UNIT II

TEST CASE DESIGN

Test case Design Strategies – Using Black Box Approach to Test Case Design – Random Testing – Requirements based testing – Boundary Value Analysis – Equivalence Class Partitioning – State based testing – Cause-effect graphing – Compatibility testing – user documentation testing – domain testing – Using White Box Approach to Test design – Test Adequacy Criteria – static testing vs. structural testing – code functional testing – Coverage and Control Flow Graphs – Covering Code Logic – Paths – code complexity testing – Evaluating Test Adequacy Criteria.

The Smart Tester

Responsibility of the tester is to

Design tests that

(i) reveal defects, and

(ii) can be used to evaluate software performance, usability, and reliability

To achieve these goals, testers must select a finite number of test cases, often from a very large execution domain

The smart tester must plan for

- testing,
- select the test cases, and
- monitor the process to insure that the resources and time allocated for the job are utilized effectively

Novice testers, taking their responsibilities seriously, might try to test a module or component using all possible inputs and exercise all possible software structures. An informed and educated tester knows that is not a realistic or economically feasible goal.

Goal of the smart tester

To **understand** the functionality, input/output domain, and the environment of use for the code being tested. For certain types of testing, the tester must also understand in detail how the code is constructed. Finally, a smart tester needs to use knowledge of the types of defects that are commonly injected during development or maintenance of this type of software.

Using this information, the smart tester must then intelligently **select a subset of test inputs** as well as combinations of test inputs that she believes have the greatest possibility of revealing defects within the conditions and constraints placed on the testing process. This takes time and effort, and the tester must chose carefully to maximize use of resources

Test Case Design Strategies

A smart tester who wants to maximize use of time and resources needs to develop effective test cases for execution-based testing

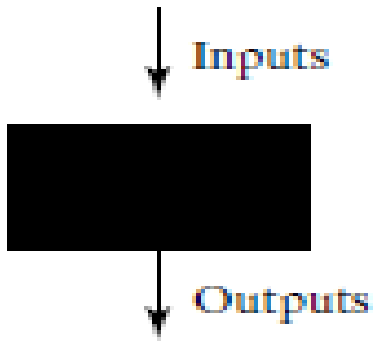
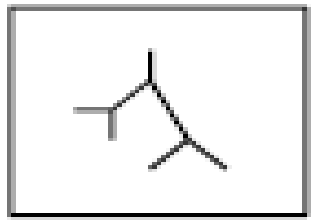
An effective test case has a good possibility of revealing a defect

Positive consequences of an effective test case

- a greater probability of detecting defects,
- a more efficient use of organizational resources,
- a higher probability for test reuse,
- closer adherence to testing and project schedules and budgets, and,
- the possibility for delivery of a higher-quality software product

Approaches used by a tester to design effective test cases

- black box (sometimes called functional or specification) approach and
- white box (sometimes called clear or glass box) approach

Test Strategy	Tester's View	Knowledge Sources	Methods
Black box		Requirements document Specifications Domain knowledge Defect analysis data	Equivalence class partitioning Boundary value analysis State transition testing Cause and effect graphing Error guessing
White box		High-level design Detailed design Control flow graphs Cyclomatic complexity	Statement testing Branch testing Path testing Data flow testing Mutation testing Loop testing

Black box Approach

In this approach, a tester considers the software-under test to be an opaque box

- There is no knowledge of its inner structure (i.e., how it works).
- The tester only has knowledge of what it does.
- The size of the software-under-test using this approach can vary from a simple module, member function, or object cluster to a subsystem or a complete software system.
- The **description of behavior** or **functionality** for the software-under-test may come from a formal specification, an Input/Process/ Output Diagram (IPO), or a well-defined set of pre and post conditions.
- Another source for information is a requirements specification document that usually describes the functionality of the software-under-test and its inputs and expected outputs.
- The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification.

Since the black box approach only considers software behavior and functionality, it is often called **functional**, or **specification-based testing**

This approach is especially useful for **revealing** requirements and specification defects

White box testing

The white box approach focuses on the inner structure of the software to be tested.

- To design test cases the tester must have a knowledge of that structure.
- The **code**, or a suitable **pseudo code** like representation must be available.
- The tester selects test cases to exercise specific internal structural elements to determine if they are working properly.
- Test cases are often designed to exercise all statements or true/false branches that occur in a module or member function.
- Designing, Executing, and Analyzing the results of white box testing is very **time consuming**, so this strategy is usually applied to smaller-sized pieces of software such as a module or member function

This approach is useful for **reveal**ing design and code-based control, logic and sequence defects, initialization defects, and data flow defects

The smart tester knows that to achieve the goal of providing users with low-defect, high-quality software, *both* of these strategies should be used to design test cases

Using the Black Box Approach to Test Case Design

➢ Random Testing

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing.

Example:

If the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen

Issues:

- Are the three values adequate to show that the module meets its specification when the tests are run? Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects?
- Should any values outside the valid domain be used as test inputs?

Use of random test inputs may **save** some of the time and effort that more thoughtful test input selection methods require.

According to many **testing experts**, selecting test inputs randomly has very little chance of producing an effective set of test data

Random testing can be very useful especially at the system level

➢ **Equivalence Class Partitioning**

- Good approach to select test inputs
- It results in a partitioning of the input domain of the software under-test
- It can also be used to partition the output domain, but this is not a common usage
- The finite number of partitions or equivalence classes **allow** the tester to select a given member of an equivalence class as a representative of that class
- All members of an equivalence class are processed in an equivalent way by the target software
- A test value in a particular class is equivalent to a test value of any other member of that class. (If one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect)

Advantages

1. It eliminates the need for exhaustive testing, which is not feasible
2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect
3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class

Guidelines for selecting input equivalence classes (Glen Myers)

1. ‘‘If an input condition for the software-under-test is specified as a *range* of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range.’’
2. ‘‘If an input condition for the software-under-test is specified as a *number* of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.’’
3. ‘‘If an input condition for the software-under-test is specified as a *set* of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.’’

4. “If an input condition for the software-under-test is specified as a “*must be*” condition, select one valid equivalence class to represent the “must be” condition and one invalid class that does not include the “must be” condition.’’

5. “If the input specification or any other information leads to the belief that an element in an equivalence class is not handled in an identical way by the software-under-test, then the class should be further partitioned into smaller equivalence classes.’’

