inference engines.

Modeling Logic with Decision Tables

- A matrix representation of the logic of a decision
- Specifies the possible conditions and the resulting actions
- Best used for complicated decision logic

Modeling Logic with Decision Tables

- Consists of three parts
  - Condition stubs
    - Lists condition relevant to decision
  - Action stubs
    - Actions that result from a given set of conditions
  - Rules
    - Specify which actions are to be followed for a given set of conditions

Modeling Logic with Decision Tables

- Indifferent Condition
  - Condition whose value does not affect which action is taken for two or more rules
- Standard procedure for creating decision tables
  - Name the condition and values each condition can assume
  - Name all possible actions that can occur
  - List all rules
  - Define the actions for each rule
  - Simplify the table

Complete decision table for payroll system example

	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Constructing a Decision Table

- PART 1. FRAME THE PROBLEM.
  - Identify the conditions (decision criteria). These are the factors that will influence the decision.
    - E.g., We want to know the total cost of a student's tuition. What factors are important?
  - Identify the range of values for each condition or criteria.
    - E.g. What are they for each factor identified above?
  - Identify all possible actions that can occur.
    - E.g. What types of calculations would be necessary?

Constructing a Decision Table

- PART 2. CREATE THE TABLE.
  - Create a table with 4 quadrants.
    - Put the conditions in the upper left quadrant. One row per condition.
    - Put the actions in the lower left quadrant. One row per action.
  - List all possible rules.
    - Alternate values for first condition. Repeat for all values of second condition. Keep repeating this process for all conditions.
    - Put the rules in the upper right quadrant.
  - Enter actions for each rule
    - In the lower right quadrant, determine what, if any, appropriate actions should be taken for each rule.
  - Reduce table as necessary.

Example

- Calculate the total cost of your tuition this quarter.
  - What do you need to know?
    - Level. (Undergrad or graduate)
    - School. (CTI, Law, etc.)
    - Status. (Full or part time)
    - Number of hours
  - Actions?

- Actions?
  - Consider CTI only (to make the problem smaller):
    - U/G
      - Part Time (1 to 11 hrs.): $335.00/per hour
      - Full Time (12 to 18 hrs.): $17,820.00
      - * Credit hours over 18 are charged at the part-time rate
    - Graduate:
      - Part time (1 to 7 hrs.): $520.00/per hour
      - Full time (>= 8 hrs.): $520.00/per hour
- Create a decision table for this problem. In my solution I was able to reduce the number of rules from 16 to 4.

Boolean Algebra

A *Boolean algebra* consists of:

- a set $B=\{0, 1\}$,
- 2 binary operations on B (denoted by $+$ & $\times$),
- a unary operation on B (denoted by $'$), such that :

$0 + 0 = 0$	$0 \times 0 = 0$
$1 + 0 = 1$	$0 \times 1 = 0$
$0 + 1 = 1$	$1 \times 0 = 0$
$1 + 1 = 1$	$1 \times 1 = 1$

$0'=1$ and $1'=0$.

Rules of a Boolean Algebra

The following axioms ('rules') are satisfied for all elements x, y& z of B:

(1) $x + y = y + x$ (commutative axioms)

$x \times y = y \times x$

(2) $x + (y + z) = (x + y) + z$ (associative axioms)

$x \times (y \times z) = (x \times y) \times z$

(3) $x \times (y + z) = (x \times y) + (x \times z)$

$x + (y \times z) = (x + y) \times (x + z)$ (distributive axioms)

(4) $x + 0 = x$ $x \times 1 = x$ (identity axioms)

(5) $x + x' = 1$ $x \times x' = 0$ (inverse axioms)

