R N S INSTITUTE OF TECHNOLOGY

CHANNASANDRA, BANGALORE - 98

SOFTWARE TESTING

NOTES FOR 8TH SEMESTER INFORMATION SCIENCE

SUBJECT CODE: 06IS81

PREPARED BY

DIVYA K

1RN09IS016

8th Semester

Information Science

divya.1rn09is016@gmail.com

NAMRATHA R

1RN09IS028

8th Semester

Information Science

namratha.1rn09is028@gmail.com

SPECIAL THANKS TO

ANANG A – BNMIT & CHETAK M - EWIT

TEXT BOOKS:
FOUNDATIONS OF SOFTWARE TESTING – Aditya P Mathur, Pearson Education, 2008
SOFTWARE TESTING AND ANALYSIS: PROCESS, PRINCIPLES AND TECHNIQUES – Mauro Pezze, Michal Young, John Wiley and Sons, 2008

CONTENTS:

UNIT 1, UNIT 2, UNIT 3, UNIT 5, UNIT 7

UNIT 1
BASICS OF SOFTWARE TESTING - 1

ERRORS AND TESTING

- Humans make errors in their thoughts, in their actions, and in the products that might result from their actions.
- Humans can make errors in an field.
  Ex: observation, in speech, in medical prescription, in surgery, in driving, in sports, in love and similarly even in software development.
- Example:
  - An instructor administers a test to determine how well the students have understood what the instructor wanted to convey
  - A tennis coach administers a test to determine how well the understudy makes a serve

Errors, Faults and Failures
Error: An error occurs in the process of writing a program
Fault: a fault is a manifestation of one or more errors
Failure: A failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to program's output

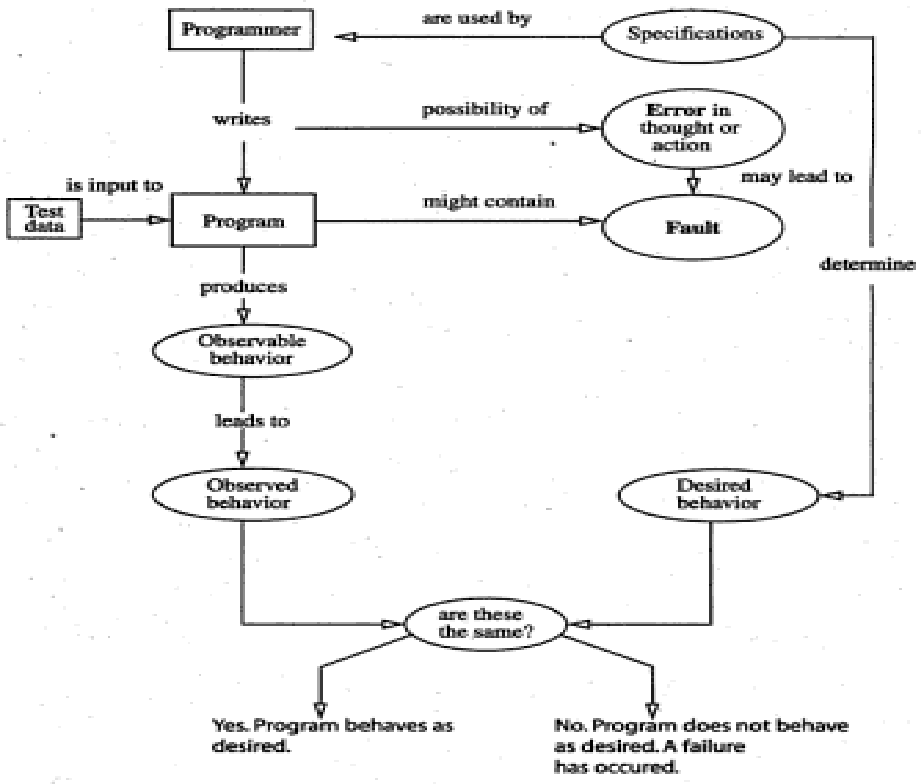


Fig. 1.1 Errors, faults, and failures in the process of programming and testing.

The programmer might misinterpret the requirements and consequently write incorrect code. Upon execution, the program might display behaviour that does not match with the expected behaviour, implying thereby that a failure has occurred.

A fault in the program is also commonly referred to as a bug or a defect. The terms error and a bug or a defect. The terms error and bug are by far the most common ways of referring to something wrong in the program text that might lead to a failure. Faults are sometimes referred to as defects.

In the above diagram notice the separation of observable from observed behaviour. This separation is important because it is the observed behaviour that might lead one to conclude that a program has failed. Sometimes conclusion might be incorrect due to one or more reasons.

Test Automation:

- Testing of complex systems, embedded and otherwise, can be a human intensive task.
- Execution of many tests can be tiring as well as error-prone. Hence, there is a tremendous need for software testing.
- Most software development organizations, automate test-related tasks such as regression testing, graphical user interface testing, and i/o device driver testing.
- The process of test automation cannot be generalized.

  General purpose tools for test automation might not be applicable in all test environments

  Ex:
    - Eggplant
    - Marathon
    - Pounder for GUI testing
    - Load & performance testing tools
        - eloadExpert
        - DBMonster
        - JMeter
        - Dieseltest
        - WAPT
        - LoadRunner
        - Grinder

Regression testing tools:

- Echelon
- Test Tube
- WinRunner
- X test

AETG is an automated test generator that can be used in a variety of applications.
Random Testing is often used for the estimation of reliability of products with respect to specific events.
Tools: DART
Large development organizations develop their own test automation tools due primarily to the unique nature of their test requirements.

Developers and Testers as two Roles:

- Developer is one who writes code & tester is one who tests code. Developer & Tester roles are different and complementary roles. Thus, the same individual could be a developer and a tester. It is hard to imagine an individual who assumes the role of a developer but never that of a tester, and vice versa.
- Certainly, within a software development organization, the primary role of a individual might be to test and hence hs individual assumes the role of a tester. Similarly, the primary role of an individual who designs applications and writes code is that of a developer.

SOFTWARE QUALITY

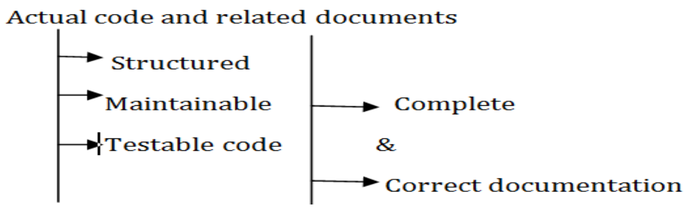
- Software quality is a multidimensional quantity and is measurable.

Quality Attributes

- These can be divided to static and dynamic quality attributes.

Static quality attributes

- It refers to the actual code and related documents.



Example: A poorly documented piece of code will be harder to understand and hence difficult to modify.
A poorly structured code might be harder to modify and difficult to test.

Dynamic quality Attributes:

- Reliability
- Correctness
- Completeness
- Consistency
- Usability
- performance

Reliability:

- It refers to the probability of failure free operation.

Correctness:

- Refers to the correct operation and is always with reference to some artefact.
- For a Tester, correctness is w.r.t to the requirements
- For a user correctness is w.r.t the user manual

Completeness:

- Refers to the availability of all the features listed in the requirements or in the user manual.
- An incomplete software is one that does not fuly implement all features required.

Consistency:

- Refers to adherence to a common set of conventions and assumptions.
- Ex: All buttons in the user interface might follow a common-color coding convention.

Usability:

- Refer to ease with which an application can be used. This is an area in itself and there exist techniques for usability testing.
- Psychology plays an important role in the design of techniques for usability testing.
- Usability testing is a testing done by its potential users.
- The development organization invites a selected set of potential users and asks them to test the product.
- Users in turn test for ease of use, functionality as expected, performance, safety and security.
- Users thus serve as an important source of tests that developers or testers within the organization might not have conceived.
- Usability testing is sometimes referred to as user-centric testing.

Performance:

- Refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement.

Reliability:

- (Software reliability is the probability of failure free operation of software over a given time interval & under given conditions.)
- Software reliability can vary from one operational profile to another. An implication is that one might say "this program is lousy" while another might sing praises for the same program.
- Software reliability is the probability of failure free operation of software in its intended environments.
- The term environment refers to the software and hardware elements needed to execute the application. These elements include the operating system(OS)hardware requirements and any other applications needed for communication.

Requirements, Behaviour and Correctness:

- Product(or) software are designed in response to requirements. (Requirements specify the functions that a product is expected to perform.) During the development of the product, the requirement might have changed from what was stated originally. Regardless of any change, the expected behaviour of the product is determined by the tester's understanding of the requirements during testing.
- Example:
  Requirement 1: It is required to write a program that inputs and outputs the maximum of these.
  Requirement 2: It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.
- Suppose that the program max is developed to satisfy requirement 1 above. The expected output of max when the input integers are 13 and 19 can be easily determined to be 19.
- Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return typed in after each number.
- The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness requirements 1.
- The second requirement in (the above example is ambiguous. It is not clear from this requirement whether the input sequence is to be sorted in ascending or descending order. The behaviour of sort program, written to satisfy this requirement, will depend on the decision taken by the programmers while writing sort. Testers are often faced with incomplete/ambiguous requirements. In such situations a testers may resort to a variety of ways to determine what behaviour to expect from the program under test).
- Regardless of the nature of the requirements, testing requires the determination of the expected behaviour of the program under test. The observed behaviour of the program is compared with the expected behaviour to determine if the program functions as desired.

Input Domain and Program Correctness

- A program is considered correct if it behaves as desired on all possible test inputs. Usually, the set of all possible inputs is too large for the program to be executed on each input.
- For integer value, -32,768 to 32,767. This requires 2^{32} executions.
- Testing a program on all possible inputs is known as "exhaustive testing".
- If the requirements are complete and unambiguous, it should be possible to determine the set of all possible inputs.

Definition: _*Input Domain*_

- The set of all possible inputs to program P is known as the input domain, or input space, of P.
- Modified requirement 2: It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request character which should be "A" when an ascending sequence is desired, and "D" otherwise while providing input to the program, the request character is entered first followed by the sequence of integers to be sorted. The sequence is terminated with a period.

Definition: _*Correctness*_

A program is considered correct if it behaves as expected on each element of its input domain.

Valid and Invalid Inputs:

- The input domains are derived from the requirements. It is difficult to determine the input domain for incomplete requirements.
- Identifying the set of invalid inputs and testing the program against these inputs are important parts of the testing activity. Even when the requirements fail to specify the program behaviour on invalid inputs, the programmer does treat these in one way or another. Testing a program against invalid inputs might reveal errors in the program.
  Ex: sort program
  < E 7 19...>
  The sort program enters into an infinite loop and neiter asks the user for any input nor responds to anything typed by the user. This observed behaviour poins to a possible error in sort.

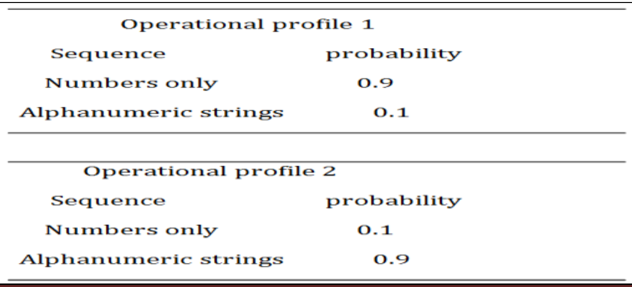
Correctness versus reliability:

- Though correctness of a program is desirable, it is almost never the objective of testing.
- To establish correctness via testing would imply testing a program on all elements in the input domain, which is impossible to accomplish in most cases that are encountered in practice.
- Thus, correctness is established via mathematical proofs of programs.
- While correctness attempts to establish that the program is error-free, testing attempts to find if there are any errors in it.
- Thus, completeness of testing does not necessarily demonstrate that a program is error-free.
- Removal of errors from the program. Usually improves the chances, or the probability, of the program executing without any failure.
- Also testing, debugging and the error-removal process together increase confidence in the correct functioning of the program under test.
- Example:

```
Integer x, y
Input x, y
If(x<y)   ←this condition should be x≤ y
{
        Print f(x, y)
}
Else(x
{
        Print g(x, y)
}
```

- Suppose that function f produces incorrect result whenever it is invoked with x=y and that f(x, y)≠ g(x, y), x=y. In its present form the program fails when tested with equal input values because function g is invoked instead of function f. When the error is removed by changing the condition x<y to x≤ *y*, the program fails again when the input values are the same. The latter failure is due to the error in function f. In this program, when the error in f is also removed, the program will be correct assuming that all other code is correct.
- A comparison of program correctness and reliability reveals that while correctness is a binary metric, reliability is a continuous metric, over a scale from 0 to 1. A program can be either correct or incorrect, it is reliability can be anywhere between 0 and 1. Intuitively when an error is removed from a program, the reliability of the program so obtained is expected to be higher than that of the one that contains the error.

Program Use and Operational Profile:



Operational profile 1

Sequence	probability
Numbers only	0.9
Alphanumeric strings	0.1

Operational profile 2

Sequence	probability
Numbers only	0.1
Alphanumeric strings	0.9

- An operational profile is a numerical description of how a program is used. In accordance with the above definition, a program might have several operational profiles depending on its users.
- Example: sort program

Testing and Debugging

- (Testing is the process of determining if a program behaves as expected.) In the process one may discover errors in the program under test. However, when testing reveals an error, (the process used to determine the cause of this error and to remove it is known as debugging.) As illustrated in figure, testing and debugging are often used as two related activities in a cyclic manner.
  Steps are
  1. Preparing a test plan
  2. Constructing test data
  3. Executing the program
  4. Specifying program behaviour
  5. Assessing the correctness of program behaviour
  6. Construction of oracle

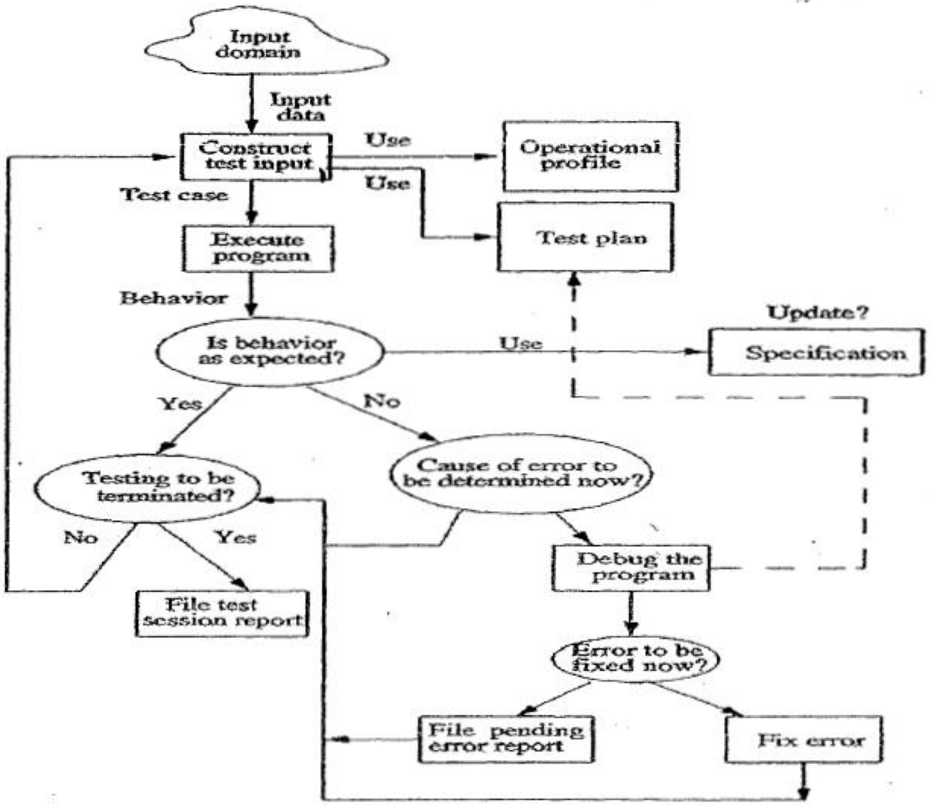


Figure: A test and Debug cycle.