After the equivalence classes have been identified in this way, **the next step** in test case design is the development of the actual test cases. A good approach includes the following steps.

1. Each equivalence class should be assigned a unique identifier. A simple integer is sufficient
2. Develop test cases for all valid equivalence classes until all have been covered by (included in) a test case. A given test case may cover more than one equivalence class
3. Develop test cases for all invalid equivalence classes until all have been covered individually. This is to insure that one invalid case does not mask the effect of another or prevent the execution of another

Ex.

```
Function square_root
    message (x:real)
    when x >= 0.0
    reply (y:real)
    where y >= 0.0 & approximately (y*y,x)
otherwise reply exception imaginary_square_root
end function
```

FIG. 4.2
A specification of a square root function.

Equivalence Classes for the specification

EC1. The input variable x is real, valid.

EC2. The input variable x is not real, invalid.

EC3. The value of x is greater than 0.0, valid.

EC4. The value of x is less than 0.0, invalid.

Boundary Value Analysis

- The test cases developed based on equivalence class partitioning can be strengthened by use of an another technique called boundary value analysis
- Many defects occur directly on, and above and below, the edges of equivalence classes. Test cases that consider these boundaries on both the input and output spaces as shown are often valuable in revealing defects.

Equivalence class partitioning vs Boundary value analysis

- Equivalence class partitioning directs the tester to select test cases from any element of an equivalence class, boundary value analysis requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases

The rules-of-thumb for boundary value analysis

1. If an input condition for the software-under-test is specified as a range of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range.

2. If an input condition for the software-under-test is specified as a *number* of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum.
3. If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set.

An Example of the Application of Equivalence Class Partitioning and Boundary Value Analysis

We are testing a module that allows a user to enter new widget identifiers into a widget data base. The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters

Conditions that apply to the input:

- it must consist of alphanumeric characters,
- the range for the total number of characters is between 3 and 15, and,
- the first two characters must be letters.

Steps in Designing the test case

- Identify input equivalence classes and give them each an identifier. Then augment these with the results from boundary value analysis.
- Tables will be used to organize and record the findings.
- Label the equivalence classes with an identifier ECxxx, where xxx is an integer whose value is one or greater.
- Each class will also be categorized as valid or invalid for the input domain

For Condition 1 – derive two equivalence classes

- **EC1. Part name is alphanumeric, valid.**
- **EC2. Part name is not alphanumeric, invalid.**

For Condition 2 – derive three equivalence classes

- **EC3. The widget identifier has between 3 and 15 characters, valid.**
- **EC4. The widget identifier has less than 3 characters, invalid.**
- **EC5. The widget identifier has greater than 15 characters, invalid.**

For Condition 3 – derive two equivalence classes

- **EC6. The first 2 characters are letters, valid.**
- **EC7. The first 2 characters are not letters, invalid.**

- The equivalence classes selected may be recorded in the form of a table. By inspecting such a table the tester can confirm that all the conditions and associated valid and invalid equivalence classes have been considered.

Example Equivalence class reporting table

Condition	Valid equivalence classes	Invalid equivalence classes
1	EC1	EC2
2	EC3	EC4, EC5
3	EC6	EC7

- Boundary value analysis is now used to refine the results of equivalence class partitioning. The boundaries to focus on are those in the allowed length for the widget identifier.
- A simple set of abbreviations can be used to represent the bounds groups

BLB—a value just below the lower bound

LB—the value on the lower boundary

ALB—a value just above the lower boundary

BUB—a value just below the upper bound

UB—the value on the upper bound

AUB—a value just above the upper bound

For our example module the values for the bounds groups are:

BLB—2	**BUB**—14
LB—3	**UB**—15
ALB—4	**AUB**—16

- In boundary value analysis, values just above the lower bound (ALB) and just below the upper bound (BUB) were selected. These are both valid cases and may be omitted if the tester does not believe they are necessary.
- The next step in the test case design process is to select a set of actual input values that covers all the equivalence classes and the boundaries. Again a table can be used to organize the results

Test inputs using equivalence class partitioning and boundary value analysis for sample module

Module name: Insert_Widget
Module identifier: AP62-Mod4
Date: January 31, 2000
Tester: Michelle Jordan

Test case identifier	Input values	Valid equivalence classes and bounds covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (typical case)	

Components required for a complete test case

Test inputs are shown in Table along with **test conditions** and **expected outputs**.

Test logs are used to record the actual outputs and conditions when execution is complete. Actual outputs are compared to expected outputs to determine whether the module has passed or failed the test

By inspecting the completed table the tester can determine whether all the equivalence classes and boundaries have been covered by actual input test cases

Based on equivalence class partitioning and boundary value analysis these test cases should have a **high possibility of revealing defects** in the module as opposed to selecting test inputs at random from the input domain

Cause – and - Effect Graphing

Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification

Approach:

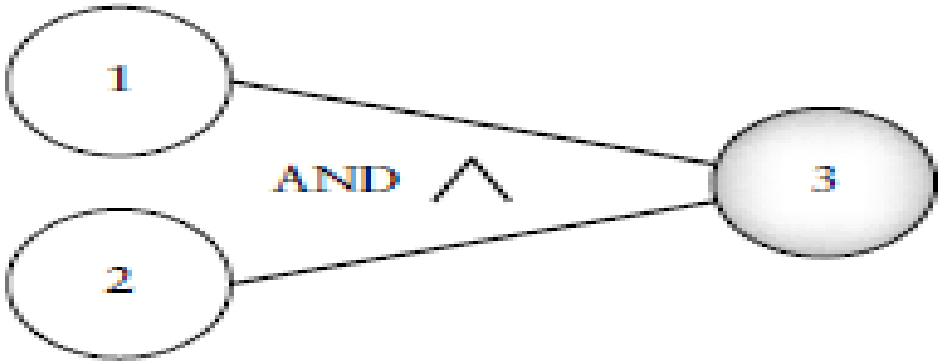
- The specification must be transformed into a graph that resembles a digital logic circuit. The tester should have knowledge of Boolean logic
- The graph itself must be expressed in a graphical language
- The graph must be converted to a decision table that the tester uses to develop test cases.

- Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming.