Laws of Boolean Algebra

- In addition to the laws given by the axioms of Boolean Algebra, we can show the following laws

$$x'' = x \qquad \text{(double complement)}$$

$$x + x = x \qquad x \times x = x \qquad \text{(idempotent)}$$

$$(x + y)' = x' \times y' \qquad (x \times y)' = x' + y' \qquad \text{(de Morgan's laws)}$$

$$x + 1 = 1 \qquad x \times 0 = 0 \qquad \text{(annihilation)}$$

$$x + (x \times y) = x \qquad x \times (x + y) = x \qquad \text{(absorption)}$$

$$0' = 1 \qquad 1' = 0 \qquad \text{(complement)}$$

Exercise

Simplify the Boolean expression

$(x' \times y) + (x \times y)$

Solution: $(x' \times y) + (x \times y)$

$= (y \times x') + (y \times x)$ (commutative)

$= y \times (x' + x)$ (distributive)

$= y \times (x + x')$ (commutative)

$= y \times 1$ (inverse)

$= y$ (identity)

Thus $(x' \times y) + (x \times y) = y$

Boolean Notation

- This means that in effect we'll be employing Boolean Algebra notation.
- The truth tables can be rewritten as

x	y	$x+y$	$x \times y$	x'
0	0	0	0	1
0	1	1	0	
1	0	1	0	0
1	1	1	1	

Notational Short-cuts

We will employ short-cuts in notation:

(1) In 'multiplication' we'll omit the symbol $\times$, & write xy for $x \times y$ (just as in ordinary algebra)

(2) The associative law says that

$$x + (y + z) = (x + y) + z$$

So we'll write this as simply $x + y + z$, because the brackets aren't necessary.

Notational Short-cuts

Similarly, write the product of 3 terms as xyz

(3) In ordinary algebra, the expression
$x + y \times z$ means $x + (y \times z)$, because of the convention that *multiplication takes precedence over addition.*

e.g. $x + yz$ means $x + (y \times z)$, and *not* $(x + y) \times z$
Similarly, $ab + cd$ means $(a \times b) + (c \times d)$

Reducing Boolean Expressions

- Is this the smallest possible implementation of this expression? No!

$$G = xyz + xyz' + x'yz$$

- Use Boolean Algebra rules to reduce complexity while preserving functionality.

- Step 1: Use idempotent law (a + a = a). So

$$xyz + xyz' + x'yz = xyz + xyz + xyz' + x'yz$$

Reducing Boolean Expressions

- Step 2: Use distributive law $a(b + c) = ab + ac$. So $xyz + xyz + xyz' + x'yz = xy(z + z') + yz(x + x')$

- Step 3: Use Inverse law $(a + a' = 1)$. So

  $xy(z + z') + yz(x + x') = xy.1 + yz.1$

- Step 4: Use Identity law $(a . 1 = a)$. So

  $xy + yz = xy.1 + yz.1 = xyz + xyz' + x'yz$

Karnaugh maps

- Alternate way of representing Boolean function
  - All rows of truth table represented with a square
  - Each square represents a minterm

x \ y	0	1
0	x'y'	x'y
1	xy'	xy

x \ y	0	1
0	1	1
1	0	0

x	y	F
0	0	1
0	1	1
1	0	0
1	1	0

Karnaugh maps

- Easy to convert between truth table, K-map, and SOP.
  - Unoptimized form: number of 1's in K-map equals number of minterms (products) in SOP.
  - Optimized form: reduced number of minterms

$$F(x,y) = x'y + x'y' = x'$$

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.

A \ B	0	1
0	0	1
1	1	0

$F = AB' + A'B$

A \ B	0	1
0	0	1
1	1	1

$F = AB + A'B + AB'$

- Three variable maps.

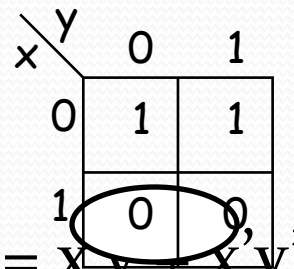
A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

Rules for K-Maps

- We can reduce functions by circling 1's in the K-map.
- Each circle represents minterm reduction.
- Following circling, we can deduce minimized and-or form.