❖ Preparing a test plan:

(A test cycle is often guided by a test plan. When relatively small programs are being tested, a test plan is usually informal and in the tester's mind or there may be no plan at all.)
Example test plan: Consider following items such as the method used for testing, method for evaluating the adequacy of test cases, and method to determine if a program has failed or not.
Test plan for sort:
The sort program is to be tested to meet the requirements given in example
1. Execute the program on at least two input sequence one with "A" and the other with "D" as request characters.
2. Execute the program on an empty input sequence
3. Test the program for robustness against erroneous input such as "R" typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the company failure report form.

❖ Constructing Test Data:

- A test case is a pair consisting of test data to be input to the program and the expected output.
- The test data is a set of values, one for each input variable.
- A test set is a collection of zero or ore cases.
  Program requirements and the test plan help in the construction of test data. Execution of the program on test data might begin after al or a few test cases have been constructed.
  Based on the results obtained, the testers decide whether to continue the construction of additional test cases or to enter the debugging phase.
  The following test cases are generated for the sort program using the test plan in the previous figure.

Test case 1:
  Test data: `<"A" 12 -29 32 >`
  Expected output: `-29 12 32`

Test case 2:
  Test data: `<"D" 12 -29 32.>`
  Expected output: `32 12 -29`

Test case 3:
  Test data: `<"A".>`
  Expected output: `No input to be sorted in ascending order`

Test case 4:
  Test data: `<"D".>`
  Expected output: `No input to be sorted in descending order`

Test case 5:
  Test data: `<"R" 3 17.>`
  Expected output: `Invalid request character;`

valid characters: "A" and "D"

Test case 6:
  Test data: `<"A" c 17.>`
  Expected output: *Invalid number*

❖ Executing the program:

- Execution of a program under test is the next significant step in the testing. Execution of this step for the sort program is most likely a trivial exercise. The complexity of actual program execution is dependent on the program itself.
- Testers might be able to construct a test harness to aid is program execution. The harness initializes any global variables, inputs a test case, and executes the program. The output generated by the program may be saved in a file for subsequent examination by a tester.

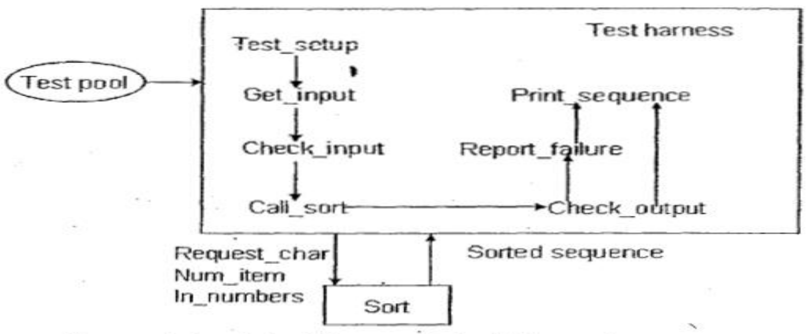


Figure: A simple test harness to test the *sort* program.

In preparing this test harness assume that:
(a) Sort is coded as a procedure

(b) The get-input procedure reads the request character & the sequence to be sorted into variables request_char, num_items and in_number, test_setup procedure-invoked first to set up the test includes identifying and opening the file containing tests.

- Check_output procedure serve as the oracle that checks if the program under test behaves correctly.
- Report_failure: output from sort is incorrect. May be reported via a message(or)saved in a file.
- Print_sequence: prints the sequence generated by the sort program. This also can be saved in file for subsequent examination.

❖ Specifying program behaviour:

State→ can be used to define program behaviour.

$$\left.\begin{array}{l}\text{State transmission diagram, (or)}\\ \text{Simply state diagram}\end{array}\right\} \longrightarrow \begin{array}{l}\text{specify program}\\ \text{behaviour}\end{array}$$

State vector: collecting the current values of program variables into a vector known as the state vector.
An indication of where the control of execution is at any instant of time can be given by using an identifier associated with the next program statement.

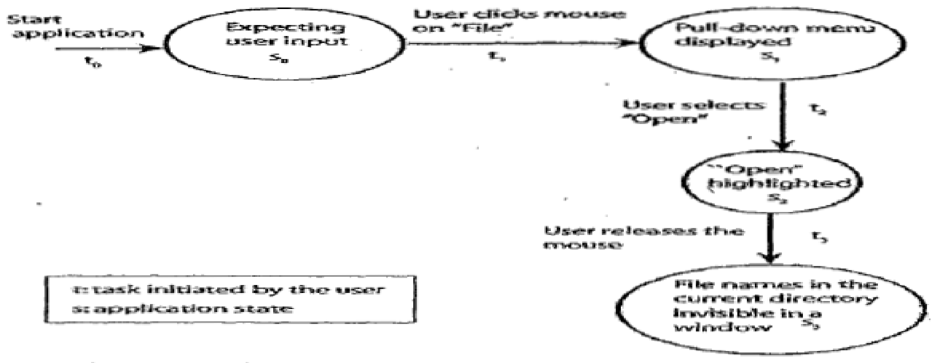

Start application
t_0
Expecting user input s_0
User clicks mouse on "File"
t_1
Pull-down menu displayed s_1
User selects "Open"
t_2
"Open" highlighted s_2
User releases the mouse
t_3
File names in the current directory visible in a window s_3
t: task initiated by the user
s: application state

Figure: A state sequence for *myapp* showing how the application is expected to behave when the user selects the `open` option under the `file` menu

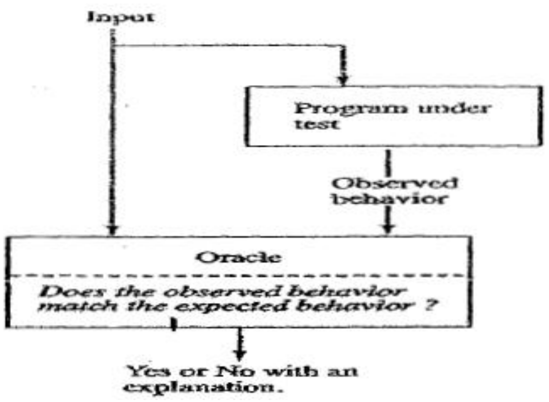
State sequence diagram can be used to specify the behavioural requirements. This same specification can then be used during the testing to ensure if the application confirms to the requirements.

❖ Assessing the correctness of program

Behaviour: It has two steps:

1. Observes the behaviour
2. Analyzes the observed behaviour.

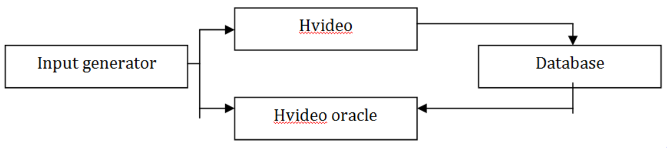
Above task, extremely complex for large distributed system
The entity that performs the task of checking the correctness of the observed behaviour is known as an oracle.


Input
Program under test
Observed behavior
Oracle
Does the observed behavior match the expected behavior ?
Yes or No with an explanation.

- But human oracle is the best available oracle.
- Oracle can also be programs designed to check the behaviour of other programs.

- **Construction of oracles:**
- Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, Requires determination of I/O relationship. When tests are generated from models such as finite-state machines(FSMs)or state charts, both inputs and the corresponding outputs are available. This makes it possible to construct an oracle while generating the tests.

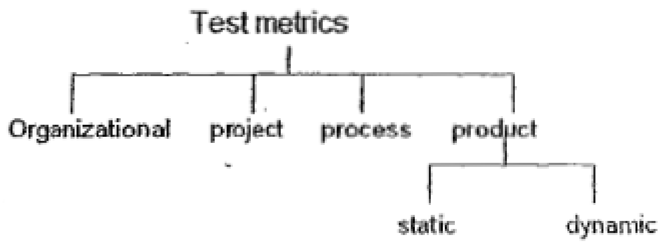
  Example: Consider a program named Hvideo that allows one to keep track of home videos. In the data entry mode, it displays a screen in which the user types in information about a DVD. In search mode, the program displays a screen into which a user can type some attribute of the video being searched for and set up a search criterion.
- To test Hvideo we need to create an oracle that checks whether the program function correctly in data entry and search nodes. The input generator generates a data entry request. The input generaor now requests the oracle to test if Hvideo performed its task correctly on the input given for data entry.

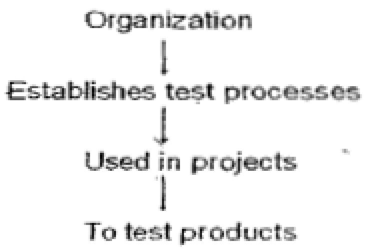


- The oracle uses the input to check if the information to be entered into the database has been entered correctly or not. The oracle returns a pass or no pass to the input generator.

TEST METRICS

- The term metric refers to a standard of measurement. In software testing, there exist a variety of metrics.





There are four general core areas that assist in the design of metrics → schedule, quality, resources and size.

Schedule related metrics:
Measure actual completion times of various activities and compare these with estimated time to completion.

Quality related metrics:
Measure quality of a product or a process

Resource related metrics:
Measure items such as cost in dollars, man power and test executed.

Size-related metrics:
Measure size of various objects such as the source code and number of tests in a test suite

Organizational metrics:
Metrics at the level of an organization are useful in overall project planning and management.
Ex: the number of defects reported after product release, averaged over a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.

- Organizational metrics allow senior management to monitor the overall strength of the organization and points to areas of weakness. Thus, these metrics help senior management in setting new goals and plan for resources needed to realize these goals.

Project metrics:

- Project metrics relate to a specific project, for example the I/O device testing project or a compiler project. These are useful in the monitoring and control of a specific project.
  1. Actual/planned system test effort is one project metrics. Test effort could be measured in terms of the tester_man_months.
  2. Project metric= $\frac{no.of\ success\ ful\ tests}{total\ number\ of\ tests\ in\ the\ system\ phase}$

Process metrics:

- Every project uses some test process. Big-bang approach well suited for small single person projects. The goal of a process metric is to assess the goodness of the process.
- Test process consists of several phases like unit test, integration test, system test, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the consttier it is to fix.

Product metrics: Generic

$\longrightarrow$ Cyclomatic complexity

$\longrightarrow$ Halstead metrics

<u>Cyclomatic complexity</u>

V(G)= E-N+2P

Program p containing N node, E edges and p connected procedures.

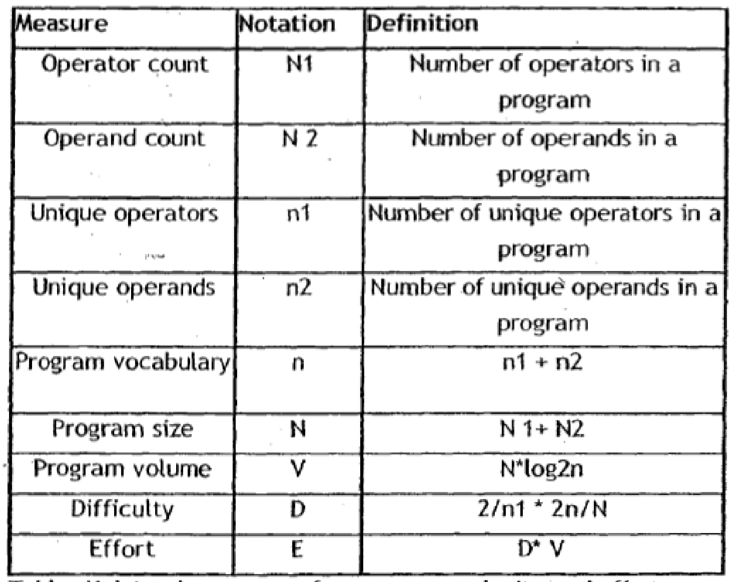
Larger value of V(G)→higher program complexity & program more difficult to understand &test than one with a smaller values.

V(G)→ values 5 or less are recommended

<u>Halstead complexity</u>

Number of error(B) found using program size(S) and effort(E)

B= $7.6E^{0.667}S^{0.33}$



Measure	Notation	Definition
Operator count	N1	Number of operators in a program
Operand count	N 2	Number of operands in a program
Unique operators	n1	Number of unique operators in a program
Unique operands	n2	Number of unique operands in a program
Program vocabulary	n	n1 + n2
Program size	N	N 1+ N2
Program volume	V	N*log2n
Difficulty	D	2/n1 * 2n/N
Effort	E	D* V

Table: Halstead measures of program complexity and effort

Product metrics: OO software

Metrics are reliability, defect density, defect severity, test coverage, cyclomatic complexity, weighted methods/class, response set, number of children.

Static and dynamic metrics:
Static metrics are those computed without having to execute the product.
Ex: no. of testable entities in an application. Dynamic metric requires code execution.
Ex: no. of testable entities actually covered by a test suite is a dynamic quality.

Testability:

- According to IEEE, testability is the "degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met".
- Two types:
  → static testability metrics
  →dynamic testability metrics

Static testability metric:
Software complexity is one static testability metric. The more complex an application, the lower the testability, that is higher the effort required to test it.

Dynamic metrics for testability includes various code based coverage criteria.
Ex: when it is difficult to generate tests that satisfy the statement coverage criterion is considered to have low testability them one for which it is easier to construct such tests.

UNIT 1 QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	How do you measure Software Quality? Discuss Correctness versus Reliability Pertaining to Programs?	Jan 10	10
2	Discuss Various types of Metrics used in software testing and Relationship?	Jan 10	10
3	Define the following i) Errors ii) Faults iii) Failure iv) Bug	June 10	4
4	Discuss Attributes associated with Software Quality?	June 10	8
5	What is a Test Metric? List Various Test Metrics ?and Explain any two?	June 10	8
6	Explain Static & Dynamic software quality Attributes?	July 11	8
7	Briefly explain the different types of test metrics.	July 11	8
8	What are input domain and program correctness?	July 11	4
9	Why is it difficult for tester to find all bugs in the system? Why might not be necessary for the program to be completely free of defects before its delivered to customers?	Dec 11	10
10	Define software quality. Distinguish between static quality attributes and dynamic quality attributes. Briefly explain any one dynamic quality attribute.	Dec 11	10

UNIT 2
BASICS OF SOFTWARE TESTING - 2

SOFTWARE AND HARDWARE TESTING

There are several similarities and differences between techniques used for testing software and hardware

Software application	Hardware product
Does not integrate over time	Does integrate over time
Fault present in the application will remain and no new faults will creep in unless the application is changed	VLSI chip, that might fail over time due to a fault that did not exist at the time chip was manufactured and tested
Built-in self test meant for hardware product, rarely, can be applied to software designs and code	BIST intended to actually test for the correct functioning of a circuit
It only detects faults that were present when the last change was made	Hardware testers generate test based on fault-models Ex: stuck_at fault model – one can use a set of input test patterns to test whether a logic gate is functioning as expected

- Software testers generate tests to test for correct functionality.
- Sometimes such tests do not correspond to any general fault model
- For example: to test whether there is a memory leak in an application, one performs a combination of stress testing and code inspection
- A variety of faults could lead to memory leaks
- Hardware testers use a variety of fault models at different levels of abstraction
- Example:
    - transistor level faults → low level
    - gate level, circuit level, function level faults → higher level
- Software testers might not or might use fault models during test generation even though the model exist
- Mutation testing is a technique based on software fault models
- Test Domain → a major difference between tests for hardware and software is in the domain of tests
- Tests for VLSI chips for example, take the form of a bit pattern. For combinational circuits, for example a Multiplexer, a finite set of bit patterns will ensure the detection of any fault with respects to a circuit level fault model.
- For software, the domain of a test input is different than that for hardware. Even for the simplest of programs, the domain could be an infinite set of tuples with each tuple consisting of one or more basic data types such as integers and reals.