The steps in developing test cases with a cause-and-effect graph are as follows

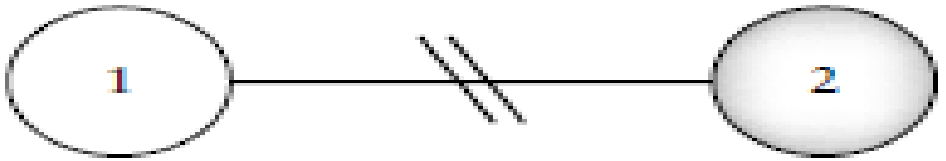
1. The tester must **decompose the specification** of a complex software component into lower-level units.
2. For each specification unit, the tester needs to **identify causes and their effects**. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The **logical relationships** between the causes and effects should be determined. It is useful to express these in the form of a set of rules .
3. From the cause-and-effect information, a Boolean **cause-and-effect graph** is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs.
4. The graph may be **annotated with constraints** that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The **graph** is then **convert**ed to a **decision table**.
6. The **columns** in the decision table are transformed into **test cases**.

Samples of Cause-and-effect Notations



Effect 3 occurs if both causes 1 and 2 are present.

Effect 2 occurs if cause 1 occurs.



Effect 2 occurs if cause 1 does not occur.

Example:

We have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that the user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message “not found” will be output.

The input conditions, or causes are as follows:

- C1: Positive integer from 1 to 80
- C2: Character to search for is in string

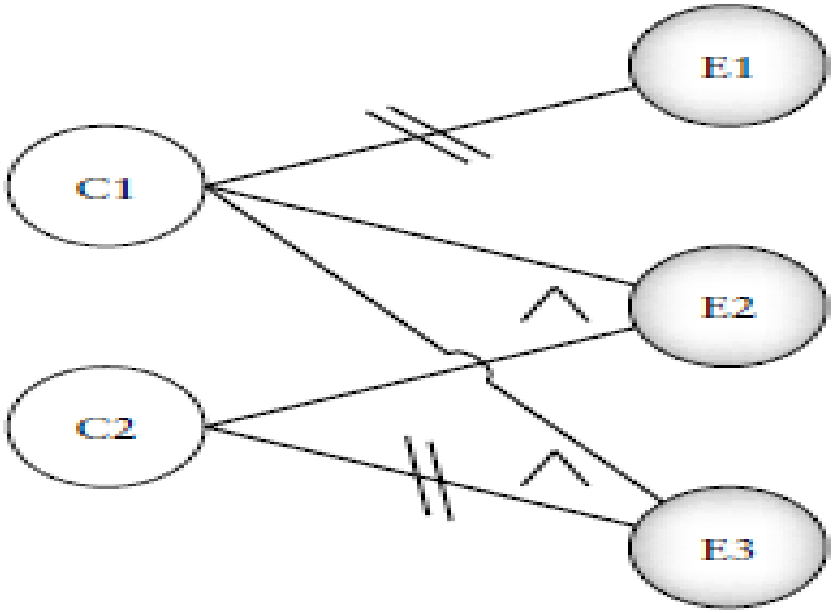
The output conditions, or effects are:

- E1: Integer out of range
- E2: Position of character in string
- E3: Character not found

The rules or relationships can be described as follows:

- If C1 and C2, then E2.
- If C1 and not C2, then E3.
- If not C1, then E1.

Based on the causes, effects, and their relationships, a cause-and-effect graph to represent this information is constructed



- The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes. Columns list each combination of causes, and each column represents a test case. Given *n* causes this could lead to a decision table with 2*n* entries
- A decision table will have a row for each cause and each effect. The entries are a reflection of the rules and the entities in the cause and effect graph. Entries in the table can be represented by a “1” for a cause or effect that is present, a “0” represents the absence of a cause or effect, and a “—” indicates a “don’t care” value

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases

Advantage

- Development of the rules and the graph from the specification allows a thorough inspection of the specification. Any omissions, inaccuracies, or inconsistencies are likely to be detected.
- Exercising combinations of test data that may not be considered using other black box testing techniques.

Major problem

- Developing a graph and decision table when there are many causes and effects to consider

State Transition Testing

Definition : State

A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.

Definition : Finite State Machine

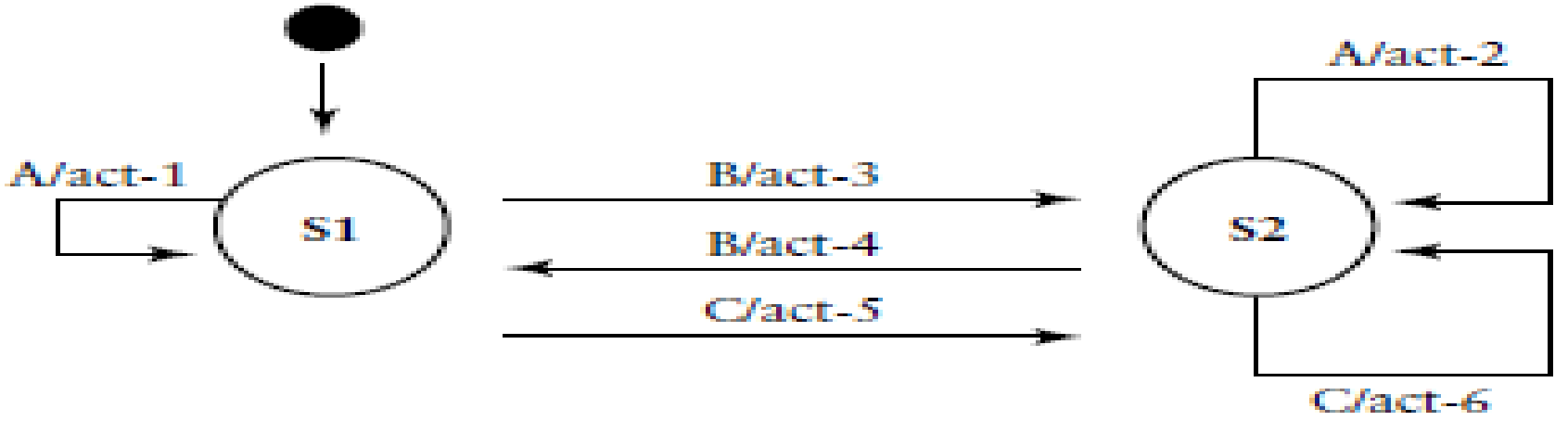
A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

Approach:

It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes

- During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules
- STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states.
- They usually will have a name or number to identify the state.
- A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states.
- Outputs/actions occurring with a state transition are also depicted on a link or arrow

Ex.