$F(x,y) = x'y' + x'y' = x'$

Rules for K-Maps

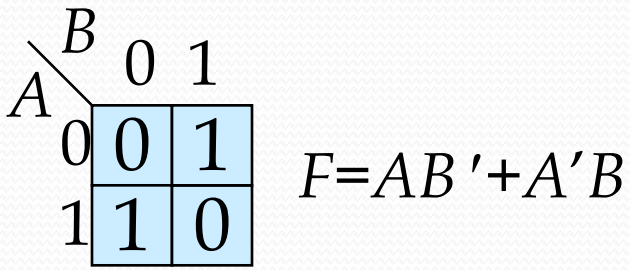
Rules to consider

1. Every cell containing a 1 must be included at least once.

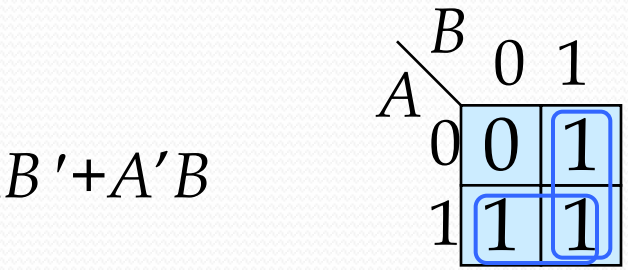
2. The largest possible “power of 2 rectangle” must be enclosed.

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.



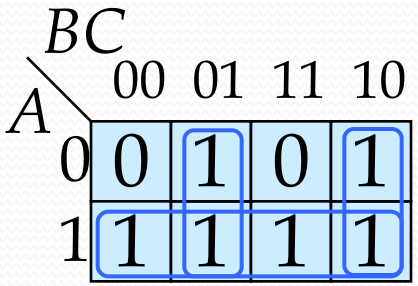
$F = AB' + A'B$



$F = AB + A'B + AB'$

$F = A + B$

- Three variable maps.

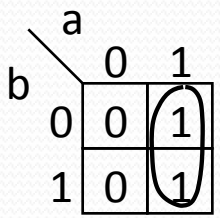


$F = A + B'C + BC'$

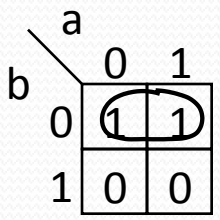
$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

More Karnaugh Map Examples

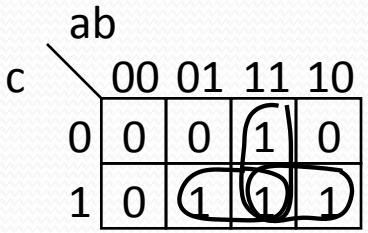
Examples



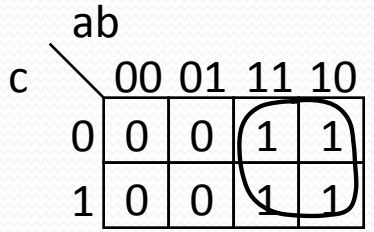
f = a



g = b'



cout = ab + bc + ac



f = a

1. Circle the largest groups possible.
2. Group dimensions must be a power of 2.
3. Remember what circling means

Specifications

Specification validation procedure / steps

1. Rewrite the specifications using consistent terminology.

2. Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.

3. Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it may seem.

4. Convert the rewritten specifications into an equivalent set of Boolean expressions.

5. Identify the default action and cases, if any are specified.

Specifications Continue

6. Enter the Boolean expressions in a KV chart and check for consistency. If
the specifications are consistent, there will be no overlaps, except for the
cases that result in multiple actions.
7. Enter the default cases, and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the
corresponding boxes of the KV chart back into English and get a
clarification, explanation, or revision.
10. If the default cases were not specified explicitly, translate the default cases
back into English and get a confirmation.

UNIT-V

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

Graph Matrices and Application: Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools.