Example

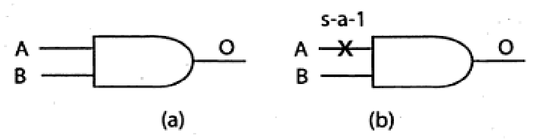
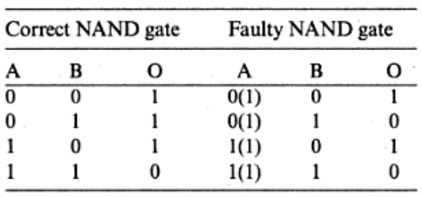


Fig. 1.11 (a) A two-input NAND gate. (b) A NAND gate with a stuck-at-1 fault in input A.



Correct NAND gate			Faulty NAND gate		
A	B	O	A	B	O
0	0	1	0(1)	0	1
0	1	1	0(1)	1	0
1	0	1	1(1)	0	1
1	1	0	1(1)	1	0

Consider a simple twp-input NAND gate in Fig.

A test bit vector V: (A=O, B=1) leads to output 0. Whereas the correct output should be 1: Thus V detects a single S-a-1 fault to the A input of the NAND gate. There could be multiple stuck-at faults also.

- Test Coverage → It is practically impossible to completely test a large piece of software, for example, an OS as well as a complex integrated circuit such as modern 32 or 64 bit Microprocessor. This leads to a notion of acceptable test coverage. In VLSI testing such coverage is measured using a fraction of the faults covered to the total that might be present with respect to a given fault model.

- The idea of fault coverage to hardware is also used in software testing using program mutation. A program is mutated by injecting a number of faults using a fault model that corresponds to mutation operators. The effectiveness or adequacy of a test case is assessed as a fraction of the mutants covered to the total number of mutatis.

TESTING AND VERIFICATION

- Program verification aims at proving the correctness of progress by showing that is contains no errors.
- This is very different from testing that aims at uncovering errors in a program.
- While verification aims at showing that a given program works for all possible inputs that satisfy a set of conditions, testing aims to show that the given program is reliable to that, no errors of any significance were found.
- Program verification and testing are best considered as complimentary techniques.
- In the developments of critical applications, such as smart cards or control of nuclear plants, one often makes use of verification techniques to prove the correctness of some artifact created during the development cycle, not necessarily the complete program.
- Regardless of such proofs, testing is used invariably to obtain confidence in the correctness of the application.
- Testing is not a perfect process in that a program might contain errors despite the success of a set of tests; verification might appear to be a perfect process as it promises to verify that a program is free from errors.
- Verification reveals that it has its own weakness.
- The person who verified a program might have made mistakes in the verification process' there might be an incorrect assumption on the input conditions; incorrect assumptions might be made regarding the components that interface with the program.
- Thus, neither verification nor testing is a perfect technique for proving the correctness of program.

DEFECT MANAGEMENT

Defect Management is an integral part of a development and test process in many software development organizations. It is a sub process of a the development process. It entails the following:

- Detect prevention
- Discovery
- Recording and reporting
- Classification
- Resolution
- Production

Defect Prevention

It is achieved through a variety of process and tools: They are,

- Good coding techniques.
- Unit test plans.
- Code Inspections.

Defect Discovery

- Defect discovery is the identification of defects in response to failures observed during dynamic testing or found during static testing.
- It involves debugging the code under test.

Defect Classification

Defects found are classified and recorded in a database. Classification becomes important in dealing with the defects. Classified into

- High severity-to be attended first by developer.

- Low severity.

Example: Orthogonal defect classification is one of the defect classification scheme which exist called ODC, that measures types of defects, this frequency, and Their location to the development phase and documents.

Resolution

Each defect, when recorded, is marked as 'open' indicating that it needs to be resolved. It required careful scrutiny of the defects, identifying a fix if needed, implementing the fix, testing the fix, and finally closing the defect indicating that every recorded defect is resolved prior to release.

Defect Prediction

- Organizations often do source code Analysis to predict how many defects an application might contain before it enters the testing the phase.
- Advanced statistical techniques are used to predict defects during the test process.
- Tools are existing for Recording defects, and computing and reporting defect related statistics.
  - BugZilla - Open source
  - Fog-Buzz - commercially available tools.

EXECUTION HISTORY

Execution history of a program, also known as execution trace, is an organized collection of information about various elements of a program during a given execution. An execution slice is an executable subsequence of execution history. There are several ways to represent an execution history,

- ➢ Sequence in which the functions in a given program are executed against a given test input,
- ➢ Sequence in which program blocks are executed.
- ➢ Sequence of objects and the corresponding methods accessed for object oriented languages such as Java An execution history may also included values of program variables.

- ▪ A complete execution history recorded from the start of a program's execution until its termination represents a single execution path through the program.
- ▪ It is possible to get partial execution history also for some program elements or blocks or values of variables are recorded along a portion of the complete path.

TEST GENERATION STRATEGIES

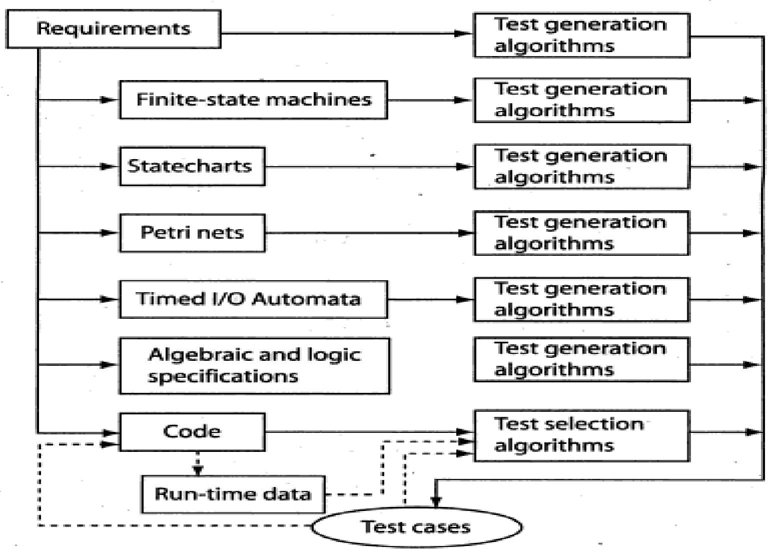


Fig. 1.12 Requirements, models, and test generation algorithms.

Test generation uses a source document. In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on knowledge of the requirements.

Fig summarizes the several strategies for test generation. These may be informal techniques that assign value to input variables without the use of any rigorous or formal methods. These could also be techniques that identify input variables, capture the relationship among these variables, and use formal techniques for test generation such as random test generation and cause effect graphing.

- Another set of strategies fall under the category of model based test generation. These strategies require that a subset of the requirements be modelled using a formal notation.
- FSMs, statecharts, petrinets and timed I/O automata are some of the well known and used formal notations for modelling various subset requirements.
- Sequence & activity diagrams in UML also exist and are used as models of subsets of requirements.
- There also exist techniques to generate tests directly from the code i.e. code based test generation.
- It is useful when enhancing existing tests based on test adequacy criteria.
- Code based test generation techniques are also used during regression testing when there is often a need to reduce the size of the suite or prioritize tests, against which a regression test is to be performed.

STATIC TESTING

- Static testing is carried out without executing the application under test.
- This is in contrast to dynamic testing that requires one or more executions of the application under test.
- It is useful in that it may lead to the discovery of faults in the application, ambiguities and errors in the requirements and other application-related document, at a relatively low cost,
- This is especially so when dynamic testing expensive.
- Static testing is complementary to dynamic testing.
- This is carried out by an individual who did not write the code or by a team of individuals.
- The test team responsible for static testing has access to requirenments document, application, and all associated documents such as design document and user manual.
- Team also has access to one or more static testing tools.
  A static testing tool takes the application code as input and generates a variety of data useful in the test process.

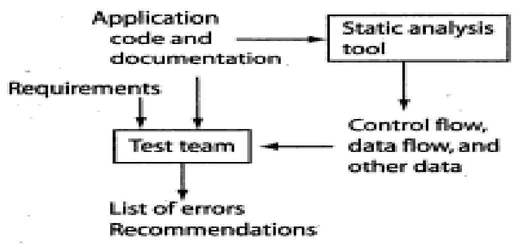

Application code and documentation
Static analysis tool
Requirements
Test team
Control flow, data flow, and other data
List of errors Recommendations

Fig. 1.13 Elements of static testing.

WALKTHROUGHS

- ✓ Walkthroughs and inspections are an integral part of static testing.
- ✓ Walkthrough are an integral part of static testing.
- ✓ Walkthrough is an informal process to review any application-related document.

eg:

requirements are reviewed---->requirements walkthrough
code is reviewed---->code walkthrough
(or)
peer code review

Walkthrough begins with a review plan agreed upon by all members of the team.

Advantages:

- improves understanding of the application.

- both functional and non functional requirements are reviewed.
- A detailed report is generated that lists items of concern regarding the requirements.

INSPECTIONS

✓ Inspection is a more formally defined process than a walkthrough. This term is usually associated with code.

✓ Several organizations consider formal code inspections as a tool to improve code quality at a lower cost than incurred when dynamic testing is used.

Inspection plan:

i. statement of purpose
ii. work product to be inspected this includes code and associated documents needed for inspection.
iii. team formation, roles, and tasks to be performed.
iv. rate at which the inspection task is to be completed
v. Data collection forms where the team will record its findings such as defects discovered, coding standard violations and time spent in each task.

Members of inspection team

a) *Moderator:* in charge of the process and leads the review.
b) *Leader:* actual code is read by the reader, perhaps with help of a code browser and with monitors for all in the team to view the code.
c) *Recorder:* records any errors discovered or issues to be looked into.
d) *Author:* actual developer of the code.

It is important that the inspection process be friendly and non confrontational.

Use of static code analysis tools in static testing

- Static code analysis tools can be provide control flow and data flow information.
- Control flow information presented in terms of a CFG, is helpful to the inspection team in that it allows the determination of the flow of control under different conditions.
- A CFG can be annotated with data flow information to make a data flow graph.
- This information is valuable to the inspection team in understanding the code as well as pointing out possible defect.

Commercially available static code analysis tools are:

- Purify → IBM Rationale
- Klockwork → Klockwork
- LAPSE (Light weight analysis for program security in eclipse) → open source tool

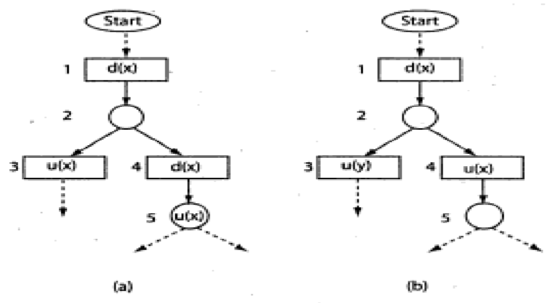


Fig. 1.14 Partial CFGs annotated with data-flow information. d(x) and u(x) imply the definition and use of variable x in a block, respectively. (a) CFG that indicates a possible data-flow error. (b) CFG with a data-flow error.

(a) CFG clearly shows that the definition of x at block 1 is used at block-3 but not at block 5.In fact the definition of x at block 1 is considered killed due to its redefinition at block 4.

(b) CFG indicates the use of variable y in the block 3.If y is not defined along the path from start to block 3,then there is a data-flow error as a variable is used before it is defined.

Several such errors can be detected by static analysis tools.

->compute complexity metrics, used as a parameter in deciding which modules to inspect first.

Model-Based Testing and Model checking:

- Model based testing refers to the acts of modeling and the generation of tests from a formal model of application behavior.
- Model checking refers to a class of techniques that allow the validation of one or more properties from a given model of an application.

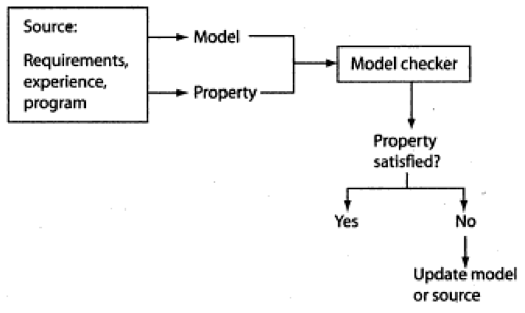


Fig. 1.15 Elements of model checking.

- Above diagram illustrates the process of model-checking. A model, usually finite state is extracted from some source. The source could be the requirements and in some cases, the application code itself.
- One or more desired properties are then coded to a formal specification language. Often, such properties are coded in temporal logic, a language for formally specifying timing properties. The model and the desired properties are then input to a model checker. The model checker attempts to verify whether the given properties are satisfied by the given model.
- For each property, the checker could come up with one of three possible answer:
  - the property is satisfy
  - the property is not satisfied.
  - or unable to determine
- In the second case, the model checker provides a counter example showing why the property is not satisfied.
- The third case might arise when the model checker is unable to terminate after an upper limit on the number of iterations has reached.
- While model checking and model based testing use models, model checking uses finite state models augmented with local properties that must hold at individual states. The local properties are known as atomic propositions and augmented models as **kripke structure.**

CONTROL FLOW GRAPH

- A CFG captures the flow of control within a program. Such a graph assists testers in the analysis of a program to a understand its behaviour in terms of the flow of control. A CFG can be constructed manually without much difficulty for relatively small programs, say containing less than about 50 statements.
- However, as the size of the program grows, so does the difficulty of constructing its CFG and hence arises the need for tools.
- A CFG is also known by the names flow graph or program and it is not to be confused with program-dependence graph(PDG).

Basic Block

- Let P denotes a program written in a procedural programming language, be it high level as C or Java or low level such as the 80x86 assembly. A basic block, or simply a block, in P is a sequence of consecutive statements with a single entry and a single exit point.
- Thus, a block has unique entry and exit points.
- Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block co inside when the block contains only one statement.
- example: the following program takes two integers x and y and output x^y.

- There are a total of 17 lines in this program including the begin and end. The execution of this program begins at line 1 and moves through lines 2, 3 and 4 to the line 5 containing an if statement. Considering that there is a decision at line 5, control could go to one of two possible destinations at line 6 and 8. Thus, the sequence of statements starting at line 1 and ending at line 5 constitutes a basic block. Its only entry point is at line 1 and the only exit point is at line 5.

Program P1.2

```
1  begin
2     int x, y, power;
3     float z;
4     input (x, y);
5     if (y<0)
6       power=-y;
7     else
8       power=y;
9     z=1;
10    while (power!=0){
11      z=z*x;
12    power=power-1;
13    }
14    if (y<0)
15      z=1/z;
16    output(z);
17 end
```

Block	Lines	Entry Point	Exit Point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Note: ignored lines 7 and 13 from the listing because these are syntactic markers, and so are begin and end that are also ignored.

Flow Graph: Definition and pictorial representation

- A flow graph G is defines as a finite set N of nodes and a finite set E of a directed edges. In a flow graph of a program P, we often use a basic block as a node and edges indicate the flow of control across basic blocker.
- A pictorial representation of a flow graph is often used in the analysis of control behaviour of a program. Each node is represented by a symbol, usually an oval or a rectangular box. These boxes are labelled by their corresponding block numbers. The boxes are connected by lines representing edges. Arrows are used to indicate the direction of flow. These edges are labelled true or false to indicate the path taken when the condition evaluates to true and false respectively.
- N={start,1,2,3,4,5,6,7,8,9,end}
- E={(start,1),(1,2),(1,3),(2,4),(3,4),(4,5),(5,6),(6,5),(5,7),(7,8),(7,9),(9,end)}

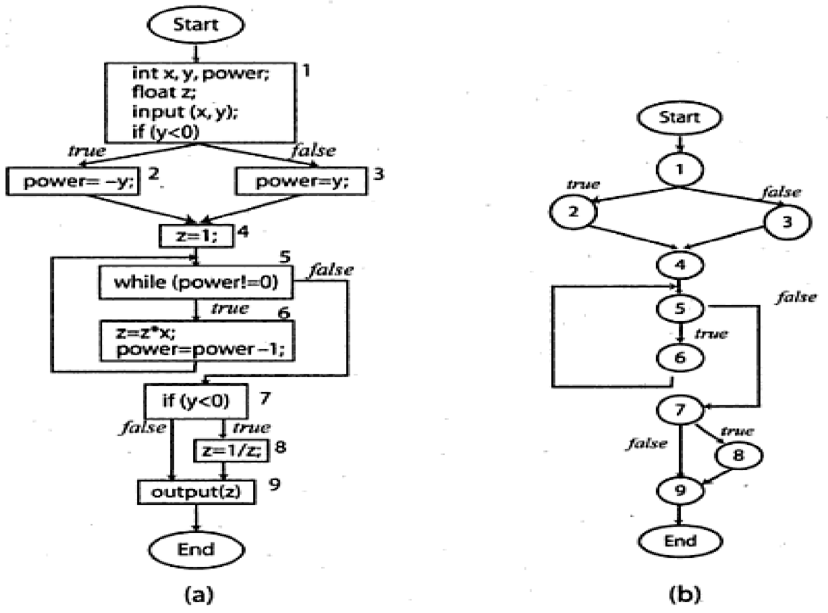


Fig. 1.16 Flow graph representations of Program P1.2. (a) Statements in each block are shown. (b) Statements within a block are omitted.