- S1 and S2 are the two states of interest. The black dot represents a pointer to the initial state from outside the machine.
- Many STGs also have “error” states and “done” states, the latter to indicate a final state for the system.
- The arrows display inputs/actions that cause the state transformations in the arrow directions.
- For example, the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol “B/act3.”

For large systems and system components, state transition graphs can become very complex. Developers can nest them to represent different levels of abstraction.

A way to simplify the STG is to use a state table representation

State Table:

	S1	S2
Inputs		
Input A	S1 (act-1)	S2 (act-2)
Input B	S2 (act-3)	S1 (act-4)
Input C	S2 (act-5)	S2 (act-6)

- The state table lists the inputs or events that cause state transitions.
- For each state and each input the next state and action taken are listed.
- STGs have been prepared by developers or analysts as a part of the requirements specification.
- The STGs should be subject to a formal inspection when the requirement/specification is reviewed
- From the tester's view point the review should ensure that

  (i) the proper number of states are represented, (ii) each state transition (input/output/action) is correct, (iii) equivalent states are identified, and (iv) unreachable and dead states are identified.

- After the STG has been reviewed formally the tester should plan appropriate test cases
- A simple approach might be to develop tests that insure that all states are entered
- A more practical and systematic approach suggested by Marik consists of testing every possible state transition
- The transition sequence requires the tester to describe the exact inputs for each test as the next step.
- The exact sequence of inputs must also be described, as well as the expected sequence of state changes, and actions.
- Providing these details makes state-based tests easier to execute, interpret, and maintain.
- In addition, it is best to design each test specification so that the test begins in the start state, covers intermediate states, and returns to the start state.
- Finally, while the tests are being executed it is very useful for the tester to have software probes that report the current state (defining a state variable may be necessary) and the incoming event.

Error Guessing:

- It is based on the tester's/developer's past experience with code similar to the code-under test, and their intuition as to where defects may lurk in the code.
- Code similarities may extend to the structure of the code, its domain, the design approach used, its complexity, and other factors.
- The tester/developer is sometimes able to make an educated "guess" as to which types of defects may be present and design test cases to reveal them.

Error guessing is an **ad hoc** approach to test design in most cases. However, if defect data for similar code or past releases of the code has been carefully recorded, the defect types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value

Configuration Testing:

- Configuration testing is the process of checking the operation of the software under testing with all the various types of hardware.

The different configuration possibilities for a standard Windows-based PC

- The PC - Compaq, Dell, Gateway, Hewlett Packard, IBM
- Components - system boards, component cards, and other internal devices such as disk drives, CD-ROM drives, video, sound, modem, and network cards
- Peripherals - printers, scanners, mice, keyboards, monitors, cameras, joysticks
- Interfaces - ISA, PCI, USB, PS/2, RS/232, and Firewire
- Options and memory - hardware options and memory sizes
- Device Drivers

All components and peripherals communicate with the operating system and the software applications through low-level software called *device drivers*. These drivers are often provided by the hardware device manufacturers and are installed when you set up the hardware. Although technically they are software, for testing purposes they are considered part of the hardware configuration.

To start configuration testing on a piece of software, the tester needs to consider which of these configuration areas would be most closely tied to the program.

Examples:

- A highly graphical computer game will require lots of attention to the video and sound areas.
- A greeting card program will be especially vulnerable to printer issues.
- A fax or communications program will need to be tested with numerous modems and network configurations.

Finding Configuration Bugs:

The sure way to tell if a bug is a configuration problem and not just an ordinary bug is to perform the exact same operation that caused the problem, step by step, on another computer with a completely different configuration.

- ✓ If the bug doesn't occur, it's very likely a configuration problem.
- ✓ If the bug happens on more than one configuration, it's probably just a regular bug.

The general process that the tester should use when planning your configuration testing are:

- Decide the types of hardware needed
- Decide what hardware brands, models, and device drivers are available
- Decide which hardware features, modes, and options are possible
- Pare down the identified hardware configurations to a manageable set
- Identify the software’s unique features that work with the hardware configurations
- Design the test cases to run on each configuration
- Execute the tests on each configuration
- Rerun the tests until the results satisfy the test team

Compatibility Testing :

- Software compatibility testing means checking that your software interacts with and shares information correctly with other software.
- This interaction could occur between two programs simultaneously running on the same computer or even on different computers connected through the Internet thousands of miles apart

Examples of compatible software

Cutting text from a Web page and pasting it into a document opened in your word processor

In performing software compatibility testing on a new piece of software, the tester needs to concentrate on

- ✓ Platform and Application Versions (Backward and Forward Compatibility, The Impact of Testing Multiple Versions)
- ✓ Standards and Guidelines (High-Level Standards and Guidelines, Low-Level Standards and Guidelines
- ✓ Data Sharing Compatibility (File save and file load, File export and file import)

Using the White Box Approach to Test Case Design

This is called the white box, or glass box, approach to test case design The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly.

The knowledge needed for the white box test design approach often becomes available to the tester in the later phases of the software lifecycle, specifically during the detailed design phase of development.

This is in contrast to the earlier availability of the knowledge necessary for black box test design. As a consequence, white box test design follows black box design as the test efforts for a given project progress in time.

Another point of contrast between the two approaches is that the black box test design strategy can be used for both small and large software components, whereas white box–based test design is most useful when testing small components.

This is because the level of detail required for test design is very high, and the granularity of the items testers must consider when developing the test data is very small.

Test Adequacy Criteria

The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements to determine if defects exist in the program structure. By exercising all of the selected structural elements the tester hopes to improve the chances for detecting defects.

Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly. Rules of this type can be used to determine whether or not sufficient testing has been carried out.

The application scope of adequacy criteria also includes:

(i) helping testers to select properties of a program to focus on during test;

(ii) helping testers to select a test data set for a program based on the selected properties;

(iii) Supporting testers with the development of quantitative objectives for testing;

(iv) indicating to testers whether or not testing can be stopped for that program.

Using the selected adequacy criterion a tester can terminate testing when he/she has exercised the target structures, and have some confidence that the software will function in a manner acceptable to the user.

If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion. They use either logic and control structures, dataflow, program text, or faults as the focal point of an adequacy evaluation.

Specification-based test data adequacy criteria focus on program specifications. some test data adequacy criteria ignore both program structure and specification in the selection and evaluation of test data.

A **test data set** is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called “coverage

When a coverage related testing goal is expressed as a percent, it is often called the "**degree of coverage**." The planned degree of coverage is specified in the test plan and then measured when the tests are actually executed by a coverage tool.