OVER VIEW

- A state graph and its associated state table are useful models for describing software (program) behavior. The finite state machine can be used as a functional (behavioral) testing tool as well as a tool for designing a testable program.

state graph

- A **state graph** is a graphical representation of the program **(its FSM)** in terms of states, transitions, inputs and outputs (erroneous or normal). It has one start state and usually, an end/destination/exit state.
- Note => In the exam you may draw only 3 state graph for simplicity.
- State graph in the above example is used to model the behavior of the program that recognizes a string occurrence at the input. It can be used to design, implement and the testing of the program

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

State table

Very big state graphs are difficult to follow as the diagrams get complicated and links get entwined. It is more convenient to represent the state graph as a table called state table or state transition table.

Each row represents the transitions from the originating state. There is one column for each input symbol (erroneous input or normal input). The entry in the table represents the new state to which the system transits to on this transition and the output it prints on the target printer device or on the output side.

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

Software implementation of state table

- There are four tables that are needed.

1. A table or a process that encodes the input values into a compact list (INPUT_CODE_TABLE)
2. A table that specifies the next state for every combination of state and input code. (TRANISITION_TABLE)
3. A table or case statement that specifies the output (or output code) associated with every state-input combination (OUTPUT_TABLE)
4. A table that stores the present state of each device or process or component or system that uses the same state table. (DEVICE_TABLE)

Software implementation of state table

The process of usage of state table is as follows:

1. the present state is fetched from the memory (from DEVICE_TABLE).
2. the present input value (symbol) is fetched from the environment. It is encoded if it is non-numerical by using the INPUT_CODE_TABLE.
3. The present state and the input code are concatenated to give a pointer (row,column) into a cell of the TRANSITION_TABLE.
4. The OUTPUT_TABLE contains a pointer to the routine to be executed when that state-input combination occurs.
5. The same pointer value is used to fetch the new state value, which is then stored in DEVICE_TABLE

Principles of state testing

As it is not possible to test every path thru a state graph, use the notion of coverage. We assume that the graph is strongly connected.

1. It is not useful or practical to plan an entire grand tour of the states for testing initially as it does not work out due to possibility of bugs.
2. During the maintenance phase only few transitions and states need to be tested which are affected.
3 For very long test input symbol sequences it is difficult to test the system.

Uses/Advantages of state testing

- State testing can find bugs which are not possible to be found with other types of testing. Normally most of systems can be modeled as state graphs.
- It can find if the specifications are complete and ambiguous. This is seen clearly if the state table is filled with multiple entries in some cells or some cells are empty. IT can also tell if some default transitions or transitions on erroneous inputs are missing.
- State testing can identify the system's seemingly impossible states and checks if there are transitions from these states to other states are defined in the specifications or not. That is, the error recovery processes are defined for such impossible states.
-

Uses/Advantages of state testing

- State testing can simplify the design of the program / system by identifying some equivalent states and then merging these states. Also, state testing using FSM can allow design/test design in a hierarchical manner if the state tables are so designed.
- The state testing can identify if the system reaches a dead state / unreachable states and allow one to correct the program specifications and make the system complete, robust and consistent.
- The bugs in the functional behavior can be caught earlier and will be less expensive if state testing is done earlier than the structural (white box) testing.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

Application areas for state testing using FSM model

Any program that processes input as a sequence of events/symbols and produces output such as detection of specified input symbol combinations, sequential format verification, parsing, etc. (compilers & translators).

- Communication Protocols: processing depends on current state of the
- protocol stack, OS, network environment and the message received
- concurrent systems,
- system failures and the corresponding recovery systems,
- distributed data bases,
- device drivers – processing depends on the state of the device &

Application areas for state testing using FSM model

- operation requested by the user or system
  - multi-tasking systems,
  - human computer interactive systems,
  - resource management systems – processing depends on availability
- levels and states of resources
  - Processing of hierarchical pop-up menus on windows based software systems – letting the user navigate thru menus
  - the web based application software, embedded systems and other systems also use this model for design and testing.