Path

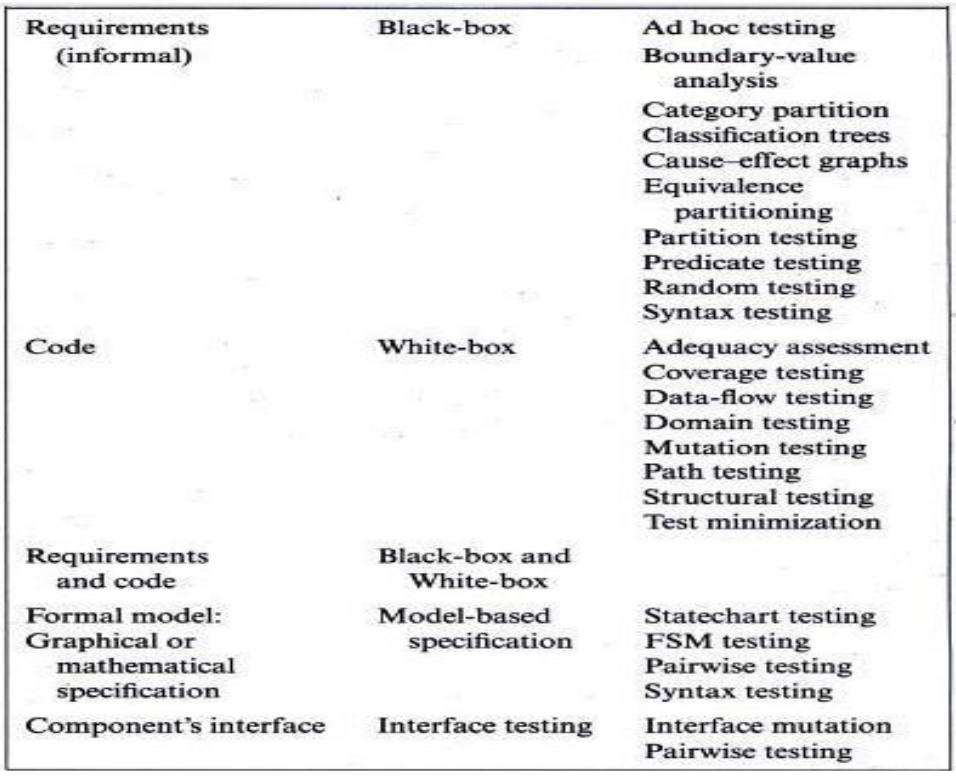
- A path through a flow graph is considered complete if the first node along the path is considered complete if the first node along the path is start and the terminating node is END.
- A path p through a flow graph for a program p is considered feasible if there exists at least one test case which when input to p causes p to be traversed. If no such test case exists, then p is considered infeasible. Whether a given path p through a program p is feasible is in general an undecidable problem.
- This statement implies that it is not possible to write an algorithm that takes as inputs an arbitrary program and a path through that program, and corr

TYPES OF TESTING

- Framework consists of a set of five classifies that serve to classify testing techniques that fall under the dynamic testing category.Dynamic testing requires the excution of program under test.Static testing consists of testing for the review and analysis of the program.
- five classifiers of testing:-
  - 1.C1:source of test generation
  - 2.C2:life cycle phase in which testing takes place
  - 3.C3:goal of a specific testing activity.
  - 4.C4:characteristics of the artifact under test
  - 5.C5:test process

Classifier C1: Source of test generation

- Black box Testing: Test generation is an essential part of testing. There are a variety of ways to generate tests, listed in table. Tests could be generated from informally or formally specified requirements and without the aid of the code that is under test. Such form of testing is commonly referred to as black box testing.



Requirements (informal)	Black-box	Ad hoc testing, Boundary-value analysis, Category partition, Classification trees, Cause–effect graphs, Equivalence partitioning, Partition testing, Predicate testing, Random testing, Syntax testing
Code	White-box	Adequacy assessment, Coverage testing, Data-flow testing, Domain testing, Mutation testing, Path testing, Structural testing, Test minimization
Requirements and code	Black-box and White-box	
Formal model: Graphical or mathematical specification	Model-based specification	Statechart testing, FSM testing, Pairwise testing, Syntax testing
Component's interface	Interface testing	Interface mutation, Pairwise testing

Model based or specification based testing:

- Model based or specification based testing occurs when the requirements are formally specified as for example, using one or more mathematical or graphical notations such as, z, statecharts, event sequence graphs

White box testing:

- White box testing refers to the test activity where in code is used in the generation of or the assessment of the test cases.
- Code could be used directly or indirectly for test generation.
    - In the direct case, a tool, or a human tester examines the code and focuses on a given path to be covered. A test is generated to cover path.
    - In the indirect case, test generated using some black box testing is assessed against some code based coverage criterion.
- Additional tests are then generated to cover the uncovered positions of the code by the analyzing which parts of the code are feasible.
- Control flow, data flow, and mutation testing can be used for direct as well as indirect code-based test generation.

Interface testing:

- Tests are often generated using a components interface.
- Interface itself forms a part of the components requirements and hence this form of testing is black box testing. However, the focus on the interface leads us to consider interface testing in its own right. Techniques such as
    - --->pairwise testing
    - --->interface mutation

Pairwise testing:

- Set of values for each input is obtained from the components requirement.

Interface mutation:

- The interface itself, such as function coded in /c orCORBA component written in an IDL,serves to extract the information needed to perform interface mutation.
    - pairwise testing:is a black box testing
    - interface mutation:is a white box testing

Ad-hoc testing:

- In adhoc testing,a tester generates tests from requirements but without the use of any systematic method.

Random testing:

- Random testing uses a systematic method to generate tests.Generation of tests using random testing requires modeling the input space randomly.

Classifier C2: Life cycle phase

- Testing activities take place throughout the software life cycle.
- Each artifact produced is often subject to testing at different levels of rigor and using different testing techniques.

Unit testing:

- Programmers write code during the early coding phase.
- They test their code before it is integrated with other system components.
- This type of testing is referred to as the unit testing.

System testing:

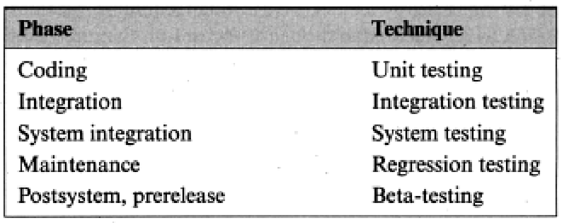
- When units are integrated and a large component or a subsystem formed, programmers do integration testing of the sub system.
- System testing is to ensure that all the desired functionality is in the system and works as per its requirements.
- Note: test designed during unit testing are not likely to be used during integrating and system testing.

Acceptance testing:

- two types:
    - -alpha testing

- -beta testing
  - Carefully selected set if customers are asked to test a system before commercialization.
  - This form of testing is referred to as beta testing.
  - In case of contract software, the customer who contracted the development performs acceptability testing prior to making the final decisions as to whether to purchase the application for deployment.

Table 1.5 Classification of techniques for testing computer software. Classifier C2: Life cycle phase



Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Postsystem, prerelease	Beta-testing

Classifier C3: Goal-directed testing

There exists a variety of goals of course finding any hidden errors is the prime goal of testing, goal-oriented testing books for specific type of failure.

Robustness testing:

- Robustness testing refers to the task of testing an application for robustness against unintended inputs. It differs from functional testing in that the tests for robustness are derived from outside of the valid (or expected) input space, whereas in the former the tests are derived from the valid input space.

Stress testing:

- In stress testing, one checks for the behavior of an application under stress. Handling of overflow of data storage, for example buffers, can be checked with the help of stress testing.

Performance testing:

- The term performance testing refers to that phase of testing where an application tested specifically with performance requirements in the view.
- Ex: An application might be required to process 1,000billing transactions per minute on a specific intel processer-based machine and running a specific OS.

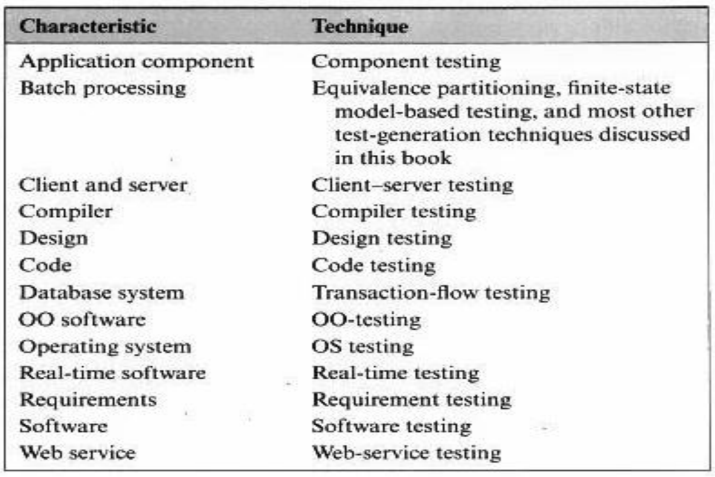
Load testing:

- The term load testing refers to that phase of testing in which an application is loaded with respect to one or more applications. The goal is to determine if the application continues to perform as required under various load conditions.
- Ex: a database server can be loaded with requests from a large number of simulated users.

Classifier C4: Artifact under test

Table 1.7 is a partial list of testing techniques named after the artifact that is being tested. For ex, during the design phase one might generate a design using SDL notation. This form of testing is known as design testing.

Table 1.7 Classification of techniques for testing computer software. Classifier C4: Artifact under test



Characteristic	Technique
Application component	Component testing
Batch processing	Equivalence partitioning, finite-state model-based testing, and most other test-generation techniques discussed in this book
Client and server	Client–server testing
Compiler	Compiler testing
Design	Design testing
Code	Code testing
Database system	Transaction-flow testing
OO software	OO-testing
Operating system	OS testing
Real-time software	Real-time testing
Requirements	Requirement testing
Software	Software testing
Web service	Web-service testing

While testing a batch processing application, it is also important to include an oracle that will check the result of executing each test script. This oracle might be a part of the test script itself. It could, for example, query the contents of a database after performing an operation that is intended to change the status of the database.

Classifier C5: Test process models

Software testing can be integrated into the software development life cycle in a variety of ways. This leads to various models for the tests process listed in the table 1.8

Table 1.8 Classification of techniques for testing computer software. Classifier C5: Test process models

Process	Attributes
Testing in waterfall model	Usually done toward the end of the development cycle
Testing in V-model	Explicitly specifies testing activities in each phase of the development cycle
Spiral testing	Applied to software increments, each increment might be a prototype that eventually leads to the application delivered to the customer. Proposed for evolutionary software development
Agile testing	Used in agile development methodologies such as eXtreme Programming (XP)
Test-driven development (TDD)	Requirements specified as tests

Testing in the waterfall model:

- The waterfall model is one of the earliest and least used, software life cycle.
- Figure 1.23 shows different phases in a development process based on the waterfall model. While verification and validation of documents produced in each phase is an essential activity, static as well as dynamic testing occurs toward the end if the process.
- Waterfall model requires adherence to an inherently sequential process, defects introduced in the early phases and discovered in the later phases could be costly to correct.
- There is a very little iterative or incremental development when using the waterfall model.

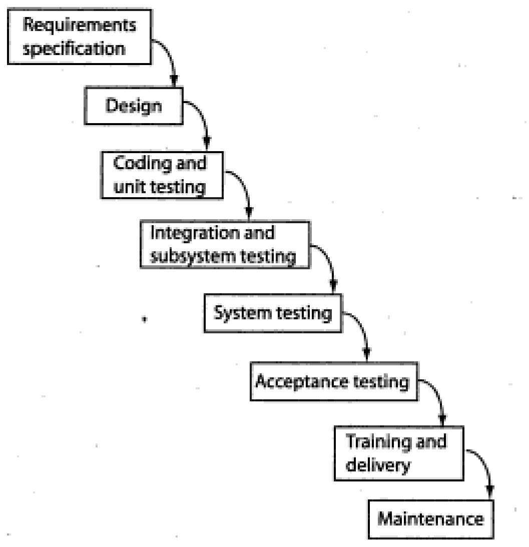


Fig. 1.23 Testing in the waterfall model. Arrows indicate the *flow* of documents from one to the next. For example, design documents are input to the coding phase. The waterfall nature of the flow led to the name of this model.

Testing in the V-model:

The v-model, as shown in the fig, explicitly specifies testing activities associated with each phase of the development cycle. These activities begin from the start and continue until the end of life cycle. The testing activities are carried out parallel with the development activities.

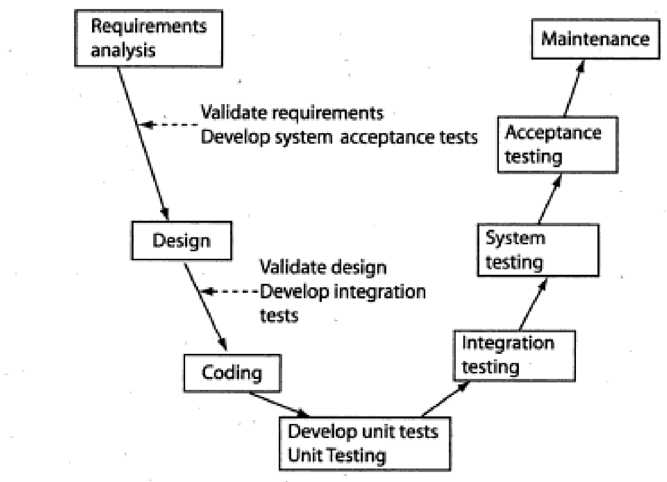


Fig. 1.24 Testing in the V-model.

Spiral testing:

- The term spiral testing is not to be confused with spiral model, through they both are similar in that both can be visually represented as a spiral of activities.
- In the spiral testing, the sophisticated of testing of test activities increases with the stages of an evolving prototype.
- In the early stages, when a prototype is used to evaluate how an application must evolve, one focuses on test planning. The focus at this stage is on how testing will be performed in the remainder of the project.
- Subsequent iterations refine the prototype based on more precise set of requirements.
- Further test planning takes place and unit & integration tests are performed.
- In the final stage ,when the requirements are well defined, testers focus on system and acceptance testing.

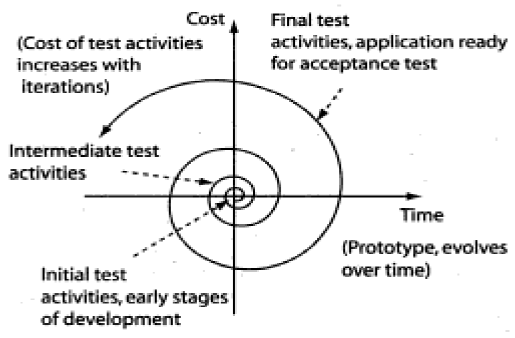

Cost
(Cost of test activities increases with iterations)
Final test activities, application ready for acceptance test
Intermediate test activities
Time
(Prototype, evolves over time)
Initial test activities, early stages of development

Fig. 1.25 A visual representation of spiral testing. Test activities evolve over time and with the prototype. In the final iteration, the application is available for system and acceptance testing.