The planned degree of coverage may be less than 100% possibly due to the following:

• The nature of the unit

—Some statements/branches may not be reachable.

—The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.

• The lack of resources

—The time set aside for testing is not adequate to achieve 100% coverage.

—There are not enough trained testers to achieve complete coverage for all of the units.

—There is a lack of tools to support complete coverage.

• Other project-related issues such as timing, scheduling, and marketing constraints

For example consider that a tester specifies “branches” as a target property for a series of tests. A reasonable testing goal would be satisfaction of the branch adequacy criterion.

the tester must develop a set of test data that insures that all of the branches (true/false conditions) in the unit will be executed at least once by the test cases.

If there are, for example, four branches in the software unit, and only two are executed by the planned set of test cases, then the degree of branch coverage is 50%. All four of the branches must be executed by a test set in order to achieve the planned testing goal.

When a coverage goal is not met, the tester develops additional test cases and re executes the code. This cycle continues until the desired level of coverage is achieved. The greater the degree of coverage, the more adequate the test

An implication of this process is that a higher degrees of coverage will lead to greater numbers of detected defects.

Static Testing vs Structural Testing

- Static Testing, a software testing technique in which the software is tested without executing the code. It has two parts as listed below:
- Review - Typically used to find and eliminate errors or ambiguities in documents such as requirements, design, test cases, etc.
- Static analysis - The code written by developers are analysed (usually by tools) for structural defects that may lead to defects.

Types of Reviews:

The types of reviews can be given by a simple diagram:

Following are the types of defects found by the tools during static analysis:

- A variable with an undefined value
- Inconsistent interface between modules and components
- Variables that are declared but never used
- Unreachable code (or) Dead Code
- Programming standards violations
- Security vulnerabilities
- Syntax violations.

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation. The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.

Structural Testing Techniques:

i. **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.

ii. Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once.

iii. Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch are covered.

Advantages of Structural Testing:

- Forces test developer to reason carefully about implementation
- Reveals errors in "hidden" code
- Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of Structural Box Testing:

- Expensive as one has to spend both time and money to perform white box testing.
- Every possibility that few lines of code is missed accidentally.
- In depth knowledge about the programming language is necessary to perform white box testing.

Coverage and Control Flow Graphs

The application of coverage analysis is typically associated with the use of control and data flow models to represent program structural elements and data. The logic elements most commonly considered for coverage are based on the flow of control in a unit of code. For example,

(i) program statements;

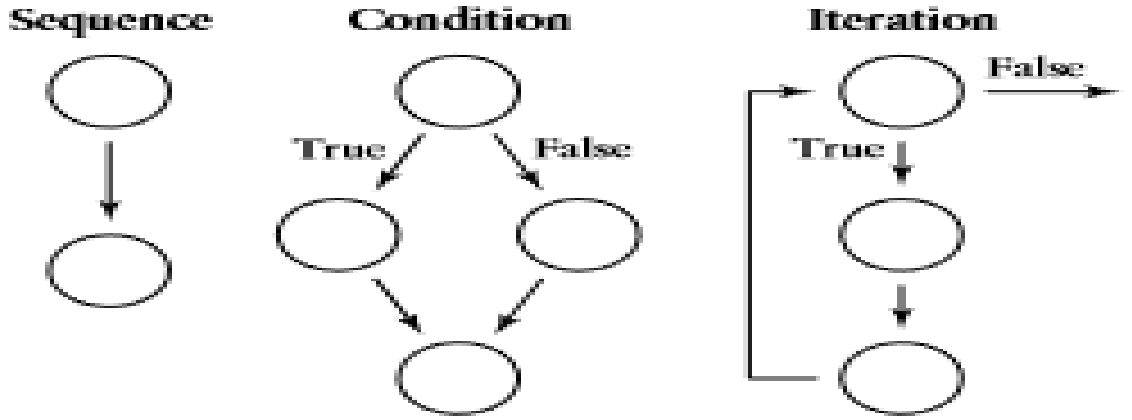
(ii) decisions/branches (these influence the program flow of control);

(iii) conditions(expressions that evaluate to true/false, and do not contain any other true/false-valued expressions);

(iv) combinations of decisions and conditions;

(v) paths (node sequences in flow graphs).

All structured programs can be built from three basic primes-sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops). Graphical representations for these three primes are shown in Figure.



Using the concept of a prime and their combinations a structured code, a (control) flow diagram for the software unit under test can be developed. The flow graph can be used by the tester to evaluate the code with respect to its testability, as well as to develop white box test cases.

In the flow graph the nodes represent sequential statements, as well as decision and looping predicates. If the first statement in the block is executed, so are all the following statements in the block. Edges in the graph represent transfer of control. The direction of the transfer depends on the outcome of the condition in the predicate (true or false).

There are commercial tools that will generate control flow graphs from code which the tester can use for developing control flow graphs especially for complex pieces of code.

A control flow representation for the software under test facilitates the design of white box–based test cases as it clearly shows the logic elements needed to design the test cases using the coverage criterion of choice.

Covering Code Logic

Logic-based white box–based test design and use of test data adequacy/ coverage concepts provide two major payoffs for the tester: (i) quantitative coverage goals can be proposed, and (ii) commercial tool support is readily available to facilitate the tester's work.

For example, if the tester selects the logic element "program statements," this indicates that he/she will want to design tests that focus on the execution of program statements.

If the goal is to satisfy the statement adequacy/ coverage criterion, then the tester should develop a set of test cases so that when the module is executed, all (100%) of the statements in the module are executed at least once.

In terms of a flow graph model of the code, all the nodes in the graph are exercised at least once by the test cases. For the code in fig and its corresponding flow graph shown, a tester would have to develop test cases that exercise nodes 1–8 in the flow graph.

If the tests achieve this goal, the test data would satisfy the statement adequacy criterion. In addition to statements, the other logic structures are also associated with corresponding adequacy/coverage criteria.

For example, to achieve complete (100%) decision (branch) coverage test cases must be designed so that each decision element in the code (if-then, case, loop) executes with all possible outcomes at least once.

In terms of the control flow model, all the edges in the corresponding flow graph must be exercised at least once. Complete decision coverage is considered to be

a stronger coverage goal than statement coverage since its satisfaction results in satisfying statement coverage as well. The statement coverage goal is so weak that it is not considered to be very useful for revealing defects.