Good-state graphs and Bad state graph

The principles of judging whether a state graph is good or bad are:

- the total number of states is equal to the product of possibilities of factors that make up the state. (ie., number of permutations of all values of all attributes/properties of the system/component)
- For every state and every input there is exactly one transition specified to exactly one, possibly the same, state.
- For every transition there is one output action specified. That output could be trivial (epsilon), but at least one output does some thing sensible.
- For every state there is a sequence of inputs that drives the system to the starting (same) state.
- A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.
- Bad state graphs contain states not reachable. It is not possible to reach every state from every other state. It is not possible to reach start state from itself.

Good-state graphs and Bad state graph

- A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.
- Bad state graphs contain states not reachable. It is not possible to reach every state from every other state. It is not possible to reach start state from itself.

OVER·VIEW

- The graphs (flow/directed) are basically behavioral representation of software program. Many problems/applications we want to have solution for :

- – to find set of covering paths, set of values that will sensitize the paths, logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether the state is reachable or not etc.. -

- usually necessitate path tracing thru the graph. Most of the testing and design strategies use graphs to represent the program and need some coverage to be obtained.

- Path tracing thru pictorial graphs and by hand is error prone, difficult to handle and often confusing when the program is large.

- Graph representation by a matrix enables matrix operations equivalent to path tracing to be used. These matrix operations are methodical and mechanical and hence are more reliable. Also, these can be implemented by software tools & automated enabling a good solution for the questions/issues that interest a designer and a tester.

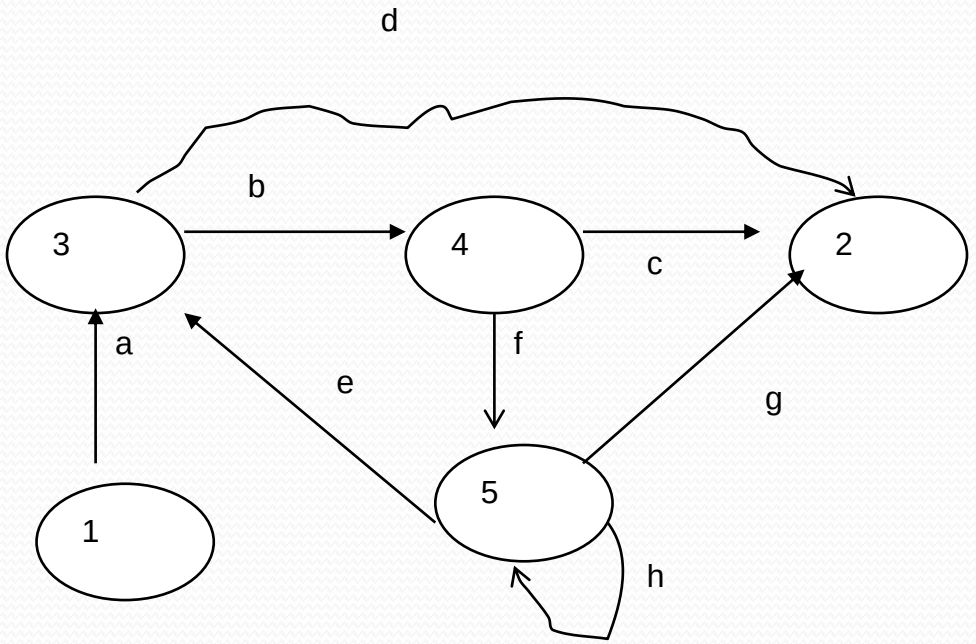


Figure **A**

MATRIX REPRESENTATION:OVER VIEW

- A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination represents a relation between the node corresponding to the row i and the node corresponding to the column j.

- The relation could be just the link name if there is a direct link between the two nodes. The matrix is a square matrix of size n (where n is the # of nodes). Self loops are shown as diagonal entries.

- Actually, an element **A [i , j]** is a path expr between node i to j (sum of parallel path products between the nodes.)

RELATION MATRIX

A matrix representation defined as just above.