Agile testing:

Agile testing involves in addition to the usual steps such as test planning, test design and test execution.
Agile testing promotes the following ideas:

- Include testing -related activities throughout a development project starting from the requirement phase.
- Work collaboratively with the customer who specifies requirements in terms of tests.
- testers and development must collaborate with each other rather than serve as adversaries and
- Test often and in small chunks.

THE SATURATION EFFECT

- The saturation effect is an abstraction of a phenomenon observed during the testing of complex software systems.
- The horizontal axis the figure refers to the test effort that increase over time.

- The test effort can be measured as, for ex, the number of test cases executed or total person days spent during the test and debug phase.
- The vertical axis refers to the true reliability (solid lines) and the confidence in the correct behavior (dotted lines) of the application under test evolves with an increase in test effort due to error correction.
- The vertical axis can also be labeled as the cumulative count of failures that are observed over time, that is as the test effort increases.
- The error correction process usually removes the cause of one or more failures.

Confidence and true reliability:

Confidence in fig refers to the confidence of the test manager in the true reliability of the application under test.

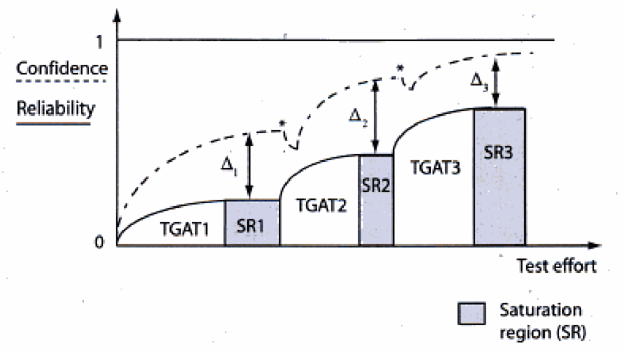


Fig. 1.26 The saturation effect observed during testing of complex software systems. Asterisk (*) indicates the point of drop in confidence due to a sudden increase in failures found; TGAT stands for test-generation and assesment techniques.

- Reliability in the figure refers to the probability of failure free operation of the application under test in its intended environment.
- The true reliability differs from the estimated reliability in that the latter is an estimate of the application reliability obtained by using one of the many statistical methods.
    - 0-indicates lowest possible confidence
    - 1-the highest possible confidence
- Similarly,
    - 0-indicates the lowest possible true reliability
    - 1-the highest possible true reliability.

Saturation region:
->assumes application A in the system test phase.
->the test team needs to generate tests, set up the test environment, and run A against the test.

1. Assume that the testers are generated using a suitable test generation method (TGAT 1) and that each test either passes or fails.
2. If we measure the test effort as the combined effort of testing, debugging and fixing the errors the true reliability increases as shown in the fig.

False sense of confidentiality:

- This false sense of confidence is due to the lack of discovery of new faults, which in turn is due to the inability of the tests generated using TGA1 to exercise the application code in ways significantly different from what has already been exercised.
- Thus, in the saturation region, the robust states of the application are being exercised, perhaps repeatedly, whereas the faults lie in the other states.

Reducing delta:

- Empirical studies reveal that every single test generation method has its limitations in that the resulting test set is unlikely to detect all faults in an application.
- The more complex an application, the more unlikely it is that tests generated using any given method will detect all faults.
- This is one of the prime regions why tests use or must use multiple techniques for test generation.

Impact on test process:

- A knowledge and application of the saturation effect are likely to be of value of any test team while designing and implementing a test process.

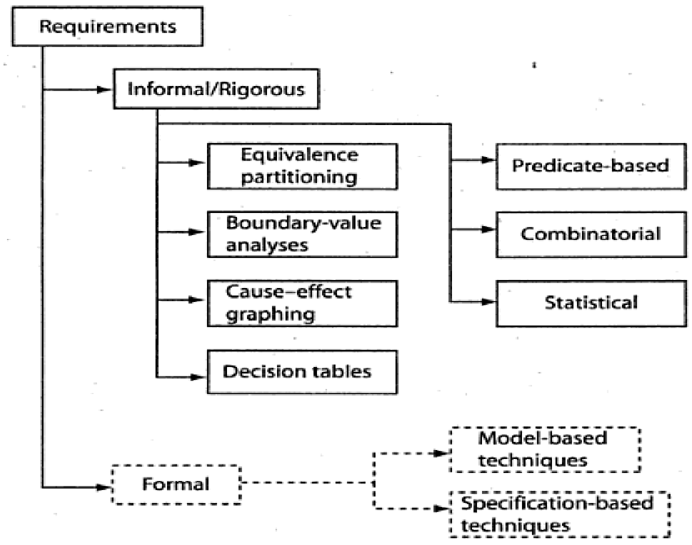
UNIT 2 QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Define the following: i)Testability ii)Verification	June 10	4
2	What is defect management? List the different activities. Explain any two.	June 10	8
3	Explain the following: i) Static testing ii) Model based testing and model checking.	June 10	8
4	Explain how CFG assists the tester in analysis of program to understand the behavior in terms of flow of control with examples?	June 11	10
5	Describe the following test classifiers: i) Source of test generation; ii) Life cycle phase; iii)Test process models.	June 11	10
6	Explain Variety of ways in which Software testing can be integrated into the Software development life cycle.	Dec 11	10
7	Consider the following program: 1) begin 10) while(power 1=0){ 2) int x,y,power; 11) z=z*x; 3) float z; 12) power=power-1; 4) input(x,y); 13) } 5) if(y<0) 14) if(y<0) 6) power=-y; 15) z=1/z; 7) else 16) output(z); 8) power=y; 17) end 9) z=1; Identify the basic blocks, their entry points and exit points. Draw the control flow graph.	Dec 11	6
8	Write a short notes on the saturation effect	Dec 11	4

UNIT 3
TEST GENERATION FROM REQUIREMENTS-1

INTRODUCTION

- A requirement specification can be informal, rigorous, formal, or a mix of these three approaches.
- The more formal the specification, the higher are the chances of automating the test generation process.
- The following figure shows variety of test generation techniques



- Often, high level designs are also considered as a part of specification
- Requirements serve as a source for the identification of a input domain of the application to be developed
- A variety of test generation techniques are available to select a subset of the input domain to serve as test set against which the application will be tested

THE TEST SELECTION PROBLEM

- Let D denote the input domain of program p, the test selection problem is to select a subset of tests such that execution of p against each element of T will reveal all errors in p.
- In general, there does not exist any algorithm, to construct such a test. However, there are heuristics and model based methods that can be used to generate tests that will reveal certain type of faults.
- The challenge is to construct a test set T subset of D that will reveal as many errors in p as possible.
- Test selection is primarily difficult because of the size and complexity of the input domain of p.
- In most practical problems, the input domain is large, that it has many elements, and often complex, that is the elements are of different types such as integers, strings, real, Boolean and structure
- The large size of the input domain prevents testers from exhaustively testing the program under test against all possible inputs
- The complexity makes it harder to select individual tests

Example: Complex input domain

Consider a procedure p in a payroll-processing system that takes an employee's record as input and computes weekly salary. Employee's record consists of

$$\left.\begin{array}{l}\text{ID: int;}\\ \text{Name: string;}\\ \text{Rate: float;}\\ \text{Hrs\_worked: int;}\end{array}\right\}\quad \text{Complex}$$

EQUIVALENCE PARTITIONING

- Test selection using equivalence partitioning allows a tester to subdivide the input domain into relatively small number of sub-domains, say N>1 refer fig(a) , which are disjoint, each subset is known as an equivalence class.

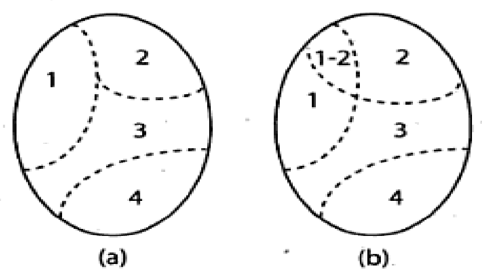


Fig. 2.2 Four equivalence classes created from an input domain. (a) Equivalence classes together constitute a partition as they are disjoint. (b) Equivalence classes are not disjoint and do not form a partition. 1–2 indicates the region where subsets 1 and 2 overlap. This region contains test inputs that belong to subsets 1 and 2.

- The equivalence classes are created assuming that the program under test exhibits the same behavior on all elements that is tests, within a class.
- One test is selected from each equivalence class
- When the equivalence classes created by two tester's are identical, tests generated are different.

Fault targeted

- The entire set of inputs can be divided into at least two subsets
  - One containing all expected(E) or legal inputs
  - Other containing all unexpected(u) or illegal inputs

    E and u are further divided into subsets (refer fig below)

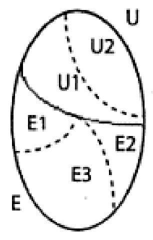


Fig. 2.3 Set of inputs partitioned into two regions one containing expected (*E*), or legal, and the other containing unexpected (*U*), or illegal, inputs. Regions *E* and *U* are further subdivided based on the expected behavior of the application under test. Representative tests, one from each region, is targeted at exposing faults that cause the application to behave incorrectly in their respective regions.

Example:

Consider an application A that takes an integer denoted by 'age' as input, legal values of 'age' are in the range [1, 2, 3 ,.........., 120]

Set of input vales is now divided into E and u.

E=[1, 2,....., 120] u= the rest.

- Furthermore, E is subdivided into [1, 2, ..., 61] and [162, 163, ...,120]

According to requirement R1

According to requirement R2

- Invalid inputs below 1 and above 120 are to be treated differently leading to subdivision of u into two categories.
- Test selected using equivalence partitioning technique aims at targeting faults in A w.r.t inputs in any of the four regions.

Relations And Equivalence Partitioning

- A relation is a set of n-ary-tuples
  Example: a method addList that returns the sum of elements in a list of integers defines a binary relation.
- Each pair in the relation consists of a list and an integer that denotes the sum of all elements in the list.
  Example: ((1,5), 6) and ((-3,14,3), 14)
- The relation computed by addList is defined as follows:
- addList: L→Z
  where, L is a set of all lists of integers and Z is set of integers.
  Suppose hat addList has an error (empty list) then,
  addList: L→Z U{error}
- Relations that help a tester partition the input domain of a program are usually of the kind= R:I→I , where I→input domain
- Below example shows a few ways to define equivalence classes based on the knowledge of requirements and the program text.
  Example: the word count method takes a word w and a filename f as input and returns the number of occurrences of w in the text contained in the file name f.
  If no file with name 'f' exists, an exception is raised.

```
1. begin
2.    string w, f;
3.    input (w, f);
4.    if(!exists(f))[raise exception; return(0)};
5.    if (length(w)==0){return(0)};
6.    return(getcount(w, f));
7. end
```

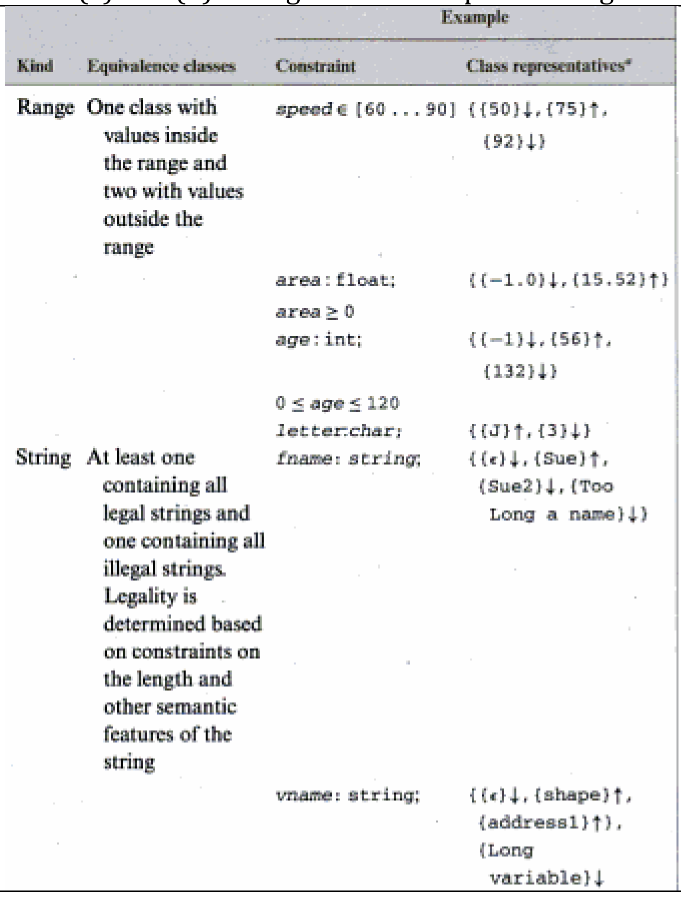
using the partitioning method, we obtain the following eg:classes
E1: consists of pairs(w, f) where w is a string and f denotes a file that exists.
E2: consists of pairs (w, f) where w is a string and f denotes a file that does not exists.

Eq.class	w	f
E1	non-null	Exists, non empty
E2	non-null	Does not exist
E3	non-null	Exists, empty
E4	null	Exists, non empty
E5	null	Does not exist
E6	null	Exists, empty

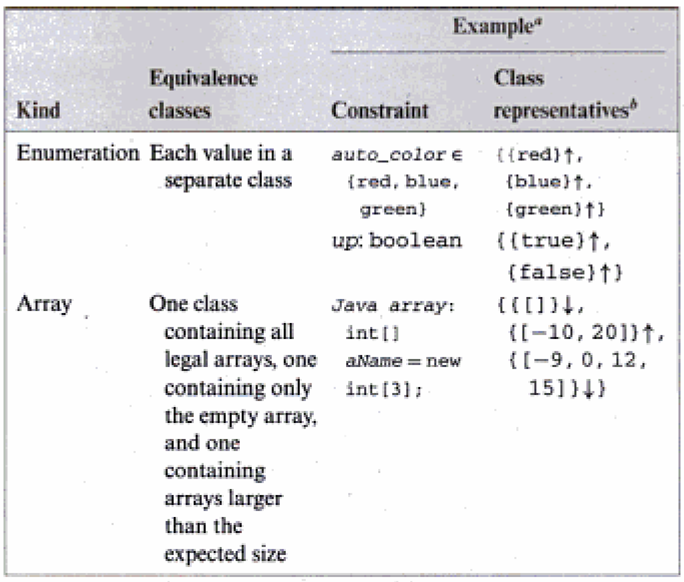
- So we note that the number of eq. Classes without any knowledge of program code is 2, whereas that with the knowledge of partial code is 6.
- Equivalence classes based on program output
  Quest 1: does the program ever generate a 0?
  Quest 2: what are the max and min possible values of the output?
  These two questions lead to following eq. Classes
  E1: output value v is 0
  E2: output value v is, the max. Possible
  E3: output value v is, the min. Possible
  E4: All other output values.

Equivalence Classes For Variables

Table (a) and (b) offer guidelines for partitioning variables into equivalence classes based on their type.



		Example	
Kind	Equivalence classes	Constraint	Class representatives[a]
Range	One class with values inside the range and two with values outside the range	`speed` ∈ [60 ... 90]	{(50)↓, (75)↑, (92)↓}
		`area`: float; `area` ≥ 0	{(−1.0)↓, (15.52)↑}
		`age`: int; 0 ≤ `age` ≤ 120	{(−1)↓, (56)↑, (132)↓}
		`letter`:char;	{(J)↑, (3)↓}
String	At least one containing all legal strings and one containing all illegal strings. Legality is determined based on constraints on the length and other semantic features of the string	`fname`: string;	{(ϵ)↓, (Sue)↑, (Sue2)↓, (Too Long a name)↓}
		`vname`: string;	{(ϵ)↓, (shape)↑, (address1)↑), (Long variable)↓



		Example[a]	
Kind	Equivalence classes	Constraint	Class representatives[b]
Enumeration	Each value in a separate class	`auto_color` ∈ {red, blue, green}	{(red)↑, (blue)↑, (green)↑}
		up: boolean	{(true)↑, (false)↑}
Array	One class containing all legal arrays, one containing only the empty array, and one containing arrays larger than the expected size	`Java array`: int[] `aName` = new int[3];	{([])↓, {[−10, 20])↑, {[−9, 0, 12, 15])↓}

Compound data types – any input data value that has more than one independent attribute is a compound type. While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure.