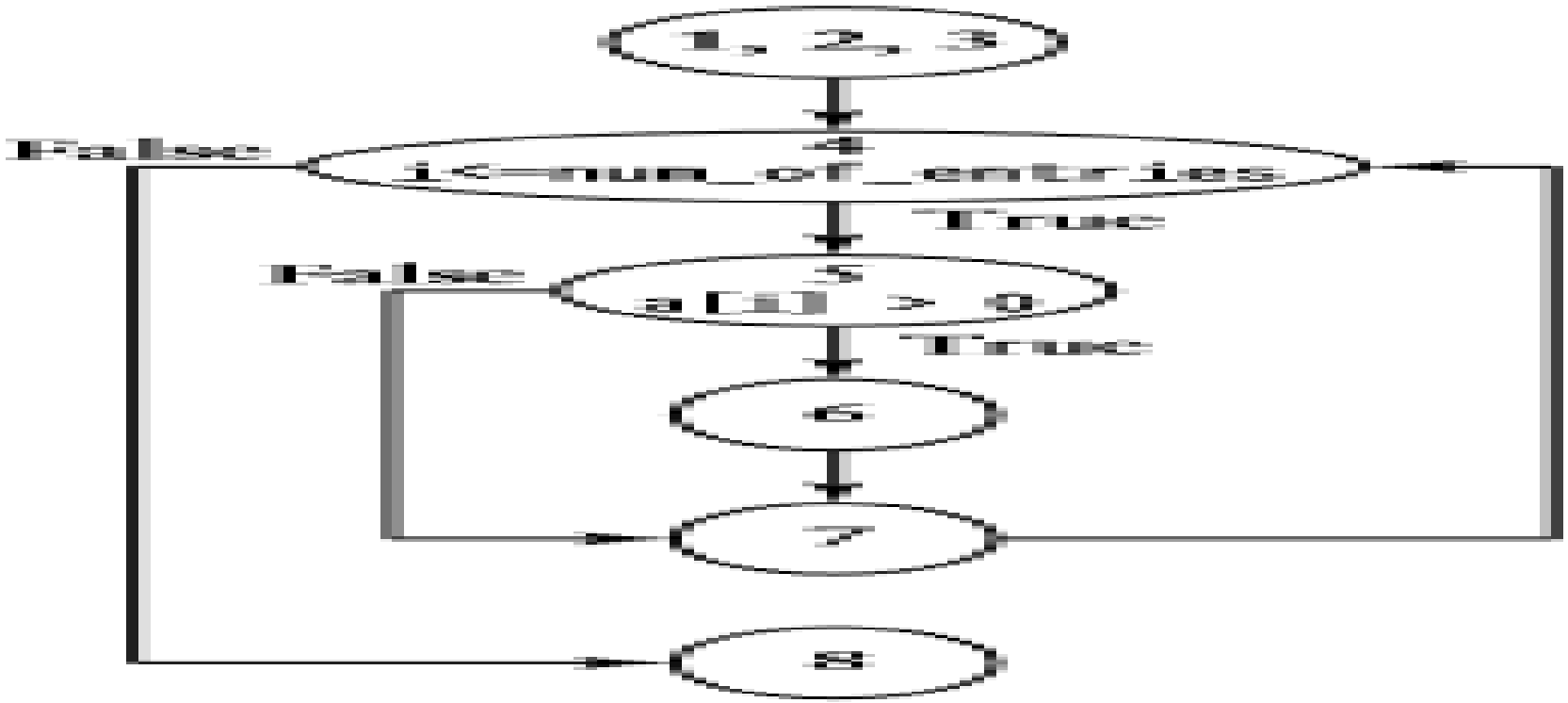
Decision (branch) coverage for the code example in fig, requires test cases to be developed for the two decision statements, that is, the four true/false edges in the control flow graph. Input values must ensure execution the true/false possibilities for the decisions in line4 (while loop) and line5 (if statement).

The “if” statement has a “null else” component, that is, there is no “else” part. However, we include a test that covers both the true and false conditions for the statement.

A possible test case that satisfies 100% decision coverage is shown in table below that satisfies both the branch adequacy criterion and the statement adequacy criterion, since all the statements1–8would be executed by this test case.

```
/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer
   array a. Input parameters are num_of_entries, an integer, and a, an array of integers with
   num_of_entries elements. The output parameter is the integer sume */

1.    pos_sum(a, num_of_entries, sum)
2.          sum = 0
3.          inti = 1
4.          while (i <= num_of_entries)
5.                if a[i] >0
6.                      sum = sum + a[i]
                  endif
7.                i = i+1
            end while
8.    end pos_sum
```



Decision or branch	Value of variable i	Value of predicate	Test case: Value of a, num_of_entries
			a = 1, −45,3 num_of_entries = 3
while	1	True	
	4	False	
if	1	True	
	2	False	

One of the inputs, “a,” is an array, it was possible to assign both positive and negative values to the elements of “a,” thus allowing coverage of both the true/false branches of the “if” statement. Since more than one iteration of the “while” loop was also possible, both the true and false branches of this loop could also be covered by one test case.

Finally, the code in the example does not contain any checks on the validity of the input parameters. For simplicity it is assumed that the calling module does the checking.

In most cases the stronger the coverage criterion, the larger the number of test cases that must be developed to insure complete coverage. For code with multiple decisions and conditions the complexity of test case design increases with the strength of the coverage criterion.

The tester must decide, based on the type of code, reliability requirements, and resources available which criterion to select, since the stronger the criterion selected the more resources are usually required to satisfy it.

Paths: Their Role in White Box–Based Test Design

Tools are available to generate control flow graphs. These tools typically calculate a value for a software attribute called McCabe's Cyclomatic Complexity V(G) from a flow graph.

The complexity value is usually calculated from the control flow graph (G) by the formula

V(G) = E + N 2 ----(1)

The value E is the number of edges in the control flow graph and N is the number of nodes. This formula can be applied to flow graphs where there are no disconnected components. The cyclomatic complexity of the flow graph in Fig above is calculated as follows:

E = 7, N = 6

V(G) = 7 – 6 + 2 = 3

The cyclomatic complexity value of a module is useful to the tester to provide an approximation of the number of test cases needed for branch coverage in a module of structured code.