Corresponding relation Matrix is

node	1	2	3	4	5
1	0	0	a	0	0
2	0	0	0	0	0
3	0	d	0	b	0
4	0	c	0	0	f
5	0	g	e	0	h

CONNECTION MATRIX

The relation in a connection matrix is defined as follows:

A [i, j] = 1 if there is a direct link from i to j.
0 otherwise

It is the relation matrix with a non zero entry replaced by **1**.

Corresponding connection Matrix is

node	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	0	0
3	0	1	0	1	0
4	0	1	0	0	1
5	0	1	1	0	1

Maths / Algebra over the relation **connection**:

1 X 1 = 1 0 X 0 = 0 1 X 0 = 0 0 X 1 = 0

A [i,j] x A [j,k] represents existence of a (directed) path from i to k

Each entry in the row **r** represents an outgoing link from the node **r**. Each entry in the column **c** represents an incoming link into the node **c**. A branch node **r** has >1 non zero entries in its row **r**. A junction node c has > 1 non-zero entries in its column.

CYCLOMATIC COMPLEXITY

Is 1 plus the number of binary decisions that are present in a given program / graph. The use of the cyclomatic complexity is in estimation of efforts needed for design & testing of a given software program.

calculated as follows

Prepare the relation matrix and from that prepare the connection matrix. For each row (represents branches ie., decisions) obtain the sum of the entries (each entry is 1 or 0). Number of binary decisions at this node / row r is equal to the sum - 1 as number of binary decisions at a node = number of outgoing branches - 1.

Sum the number of binary decisions of each row and add 1 to get the cycl. Complexity.

EXAMPLE:

First row : total = 1 then minus 1 => 0

2nd row	0	nil	0
3rd row	2	1	1
4th row	2	1	1
5th row	3	1	2

		Total	4

example:

cyclomatic complexity = 4+1 = 5

$$A[i, j] = \sum_{K=1 \text{ to } n} a_{ik}\ a_{kk}\ a_{kj}$$

RELATIONSHIPS

A relation is a property that exists between two elements / objects of interest. For example, a R b could denote the relation R as existence of a directed link from a to b.

Redefinitions of graphs:

A graph consists of a set of abstract objects called nodes and a (binary) relation R between (among pairs of) the nodes. If a R b, it means that node a has the relation to node b – which denoted by a link from a to b.

PROPERTIES OF RELATIONSHIPS

1. A relation R is **transitive i**f a R b and b R c imply a R c. Examples of transitive relations are **is connected to** , >= etc.
2. Examples for **an Intransitive relation** are is acquainted with, is a neighbor of, not equal to etc..
3. A relation R is reflexive if for every a, a R a. It is equivalent to a self loop at every node a in the graph.
4. Examples of **irreflexive relation** are not equals, is under etc.
5. A relation is **a Symmetric relation** if for every a and b , a R b implies b R a. It means that if there is a link from a to b then there is a link between b to a. Then it is as good as an undirected graph (unless the link weights are different for the different directions).
6. Examples of symmetric relation are : is a relative of, is a friend of, equals , not equals etc..
7. A relation R is **Anti-symmetric** relation if for every a and b, if **a R b** and **b R a** imply that a = b. That is there is no symmetry between any two different elements.
8. **An equivalence relation** is a relation that satisfies the reflexive, transitive and symmetric properties. Example : is equal to.

POWERS OF GRAPH MATRIX

A (connection or relation) graph matrix product represents existence of a path or its path expression respectively between two nodes with a common intermediate node.

$C = A \times B \qquad C[i, j] \text{ or } c_{ii} = \sum_{K=1 \text{ to } n} \mathbf{a}_{ik}\, \mathbf{b}_{kj}$

- $\sum \mathbf{a}_{ik}\, \mathbf{b}_{kj}$ implies set (sum) of all paths between I and j thru all intermediate nodes k in the graph.

A^2 represents the set of all path segments of length two (links) ie., each non-zero element A [i , j] is the path expression for the set of 2 link long path segments from node i to j.