Unidimensional versus Multi-Dimensional Partitioning

Unidimensional partitioning *(commonly used)*

- One way to partition the input domain is to consider one input variable at a time.
- Thus each input variable leads to a partition of the input domain.
- We refer to this style of partitioning as unidimensional equivalence partitioning

Multidimensional partitioning

- Another way is to consider the input domain ***I*** as the set product of the input variables and define a relation on ***I.***
- This procedure creates one partition consisting of several equivalence classes.
- We refer to this method as multidimensional equivalence partitioning

Example: consider the application that requires two integers input x and y. Each of these inputs is expected to lie in the following ranges

$3 \le x \le 7$ and $5 \le y \le 9$

For unidimensional partitioning, we apply the partitioning guidelines to x and y individually. This leads to six equivalence classes

E1: $x<3$	E4: $y<5$
E2: $3 \le x \le 7$	E5: $5 \le y \le 9$
E3: $x>7$	E6: $y>9$

For multidimensional partitioning, we consider the input domain to be the set product X×Y. This leads to 9 equivalence classes

E1: x<3, y<5	E5: 3≤ x ≤7, 5≤ y ≤9
E2: x<3, 5≤ y ≤9	E6: 3≤ x ≤7, y>9
E3: x<3, y>9	E7: x>7, y>5
E4: 3≤ x ≤7, y<5	E8: x>7, 5≤ y ≤9
E9: x>7, y>9	

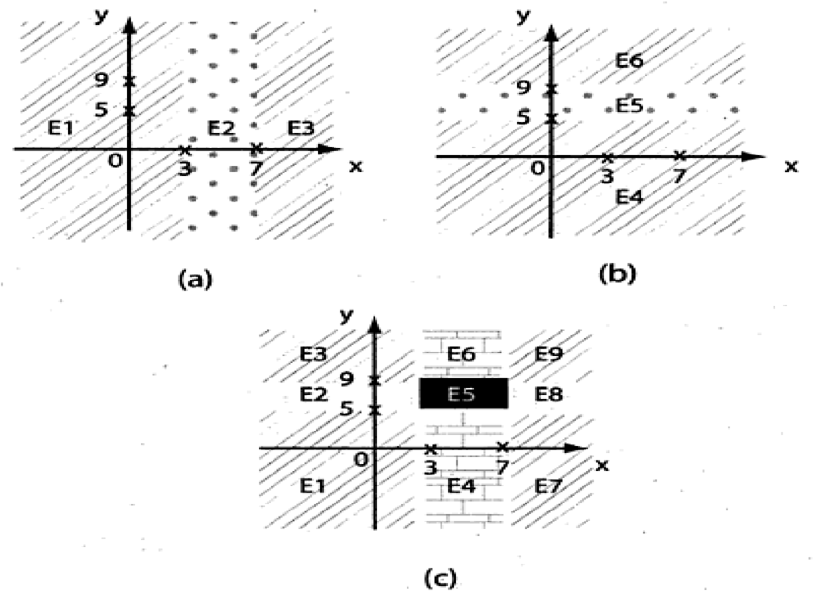


Figure: geometric representation of equivalence classes derived using uni-dimensional partitioning based on x and y in (a) and (b) respectively and using multi-dimensional partitioning as in (c)

Systematic Procedure for Equivalence Partitioning

1. *Identify the input domain:*
   Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use .Environment variables also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.
2. *Equivalence classing:*
   Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable is done based on the expected behaviour of the program.
   Values for which the program is expected to behave in the "same way" are grouped together. Note that the "same way" needs to be defined by the tester.
3. *Combine equivalence classes:*
   This step is usually omitted, and the equivalence classes defined for each variable are directly used to select test cases. However, by combining the equivalence classes, one misses the opportunity to generate useful tests.
   Eq. classes are combined using multidimensional approach.
4. *Identify infeasible equivalence classes:*
   An infeasible equivalence class is one that contains combination of input data that cannot be generated during test. Such an equivalence class may arrive due to several reasons.

Example: boiler control system (BCS)

- The control software (cs) is required to offer several options.
  One of the options, c (for control), is used by a human operator to give one of three commands (cmd).
  ➔ Change the boiler temperature (temp).
  ➔ Shut down the boiler (shut).
  ➔ Cancel the request (cancel).
- Command temp causes cs to ask the operator to enter the amount by which the temperature is to be changed (tempch) Values of tempch are in the range -10 to 10 in increments of 5 degrees Fahrenheit.

- Selection of option c forces the BCS to examine variable V. If V is set t GUI, the operator is asked to enter one of the three commands via GUI. However, if V is set to file, BCS obtains the command from a command line.

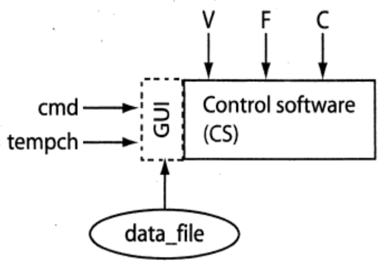


Fig. 2.5 Inputs for the boiler-control software. *V* and *F* are environment variables. Values of *cmd* (command) and *tempch* (temperature change) are input via the GUI or a data file depending on *V*. *F* specifies the data file.

- The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is temp. The filename is obtained from the variable
- Inputs for the boiler-control software. V and F are environment variables. Values of cmd (command) and tempch (temperature change) are input via the GUI or a data file depending on V. F specifies the data file.

★ *Identify the input domain:*

First we examine the requirements identify input variables, their types, and values.
These are listed below:

variable	kind	type	Value(s)
V	Environment	Enum	{ GUI, file }
F	Environment	String	A file name
cmd	Input via GUI or file	Enum	{ temp, cancel shut }
tempch	Input via GUI or file	enum	{ -10, -5, 5, 10 }

Therefore, domain subset of S= V×F×cmd×tempch
Eg: (GUI ,-, temp,-5)
(-) is Don't care

★ *Equivalence classing:*

Variable	partition
V	{{GUI}, {file}, {undefined}}
F	f_valid, f_invalid
cmd	{{temp}, {cancel}, {shut}, {c_invalid}}
tempch	{{t_valid}, {t_invalid}}

★ *Combine equivalence class:*

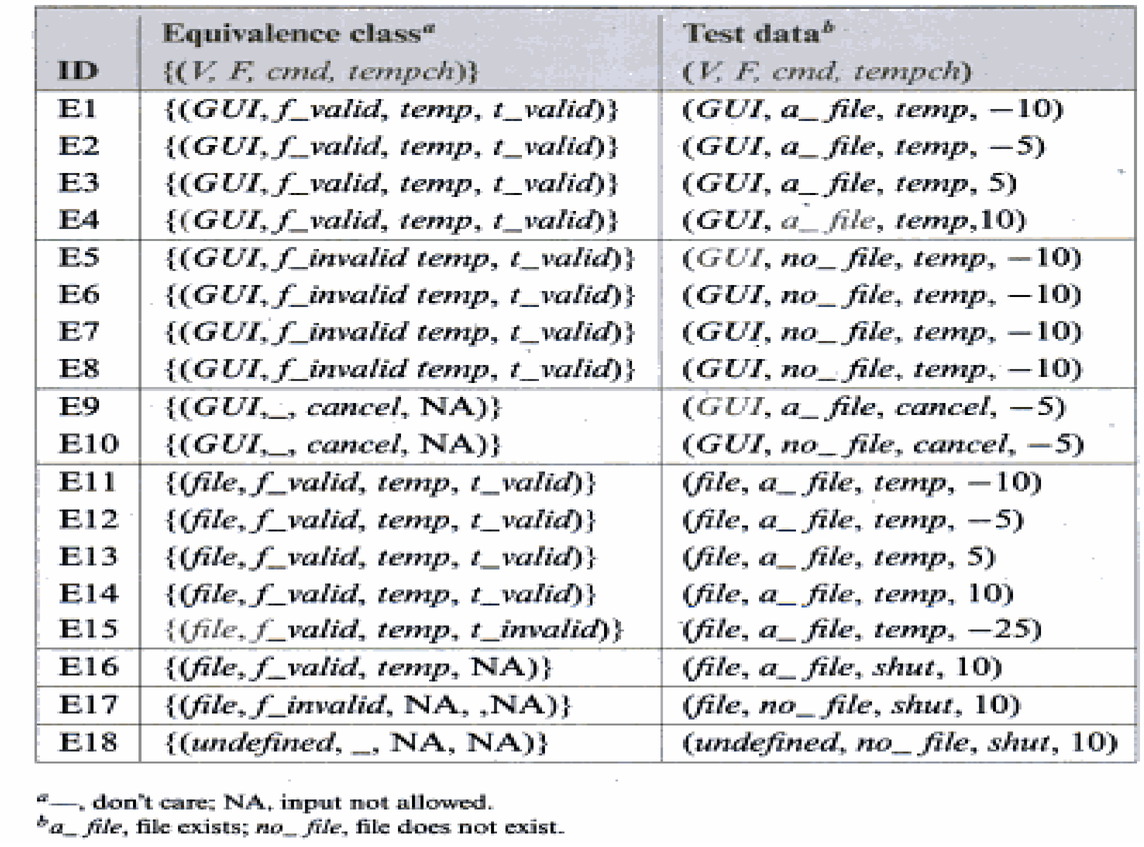
Note that tinvalid, tvalid, finvalid and fvalid denote sets of values. " undefined " denotes one value.

★ *Discard infeasible equivalence classes:*

Note that the GUI requests for the amount by which the boiler temp has to be changed only when the operator selects temp for cmd. Thus all eq. Classes that match the following template are infeasible.
{(V, F, {cancel, shut, cinvalid}, tvalid U tinvalid)}

Test Selection Based On Equivalence Classes

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values. In the most general case, a tester simply selects one test that serves as a representative of each equivalence class.

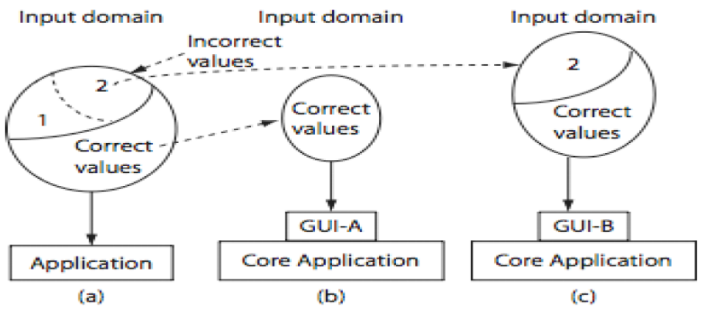


ID	Equivalence class[a] {(V, F, cmd, tempch)}	Test data[b] (V, F, cmd, tempch)
E1	{(GUI, f_valid, temp, t_valid)}	(GUI, a_file, temp, −10)
E2	{(GUI, f_valid, temp, t_valid)}	(GUI, a_file, temp, −5)
E3	{(GUI, f_valid, temp, t_valid)}	(GUI, a_file, temp, 5)
E4	{(GUI, f_valid, temp, t_valid)}	(GUI, a_file, temp,10)
E5	{(GUI, f_invalid temp, t_valid)}	(GUI, no_file, temp, −10)
E6	{(GUI, f_invalid temp, t_valid)}	(GUI, no_file, temp, −10)
E7	{(GUI, f_invalid temp, t_valid)}	(GUI, no_file, temp, −10)
E8	{(GUI, f_invalid temp, t_valid)}	(GUI, no_file, temp, −10)
E9	{(GUI,_, cancel, NA)}	(GUI, a_file, cancel, −5)
E10	{(GUI,_, cancel, NA)}	(GUI, no_file, cancel, −5)
E11	{(file, f_valid, temp, t_valid)}	(file, a_file, temp, −10)
E12	{(file, f_valid, temp, t_valid)}	(file, a_file, temp, −5)
E13	{(file, f_valid, temp, t_valid)}	(file, a_file, temp, 5)
E14	{(file, f_valid, temp, t_valid)}	(file, a_file, temp, 10)
E15	{(file, f_valid, temp, t_invalid)}	(file, a_file, temp, −25)
E16	{(file, f_valid, temp, NA)}	(file, a_file, shut, 10)
E17	{(file, f_invalid, NA, ,NA)}	(file, no_file, shut, 10)
E18	{(undefined, _, NA, NA)}	(undefined, no_file, shut, 10)

[a] —, don't care; NA, input not allowed.
[b] a_file, file exists; no_file, file does not exist.

GUI Design And Equivalence Classes

- ★ While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry.
- ★ For example, the requirement for an application. The application places a constraint on an input variable X such that it can assume integral values in the range 0..4. However, testing must account for the possibility that a user may inadvertently enter a value for X that is out of range.
- ★ Suppose that all data entry to the application is via a GUI front end. Suppose also that the GUI offers exactly five correct choices to the user for X. In such a situation it is impossible to test the application with a value of X that is out of range. Hence only the correct values of X will be input.



Restriction of the input domain through careful design of the GUI. Partitioning of the input domain into equivalence classes must account for the presence of GUI as shown in (b) and (c). GUI-A protects all variables against incorrect input while GUI-B does allow the possibility of incorrect input for some variables.

BOUNDARY VALUE ANALYSIS

- BVA is a test selection technique that targets faults in applications at the boundaries of equivalence classes.
- While equivalence partitioning selects tests from within equivalence classes, boundary-value analysis focuses on tests at and near the boundaries of equivalence classes.
- Certainly, tests derived using either of the two techniques may overlap.
- Once the input domain has been identified, test selection using boundary value analysis proceeds as follows:
    1. ***Partition the input domain using one-dimensional partitioning:*** This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning.
    2. ***Identify the boundaries for each partition****:* Boundaries may also be identified using special relationships among the inputs.
    3. ***Select test data such that each boundary value occurs in at least one test input***

BVA example

- Consider a method "fp" (find price) that takes two inputs –'code' and 'qty', both are integers.

1. Create equivalence classes

Assuming that an item code must be in the range 99 to 999 and qty in the range 1 to 100,

Equivalence classes for 'code'

E1: values less than 99
E2: values in the range
E3: values greater than 999

Equivalence classes for 'qty'

E4: values less than 1
E5: values in the range
E6: values greater than 100

2. Identify boundaries

Below fig shows equivalence classes and boundaries for (a) code and (b) qty. Values at and near the boundary are listed and marked with and "X" and "*" respectively.

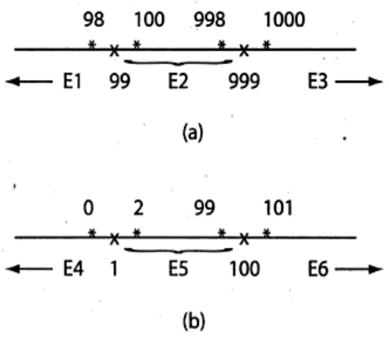


Fig. 2.7 Equivalence classes and boundaries for variables (a) *code* and (b) *qty* in Example 2.11. Values at and near the boundary are listed and marked with an "x" and "∗", respectively.

3. Construct test set

Test selection based on boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary.

Consider the following test set

```
T = { t₁ : (code = 98, qty = 0),
      t₂ : (code = 99, qty = 1),
      t₃ : (code = 100, qty = 2),
      t₄ : (code = 998, qty = 99),
      t₅ : (code = 999, qty = 100),
      t₆ : (code = 1000, qty = 101)
    }
```

➔ Consider the following faulty code skeleton for method “fp”

```
public void fP(int code, qty)
{
  if (code<99 && code>999)
   {display_error("Invalid code"); return;}
  // Validity check for  qty is missing !
  // Begin processing  code  and qty.
  :
}
```

➔ t1 and t6 tests indicate that value of ‘code’ is incorrect. But these two tests fails to check that the validity check on ‘qty’ is missing from the program.