The cyclomatic complexity value and the control flow graph give the tester another tool for developing white box test cases using the concept of a **path.**

A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node. A path may go through a given segment of the control flow graph one or more times. A path is designated by the sequence of nodes it encompasses. For example, one path from the graph in fig is

1-2-3-4-8

where the dashes represent edges between two nodes. For example, the sequence "4-8" represents the edge between nodes 4 and 8.

Cyclomatic complexity is a measure of the number of so-called "independent" paths in the graph. Deriving a set of independent paths using a flow graph can support a tester in identifying the control flow features in the code and in setting coverage goals.

The independent paths are defined as any new path through the graph that introduces a new edge that has not be traversed before the path is defined.

A set of independent paths for a graph is sometimes called a **basis set**. For the flow graph in fig above, the following set of independent paths starting with the first path identified above can be derived.

(i) 1-2-3-4-8
(ii) 1-2-3-4-5-6-7-4-8
(iii) 1-2-3-4-5-7-4-8

The number of independent paths in a basis set is equal to the cyclomatic complexity of the graph. The cyclomatic complexity for a flow graph also gives an approximation (usually an upper limit) of the number of tests needed to achieve branch (decision) coverage.

If white box test cases are prepared so that the inputs cause the execution of all of these paths, we can be reasonably sure that we have achieved complete statement and decision coverage for the module.

Logic-based testing criterion based on the path concept calls for complete path coverage; that is, every path (as distinguished from independent paths) in a module must be exercised by the test set at least once. This may not be a practical goal for a tester.

For example, even in a small and simple unit of code there may be many paths between the entry and exit nodes. Adding even a few simple decision statements increases the number of paths. Thus, complete path coverage for even a simple module may not be practical, and for large and complex modules it is not feasible.

In addition, some paths in a program may be unachievable, that is, they cannot be executed no matter what combinations of input data are used. The latter makes achieving complete path coverage an impossible task.

Under these circumstances coverage goals are best expressed in terms of the number of feasible or achievable paths, branches, or statements respectively.

Code complexity Testing
Data Flow and White Box Test Design
A variable is defined in a statement when its value is assigned or changed. For example in the statements

Y = 26 * X
Read (Y)

the variable Y is defined, that is, it is assigned a new value. In data flow notation this is indicated as a **def** for the variable Y.

We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.

A predicate use (**p-use**) for a variable indicates its role in a predicate. A computational use(**c-use**) indicates the variable's role as a part of a computation. In both cases the variable value is unchanged. For example, in the statement

Y = 26 * X

the variable X is used. Specifically it has a c-use. In the statement,

if (X >98)

Y = max

X has a predicate or p-use. There are other data flow roles for variables such as undefined or dead. An analysis of data flow patterns for specific variables is often very useful for defect detection.

Testers and developers can utilize data flow tools that will identify and display variable role information. These should also be used prior to code reviews to facilitate the work of the reviewers.

A path from a variable definition to a use is called a **def-use path**.

```
1       sum = 0                                  sum, def
2       read (n),                                n, def
3       i = 1                                    i, def
4       while (i <= n)                           i, n p-use
5               read (number)                    number, def
6               sum = sum + number               sum, def, sum, number, c-use
7               i = i + 1                        i, def, c-use
8       end while
9       print (sum)                              sum, c-use
```

FIG. 5.4

Sample code with data flow information.

To satisfy the all def-use criterion the tester must identify and classify occurrences of all the variables in the software under test. Then for each variable, test data is generated so that all definitions and all uses for all of the variables are exercised during test.

The variables of interest are sum, i, n, and number. Since the goal is to satisfy the all def-use criteria the def-use occurrences for each of these variables needs to tabulated. The data flow role for each variable in each statement of the example is shown beside the statement in italics.

On the table each def-use pair is assigned an identifier. Line numbers are used to show occurrence of the def or use. Note that in some statements a given variable is both defined and used.

Table for n

pair id	def	use
1	2	4

Table for number

pair id	def	use
1	5	6

Table for sum

pair id	def	use
1	1	6
2	1	9
3	6	6
4	6	9

Table for i

pair id	def	use
1	3	4
2	3	7
3	7	7
4	7	4

The tester then generates test data to exercise all of these def-use pairs In many cases a small set of test inputs will cover several or all def-use paths. Here, two sets of test data would cover all the def-use pairs for the variables:

Test data set 1: n = 0

Test data set 2: n = 5, number = 1,2,3,4,5

Set 1 covers pair 1 for n, pair 2 for sum, and pair 1 for i. Set 2 covers pair 1 for n, pair 1 for number, pairs 1,3,4 for sum, and pairs 1,2,3,4 for i. Note even for this small piece of code there are four tables and four def-use pairs for two of the variables.

Loop Testing

Loops are among the most frequently used control structures. Many defects are associated with loop constructs often due to poor programming practices and lack of reviews. Special attention should be paid to loops during testing.

Beizer has classified loops into four categories: simple, nested, concatenated, and unstructured. If instances of unstructured loops are found in legacy code they should be redesigned to reflect structured programming techniques.

Loop testing strategies focus on detecting common defects associated with these structures. For example, in a simple loop that can have a range of zero to n iterations, test cases should be developed so that there are:

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;

(ii) one iteration of the loop;

(iii) two iterations of the loop;

(iv) k iterations of the loop where $k < n$;

(v) n - 1 iterations of the loop;

(vi) n + 1 iterations of the loop (if possible).

Mutation Testing

Mutation testing requires knowledge of code structure, but it is classified as a fault-based testing approach. It considers the possible faults that could occur in a software component as the basis for test data generation and evaluation of testing effectiveness.

Mutation testing makes two major assumptions:

1. The competent programmer hypothesis. This states that a competent programmer writes programs that are nearly correct. Therefore we can assume that there are no major construction errors in the program; the code is correct except for a simple error(s).
2. The coupling effect. This effect relates to questions a tester might have about how well mutation testing can detect complex errors since the changes made to the code are very simple. DeMillo states that test data that can distinguish all programs differing from a correct one only by simple errors are sensitive enough to distinguish it from programs with more complex errors.

Mutation testing starts with a code component, its associated test cases, and the test results. The original code component is modified in a simple way to provide a set of similar components that are called **mutants**.

Each mutant contains a fault as a result of the modification. The original test data is then run with the mutants. If the test data reveals the fault in the mutant (the result of the modification) by producing a different output as a result of execution, then the mutant is said to be killed.

If the mutants do not produce outputs that differ from the original with the test data, then the test data are not capable of revealing such defects. The tests cannot distinguish the original from the mutant.