A^{n-1} represents the set of all path segments of length n-1 (links) ie., each non-zero element A [i , j] is the path expression for the set of n-1 link long path segments from node i to j.

Graph matrix product is not necessarily commutative, but is associative and distributive. The non-zero elements along principal diagonal represent self-loops on those nodes

PARTITIONING ALGORITHM – GROUPING LOOPS AND OBTAINING LOOP-FREE GRAPH

We will consider a graph (over a transitive relation). The graph may have loops. We would like to partition the graph by grouping the nodes in such a way that every node is contained with in one group of another. Such a graph is partially ordered.

The main use is to embed the loops into subroutines to have a resulting loop-free flow in the program/graph. This enables a better design and analysis and testing design.

Here we use the **connection matrix**. To do grouping of loops we just need to calculate intersection of the transitive closure of graph matrix with itself.

Ie., find (A + **I**) n # (A + **I**) nT **Remember this**

If n = 5 then calculate A and its square, then its square and multiply with A. Then obtain its transpose and then do intersection. Here, the intersection means binary AND directly element a [i, j I by element a [i,j].

Once we calculate the intersection as above, the rows or columns which are identical will form a group. Replace them by one row/column. Now redraw the graph with reduced number of nodes. Each group is a merger of the loop-forming node set. The result will be a loop-free directed graph which is partially ordered.

Example: PARTITIONING ALGORITHM

Take an example with just 3 nodes only.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad A+I = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad (A+I)2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad (A+I)3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Now find (A+I)3 Transpose $= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ Intersection result $= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$

As you see columns 2,3 and rows 2,3 are identical. Merge the two nodes.
You have now the following with node 2,3 engulfing & enclosing the loop.

NODE REDUCTION ALGORITHM

- The advantage of this node reduction algorithm using graph matrices is that we do not need to redraw the graph after reduction of a node. Calculations are done methodically and can be automated. It is actually matrix equivalent of the node reduction algorithm in the chapter 'paths, path expressions & regular expr..'.

- **Steps are :**

1. select a node for removal Other than start or end node. Replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms and simplify as much as you can.
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4. the result is a matrix whose size has been reduced by 1. Continue until the two nodes **entry** and **exit** exist.

THE ALGORITHM:

Step 1 : Before step1, Always first eliminate the self nodes ie., removal of a non-zero diagonal entry **d**. Multiply each path expr. in the row r with d* where d is a non-zero diagonal element in row/column r. That is, multiply each outgoing link from node r with d* . Now replace d with o.

Step 2: Select the last column **c**. Take one non-zero element A [r, c] in column c. Pre-Multiply each entry A [c, j] in the (last) row **c** with A[r, c]. Add the result to the element in the intersection of c and j , ie., to A [r, j]. Repeat this for every non-zero element in the last column. Now, remove the last row & column.

Step3: actually amounts to multiplying each incoming link into a node with each outgoing link from that node and then replacing those two links by a direct path bypassing this last node. In the above A [r, c] is an incoming link into node c and A [c, j] is an outgoing link from node c. Multiply and add the result to path expr. from node r to node j ie., A [r, j].

Application Of Node Reduction Algorithm

a. maximum number of paths – to find

use the relation matrix with the entry being the number of paths between a & b. When you multiply and add, use the arithmetic rules defined for this property.

b. Probability of getting there

Use the probability rules for the product and sum. Use the relation matrix with the entries being the weights (probability of getting to the neighboring node from a node).

c. Get/ Return or Push/Pop problem

Use the same arithmetic rules for G & R as defined earlier in path expressions, regular expressions chapter.