➔ none of the other tests will be able to reveal the missing-code error. By separating the correct and incorrect values of different input variable we increase the possibility of detecting the missing-code error.

CATEGORY PARTITION METHOD

- Category partition method is a systematic approach to the generation of tests from requirements.
- The method consists of mix of manual and automated steps.
- Below fig shows the steps in the generation of tests using the category-partition method.
- Tasks in solid boxes are performed manually and generally difficult to automate.
- Dashed boxes indicate tasks that can be automated.

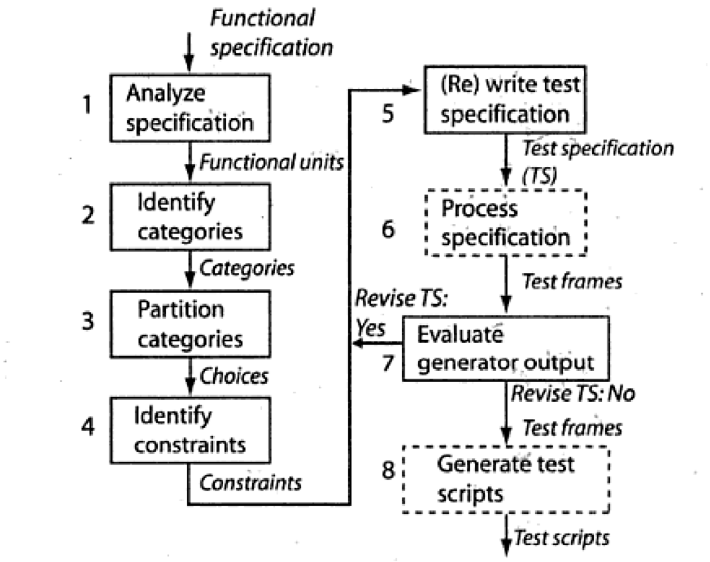


Fig. 2.9 Steps in the generation of tests using the category-partition method. Tasks in solid boxes are performed manually and generally difficult to automate. Dashed boxes indicate tasks that can be automated.

Step 1: analyse specification
→Here the tester identifies each functional unit that can be tested separately.

Step 2: identify categories
→For each testable unit, the given specification is analysed and the inputs isolated.
→Next e determine characteristics (or a category)of each parameter and environmental object.

Step 3: partition categories
→For each category, the tester determine different cases against which the functional unit must be tested.
→Each case is also referred to as a choice.
→One or more cases are expected for each category.

Step 4: identify constraints
→A constraint is specified using a property list and selector expression.
→Property lit has the following form:
[property p1,p2....]
where, property is the keyword and p1, p2... names of individual properties.
→A selector expression take one of the following forms.
[if p]
[if p and p2 and...]
Ex for special properties- [error] , [single]

Step 5: (Re) write test specification
→Tester now writes a complete test specification.
→The specification is written in a test specification language (TSL) conforming to a precise syntax.

Step 6: process specification
→TSL specification written in step 5 is processed by an automatic test-frame generator.
→This results in a number of test frames.
→The test frames are analysed by the tester for redundancy.

Step 7: Evaluate generator output
→Here tester examines the test frames for any redundancy or missing cases.
→This might lead to a modification in the test specification (step 5) and a return to step 6.

Step 8: Generate these scripts
→Test cases generated from test frames are combined into test scripts.
→A test script is a grouping of test cases.
→Generally, test cases that do not require any changes in settings of the environment objects are grouped together.
→This enables a test driver to efficiently execute the tests.

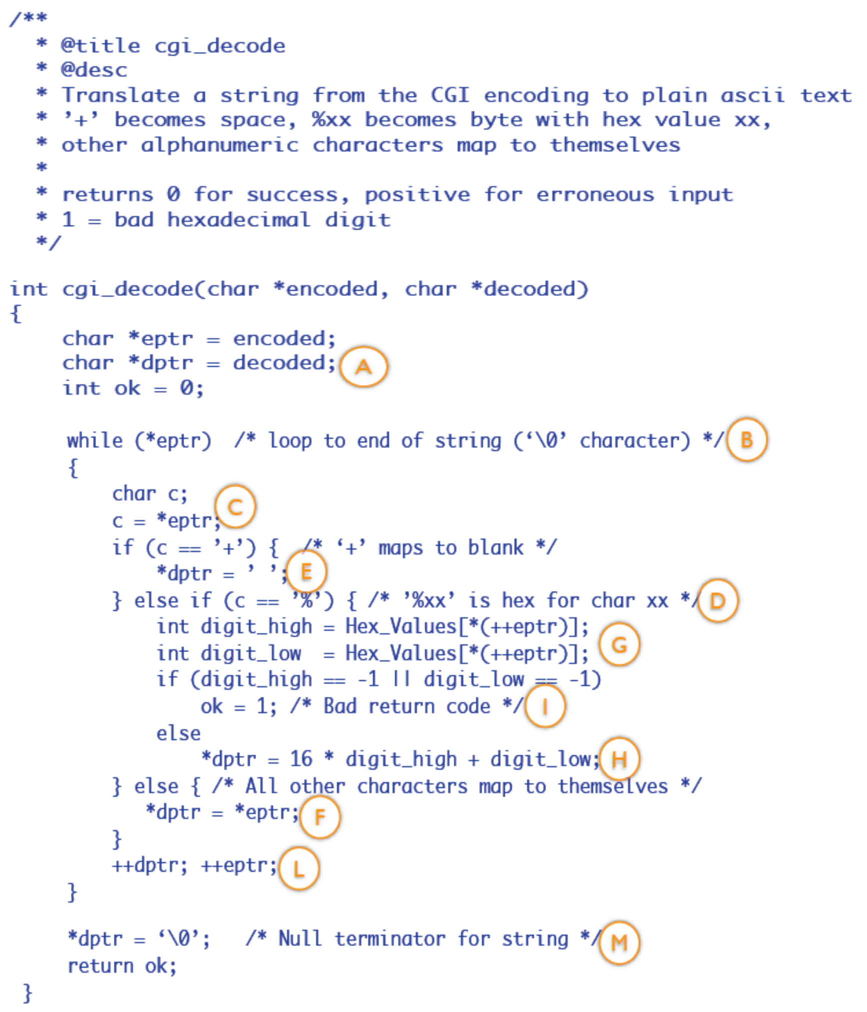
UNIT 3 QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the following i) Equivalence Partitioning ii)Boundary Value Analysis	June 10	4
2	Explain the steps associated in creating equivalence classes for the given problem requirements?	June 10	8
3	Identify the steps in generation of tests in category partition Method? Explain any two?	June 10	8
4	Describe the steps involved in a systematic procedure for equivalence partitioning by considering boiler control system as an example.	June 11	10
5	Explain the steps involved in the generation of tests using the category partition method with suitable examples.	June 11	10
6	What is Equivalence Partitioning? Explain the systematic procedure for Equivalence Partitioning by considering Boiler Control System Example.	Dec 11	10
7	What is boundary value analysis? Explain the procedure for BVA by considering your own example	Dec 11	10

UNIT 5
STRUCTURAL TESTING

OVERVIEW

- Testing can reveal a fault only when execution of the faulty element causes a failure
- Control flow testing criteria are defined for particular classes of elements by requiring the execution of all such elements of the program
- Control flow elements include statements, branches, conditions and paths.
- A set of correct program executions in which all control flow elements are exercised does not guarantee the absence of faults
- Execution of a faulty statement may not always result in a failure
- Control flow testing complements functional testing by including cases that may not be identified from specifications alone
- Test suites satisfying control flow adequacy criteria would fail in revealing faults that can be caught with functional criteria
- Example – missing path faults
- Control flow testing criteria are used to evaluate the thoroughness of test suites derived from functional testing criteria by identifying elements of the programs
- Unexecuted elements may be due to natural differences between specification and implementation, or they may reveal flaws of the software or its development process
- Control flow adequacy can be easily measured with automatic tools



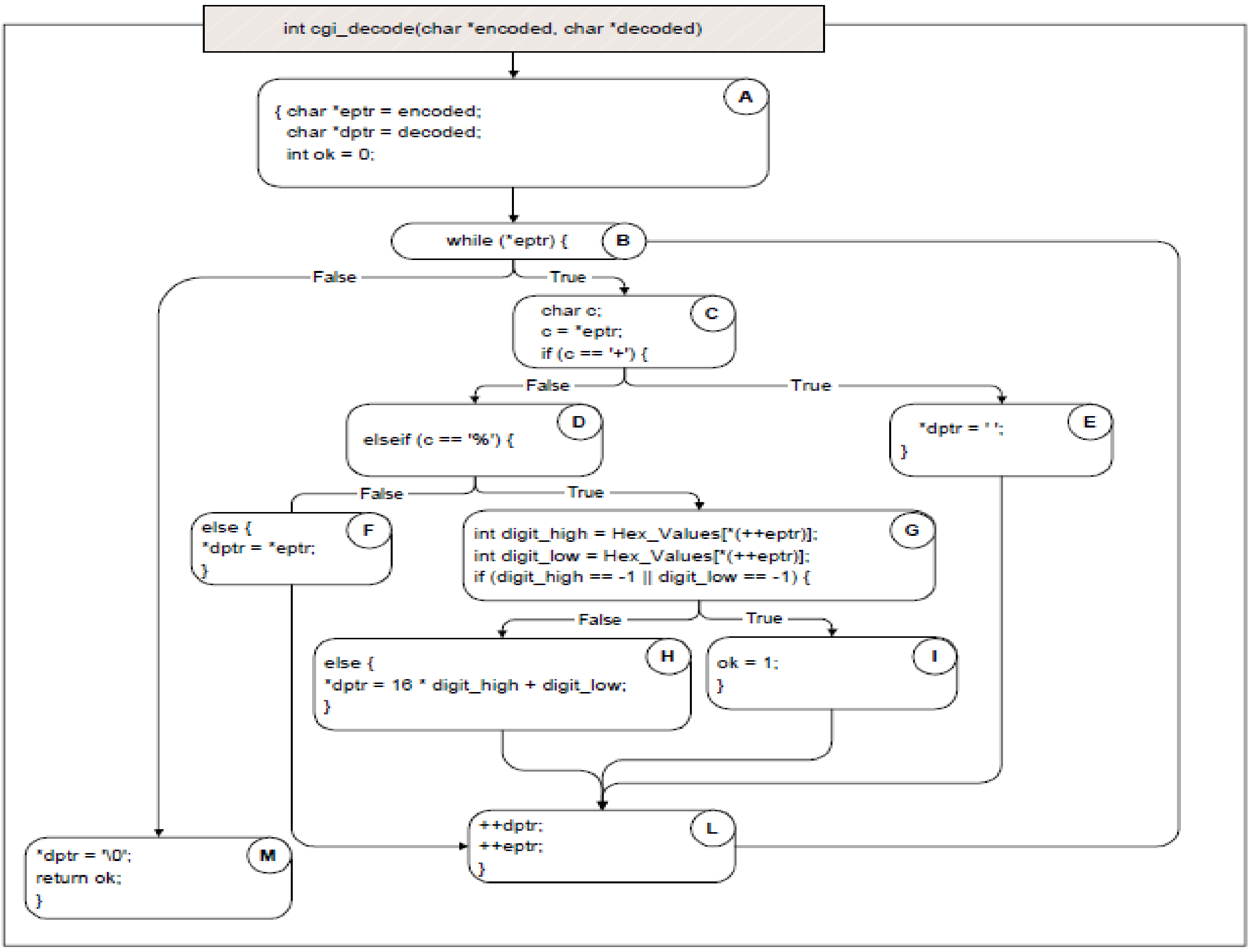
```
/**
 * @title cgi_decode
 * @desc
 * Translate a string from the CGI encoding to plain ascii text
 * '+' becomes space, %xx becomes byte with hex value xx,
 * other alphanumeric characters map to themselves
 *
 * returns 0 for success, positive for erroneous input
 * 1 = bad hexadecimal digit
 */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr)  /* loop to end of string ('\0' character) */
    {
        char c;
        c = *eptr;
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' ';
        } else if (c == '%') { /* '%xx' is hex for char xx */
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */
            else
                *dptr = 16 * digit_high + digit_low;
        } else { /* All other characters map to themselves */
            *dptr = *eptr;
        }
        ++dptr; ++eptr;
    }

    *dptr = '\0';    /* Null terminator for string */
    return ok;
}
```

Figure 5.1: The C function cgi decode, which translates a cgi-encoded string to a plain ASCII string (reversing the encoding applied by the common gateway interface of most web servers).



```
int cgi_decode(char *encoded, char *decoded)

A:  { char *eptr = encoded;
      char *dptr = decoded;
      int ok = 0;

B:  while (*eptr) {

C:  char c;
    c = *eptr;
    if (c == '+') {

D:  elseif (c == '%') {

E:  *dptr = ' ';
    }

F:  else {
    *dptr = *eptr;
    }

G:  int digit_high = Hex_Values[*(++eptr)];
    int digit_low = Hex_Values[*(++eptr)];
    if (digit_high == -1 || digit_low == -1) {

H:  else {
    *dptr = 16 * digit_high + digit_low;
    }

I:  ok = 1;
    }

L:  ++dptr;
    ++eptr;
    }

M:  *dptr = '\0';
    return ok;
    }
```

Figure 5.2: Control Flow graph of function cgi decode from previous Figure

$$
\begin{array}{lcl}
T_0 & = & \{ \text{`` '', ``test'', ``test+case\%1Dadequacy''} \} \\
T_1 & = & \{ \text{``adequate+test\%0Dexecution\%7U''} \} \\
T_2 & = & \{ \text{``\%3D'', ``\%A'', ``a+b'', ``test''} \} \\
T_3 & = & \{ \text{`` '', ``+\%0D+\%4J''} \} \\
T_4 & = & \{ \text{``first+test\%9Ktest\%K9''} \}
\end{array}
$$

Table 5.1: Sample test suites for C function cgi decode from Figure 5.1

STATEMENT TESTING

- Statements are nothing but the nodes of the control flow graph.
- Statement adequacy criterion:

  Let T be a test suite for a program P. T satisfies the statement adequacy criterion for P, iff, for each statement S of P, there exists at least one test case in T that causes the execution of S.

  This is equivalent to stating that every node in the control flow graph model of the program 1 is visited by some execution path exercised by a test case in T.

- Statement coverage:

  The statement coverage $C_{statement}$ of T for *P* is the fraction of statements of program P executed by at least one test case in T

$$Cstatement = \frac{number\ of\ executed\ statements}{number\ of\ statements}$$

T satisfies the statements adequacy criterion if $C_{statement} = 1$

- Basic block coverage: Nodes in a control flow graph often represent basic blocks rather than individual statements, and so some standards refers to basic coverage or node coverage
- Examples: in program 1, it contains
  - ➔ A test suite T_o = {" ","test","testcase%1Dadequacy }
    $Cstatement = \frac{17}{18}$ = 94% or node coverage $= \frac{10}{11}$ = 91%
    So it does not satisfy the statement adequacy criteria
  - ➔ A test suite T_1 ={"adequate + test%0Dexecution %TU"}
    $Cstatement = \frac{18}{18}$=1 or 100%

    So it satisfies the statement adequacy criterion
  - ➔ A test suite T_2={"%3D","%A","a+b","test"}
    $Cstatement = \frac{18}{18}$=1 or 100%

- ✓ Coverage is not monotone with respect to the size of the test suites, i.e., test suites that contain fewer test cases may achieve a higher coverage than test suites that contain more test cases.
- ✓ Criteria can be satisfied by many test suites of different sizes.
- ✓ A test suite with fewer test cases may be more difficult to generate or may be less helpful in debugging.
- ✓ Designing complex test cases that exercise many different elements of a unit is seldom a good way to optimize a test suite, although it may occasionally be justifiable when there is large and unavoidable fixed cost (e.g., setting up equipment) for each test case regardless of complexity.
- ✓ Control flow coverage may be measured incrementally while executing a test suite.
- ✓ The increment of coverage due to the execution of a specific test case does not measure the absolute efficacy of the test case.
- ✓ Measures independent from the order of execution may be obtained by identifying *independent statements*.