The tester then must develop additional test data to reveal the fault and kill the mutants.

Evaluating Test Adequacy Criteria

One source of information the tester can use to select an appropriate criterion is the test adequacy criterion hierarchy as shown in fig below which describes a subsumes relationship among the criteria. Satisfying an adequacy criterion at the higher levels of the hierarchy implies a greater thoroughness in testing.

The criteria at the top of the hierarchy are said to subsume those at the lower levels. For example, achieving all definition-use (def-use) path adequacy means the tester has also achieved both branch and statement adequacy.

A set of axioms are presented that allow testers to formalize properties which should be satisfied by any good program-based test data adequacy criterion. Testers can use the axioms to:

i. Recognize both strong and weak adequacy criteria.
ii. Focus attention on the properties that an effective test data adequacy criterion should exhibit;
iii. Select an appropriate criterion for the item under test;
iv. Stimulate thought for the development of new criteria; the axioms are the framework with which to evaluate these new criteria.

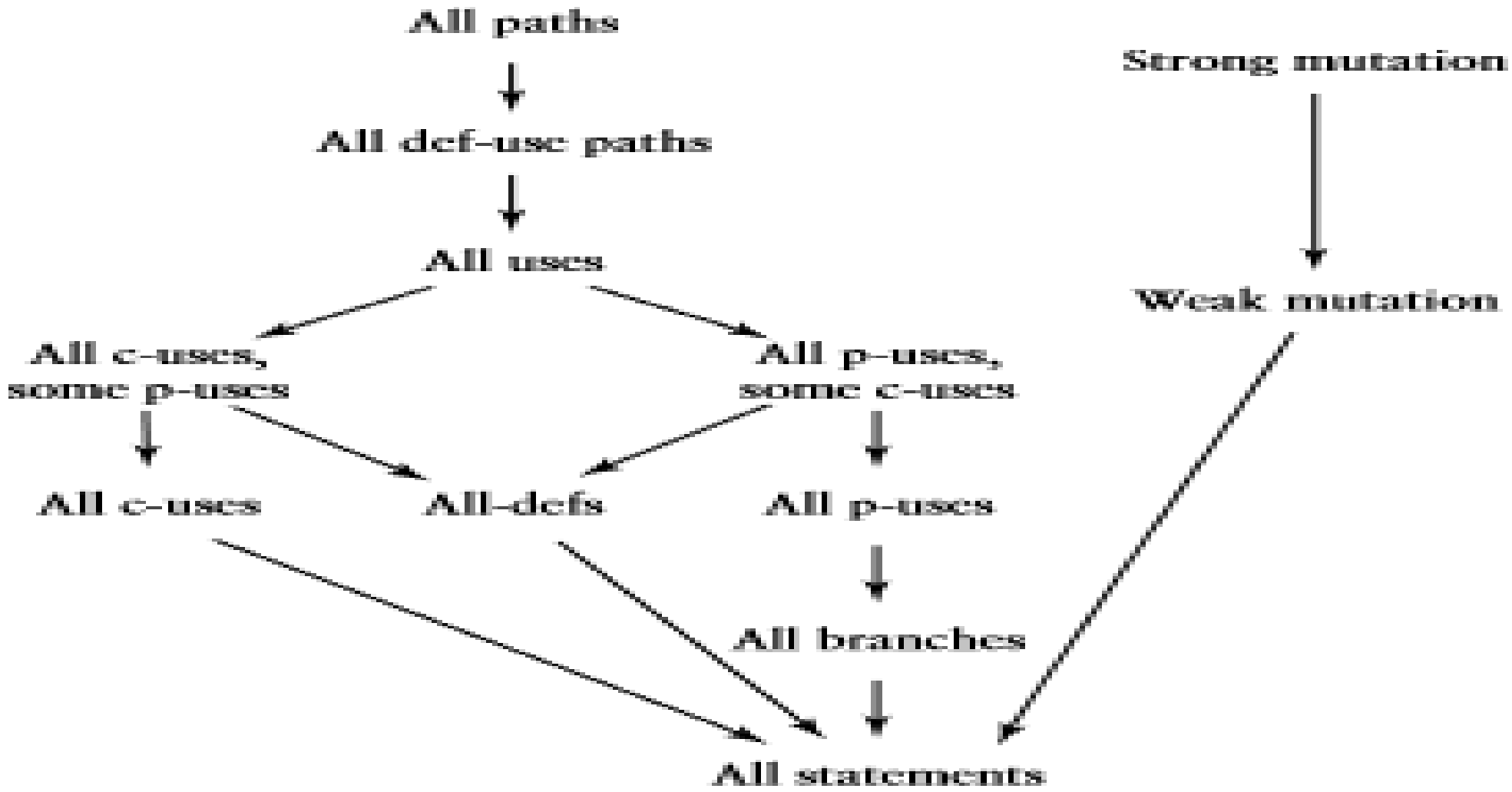


FIG. 5.5

A partial ordering for test adequacy

The axioms are based on the following set of assumptions:

(i) programs are written in a structured programming language;

(ii) programs are SESE (single entry/single exit);

(iii) all input statements appear at the beginning of the program;

(iv) all output statements appear at the end of the program.

The axioms/properties are the following:

1. Applicability Property - for all programs we should be able to design an adequate test set that properly tests it.

2. Non-exhaustive Applicability Property - a tester does not need an exhaustive test set in order to adequately test a program.

3. Monotonicity Property - "If a test set T is adequate for program P, and if T is equal to, or a subset of T, then T is adequate for program P."

4. Inadequate Empty Set – If a program is not tested at all, a tester cannot claim it has been adequately tested.

5. Anti extensionality Property – Just because two programs are semantically equivalent(they may perform the same function) does not mean we should test them the same way.

6. General Multiple Change Property - semantic closeness is not sufficient to imply that two programs should be tested in the same way.

7. Anti decomposition Property - although an encompassing program has been adequately tested, it does not follow that each of its components parts has been properly tested.

8. Anti composition Property - adequately testing each individual program component in isolation does not necessarily mean that we have adequately tested the entire program.

9. Renaming Property - an inessential change in a program such as changing the names of the variables should not change the nature of the test data that are needed to adequately test the program.

10. Complexity Property - for every program, there are other programs that require more testing.

11. Statement Coverage Property - If the test set T is adequate for P, then T causes every executable statement of P to be executed.