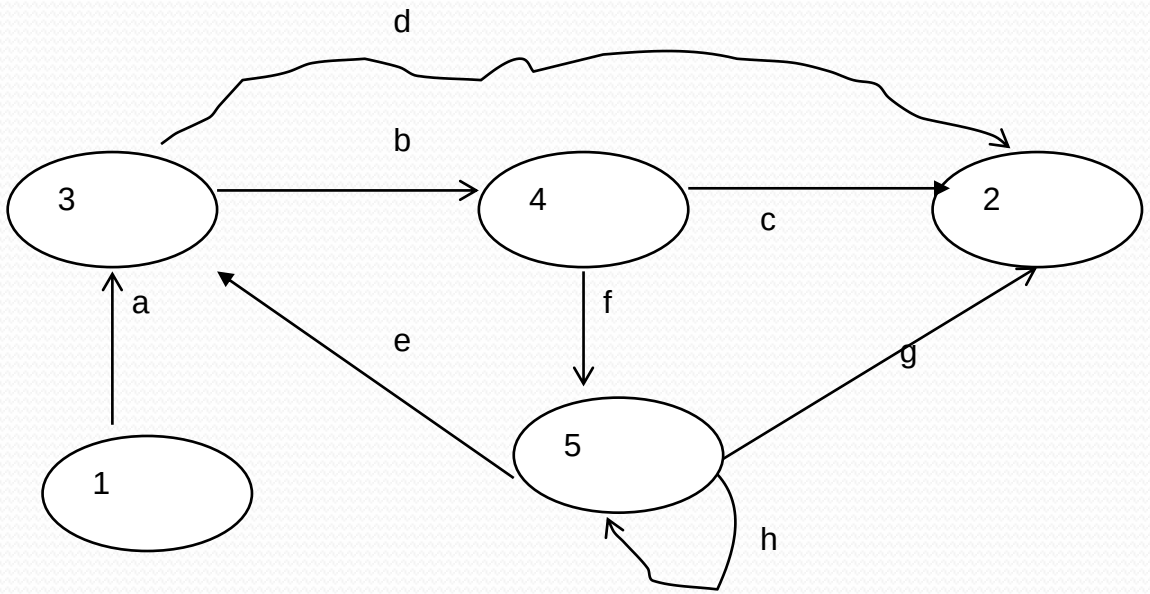
Building software tools using graph matrices for implementation of node reduction algorithm / partitioning algorithm

1. **representation of graph matrix – array / linked list**
2. **issues, merits & demerits of the above representations.**

The degree of a node is the sum of out-degree and in-degree. Usually the degree of a node in a good program is around 3 to 4. We use arrays or linked lists. There are advantages of representation by linked lists over arrays. These are :

1. space grows as O(n2). For a linked list it is only k * n where k is avg degree of a node which is 3 to 4.
2. Weights are complicated for the links and have several components. So would require an additional array for link weights.
3. Variable length weights – as the weights can be expressions or numbers or others, we need a 2-d string array representation for link weight array. Most of the entries are null as graph matrix is sparse usually.
4. Processing time – As each array element is processed whether null or not, the processing time for the null elements could be significant as there are many of them usually in a graph matrix.

EXAMPLE



EXAMPLE-CONTINUE…

Linked list representation

linked List entry (one node in the listed list)	**its content**
node 1,3; a	
2	node 2, exit
3	node 3, 2 ; d
	3, 4 ; b
4	node 4, 2 ; c
	4, 5 ; f
5	node 5, 2 ; g
	5 ,3 ; e
	5 ,5 ; h

Corresponding relation Matrix is

node	1	2	3	4	5
1	0	0	a	0	0
2	0	0	0	0	0
3	0	d	0	b	0
4	0	c	0	0	f
5	0	g	e	0	h

EXAMPLE-CONTINUE…

Linked list representation

linked List entry (one node in the listed list)	**its content**
node 1,3; a	
2	node 2, exit
3	node 3, 2 ; d
	3, 4 ; b
4	node 4, 2 ; c
	4, 5 ; f
5	node 5, 2 ; g
	5 ,3 ; e
	5 ,5 ; h

Corresponding relation Matrix is

node	1	2	3	4	5
1	0	0	a	0	0
2	0	0	0	0	0
3	0	d	0	b	0
4	0	c	0	0	f
5	0	g	e	0	h