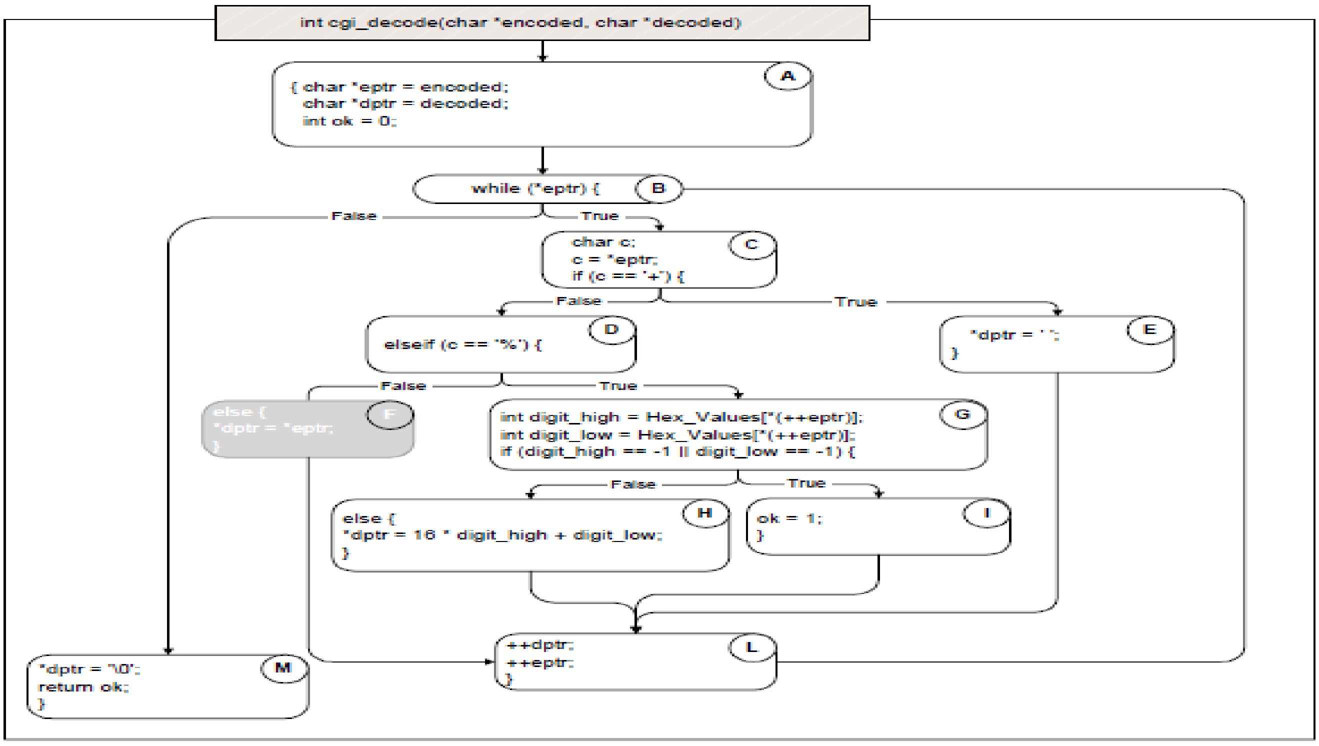


Figure 5.3: The control flow graph of function cgi decode which is obtained from the program of Figure 5.1 after removing node F.

BRANCH TESTING

- A test suite can achieve complete statement coverage without executing all the possible branches in a program.
- Consider, for example, a faulty program obtained from program by removing line 41.
- The control flow graph of program is shown in Figure 5.3.
- In the new program there are no statements following the false branch exiting node

- Branch adequacy criterion requires each branch of the program t be executed by at least one test case.
  Let T be a test suite for a program P. T satisfies the branch adequacy criterion for P. Iff, for each branch B of P, there exists at least one test case in T that causes the execution of B.
  This is equivalent to stating that every edge the control flow graph model of program P belongs to some execution path exercised by a test case in T
- The branch coverage C_{Branch} of T for P is the fraction of branches of program P executed by at least one test case in T
  $C_{Branch} = \frac{number\ of\ executed\ branches}{number\ of\ branches}$

T satisfies the branch adequacy criterion if C_{Branch} = 1
Examples:

- T_3={" "," +%0D+4J"}

100% statement coverage
88% branch coverage $C_{Branch} = \frac{7}{8} = 0.88$

- T_2={"%3D","%A","a+b","test"}
  100% statement coverage
  100% branch coverage
  $C_{Branch} = \frac{8}{8} = 1$

Test suite satisfies the branch adequacy criterion, and would reveal the fault. Intuitively, since traversing all edges of a graph causes all nodes to be visited, test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program.

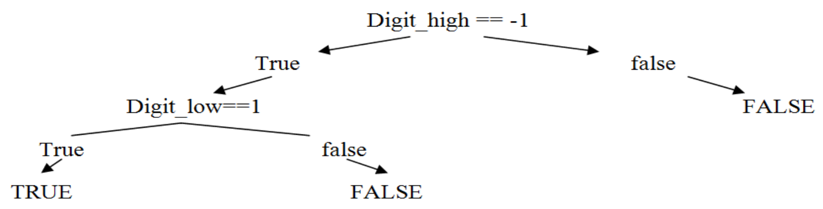
CONDITION TESTING

- Branch coverage is useful for exercising faults in the way a computation has been decomposed into cases. Condition coverage considers this decomposition in more detail, forcing exploration not only of both possible results of a boolean expression controlling a branch, but also of different combinations of the individual conditions in a compound boolean expression.
- Condition adequacy criteria overcome this problem by requiring different basic conditions of the decisions to be separately exercised.
- *Basic condition adequacy criterion*: requires each basic condition to be covered
  A test suite T for a program P covers all basic conditions of P, i.e. it satisfies the basic condition adequacy criterion, iff each basic condition in P has a true outcome in at least one test case in T and a false outcome in at least one test in T.

- *Basic condition coverage* ($C_{basic\_condition}$) of T for is the fraction of the total no. of truth values assumed by the basic in T $C_{basic\_condition} = \frac{total\ no.of\ truth\ values\ assumed\ by\ all\ basic\ conditions}{2 \times no.of\ basic\ conditions}$
- Basic conditions versus branches
  - Basic condition adequacy criterion can be satisfied without satisfying branch coverage
    - For ex: the test suite T_4= {“first + test %9ktet%k9”}
    - Satisfies basic condition adequacy criterion, but not the branch condition adequacy criterion. (Therefore the outcome of decision at line 27 is always false)
  - Thus branch and basic condition adequacy criterion are not directly comparable (neither implies the other)

Branch and condition adequacy criterion: A test suite satisfies the branch and condition adequacy criterion if it satisfies both the branch adequacy criterion and the condition adequacy criterion.
A more complete extension that includes both the basic condition and the branch adequacy criteria is the *compound condition adequacy criterion*, which requires a test for each possible combination of basic conditions.

For ex: the compound condition at line 27 would require covering the three paths in the following tree



Consider the number of cases required for compound condition coverage of the following two Boolean expressions, each with five basic conditions. For the expression `a && b && c && d && e`, compound condition coverage requires:

Test Case	a	b	c	d	e
(1)	**True**	**True**	**True**	**True**	**True**
(2)	**True**	**True**	**True**	**True**	**False**
(3)	**True**	**True**	**True**	**False**	–
(4)	**True**	**True**	**False**	–	–
(5)	**True**	**False**	–	–	–
(6)	**False**	–	–	–	–

MCDC

- An alternative approach that can be satisfied with the same number of test cases for boolean expressions of a given length regardless of short-circuit evaluation is the *modified condition adequacy criterion*, also known as *modified condition / decision coverage* or MCDC.
- Key idea: Test important combinations of conditions, avoiding exponential blowup
- A combination is “important” if each basic condition is shown to independently affect the outcome of each decision
- MC/DC can be satisfied with N+1 test cases, making it attractive compromise b/w no. of required test cases & thoroughness of the test

PATH TESTING

- Path adequacy criterion:
  A test suite T for a program P satisfies the path adequacy criterion iff, for each path p of P there exists at least one test case in T that causes the execution of p.
  This is equivalent to stating that every path in the CFG model of prog p is exercised by a test case in T
- Path coverage:
  The path coverage C_{path} of T for P is the fraction of path of program P executed by at least one test case in T

$C_{path} = \frac{no.of\ executed\ paths}{no.of\ paths}$

- Practical path coverage criteria
  - ➔ The no. of paths in a program with loops is unbounded, so the previously defined criterion cannot be satisfied for these programs. For program with loops, the denominator in the computation of path coverage is infinite, thus the path coverage becomes zero.
  - ➔ To obtain a practical criterion, it is necessary to partition the infinite set of path into a finite number of classes and require only that representatives from each class be explored.
  - ➔ Useful criteria can be obtained by
    - Limiting the no. of paths to be covered i.e.(no. of traversals of loops)
    - Limiting the length of the paths to be traversed
    - Limiting the dependencies among selected paths
- Boundary interior criterion groups together paths that differ only in the sub-path they follow when repeating the body of a loop.
- Fig below shows deriving a tree from CFG to derive sub-paths for boundary/interior testing
  (a) Is the CFG of come C function
  (b) Is a tree derived from (a) by following each path in the CFG up to the first repeated node. The set of paths from the root of the tree to each leaf is the required set of sub-paths for boundary/interior coverage.

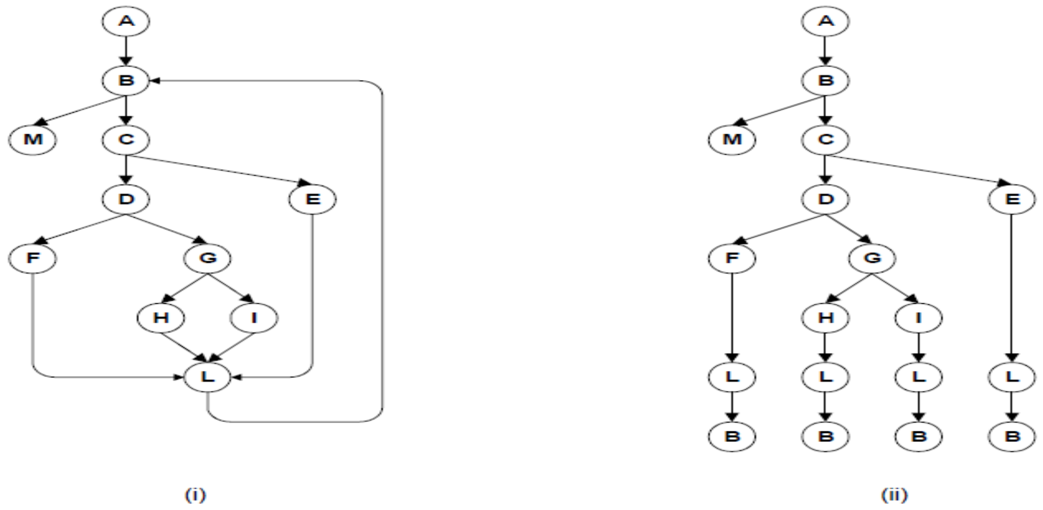


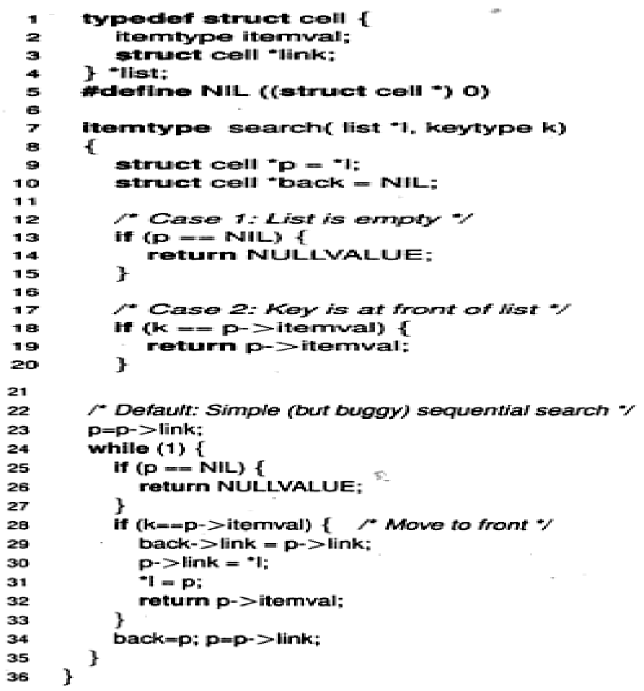
Figure 5.4: Deriving a tree from a control flow graph to derive sub-paths for boundary/interior testing. Part (i) is the control flow graph of the C function cgi decode, identical to Figure 14.1 but showing only node identifiers without source code. Part (ii) is a tree derived from part (i) by following each path in the control flow graph up to the first repeated node. The set of paths from the root of the tree to each leaf is the required set of sub-paths for boundary/ interior coverage.

- Limitations of boundary interior adequacy

```
If (a) {                    The sub-path through this control flow can include or
    S1;                     exclude each of the statements.
}
If (b) {                    Si, so that in total N branches result in 2^N paths that must be
      S2;                       traversed
}
If (c) {
S3;                         choosing input data to force execution of one particular path
}                            may be very difficult, or even impossible if the conditions
 ...                        are not independent
If (X){
```

Sn;
}

- Loop boundary adequacy criterion: it is a variant of boundary/interior criterion that treats loop boundaries similarly but is less stringent w.r.t. other differences among paths
  A test suite T for a program P satisfies the loop boundary adequacy criterion, iff, for each loop l in P.
  - ➔ In at least one execution, control reaches the loop, and then the loop control condition evaluated to False at the first time it is evaluated.
  - ➔ In at least one execution, control reaches the loop, and then the body of the loop is executed exactly once before control leaves the loop.
  - ➔ In at least one execution, the body of the loop is repeated more than once.
- Linear code sequence and a jump(LCSAJ) adequacy
  - ➔ LCSAJ is defined as a body of code through which the flow of control may proceed sequentially terminated by a jump in the control flow.
  - ➔ TER_1= statement coverage
  - ➔ TER_2= branch coverage
  - ➔ TER_{n+1}= coverage of n consequtive LCSAJs
- Cyclomatic testing:
  - ➔ Cyclomatic number is the number of independent paths in the CFG
    - o A path is representable as a bit vector, where each component of the vector represents an edge.
    - o "Dependence" is ordinary linear dependence b/w (bit) vectors
  - ➔ If e=number of edges,
    n=number of nodes,
    c=number of connected components of a graph,
    then,
    $$\text{cyclomatic number} = \begin{cases} e - n + c \text{ for any arbitrary graph} \\ e - n + 2 \text{ for a CFG} \end{cases}$$
  - ➔ Cyclomatic testing does not require that any particular basis set is covered. Rather it counts the number of independent paths that have actually been covered, and the coverage criterion is satisfied when this count reaches the cyclomatic complexity of the code under test.



```
typedef struct cell {
   itemtype itemval;
   struct cell *link;
} *list;
#define NIL ((struct cell *) 0)

itemtype  search( list *l, keytype k)
{
   struct cell *p = *l;
   struct cell *back = NIL;

   /* Case 1: List is empty */
   if (p == NIL) {
      return NULLVALUE;
   }

   /* Case 2: Key is at front of list */
   if (k == p->itemval) {
      return p->itemval;
   }

  /* Default: Simple (but buggy) sequential search */
  p=p->link;
  while (1) {
     if (p == NIL) {
        return NULLVALUE;
     }
     if (k==p->itemval) {   /* Move to front */
        back->link = p->link;
        p->link = *l;
        *l = p;
        return p->itemval;
     }
     back=p; p=p->link;
  }
}
```

Figure 5.5: A C function for ***searching and dynamically rearranging a linked list***, excerpted from a symbol table package. Initialization of the back pointer is missing, causing a failure only if the search key is found in the second position in the list.

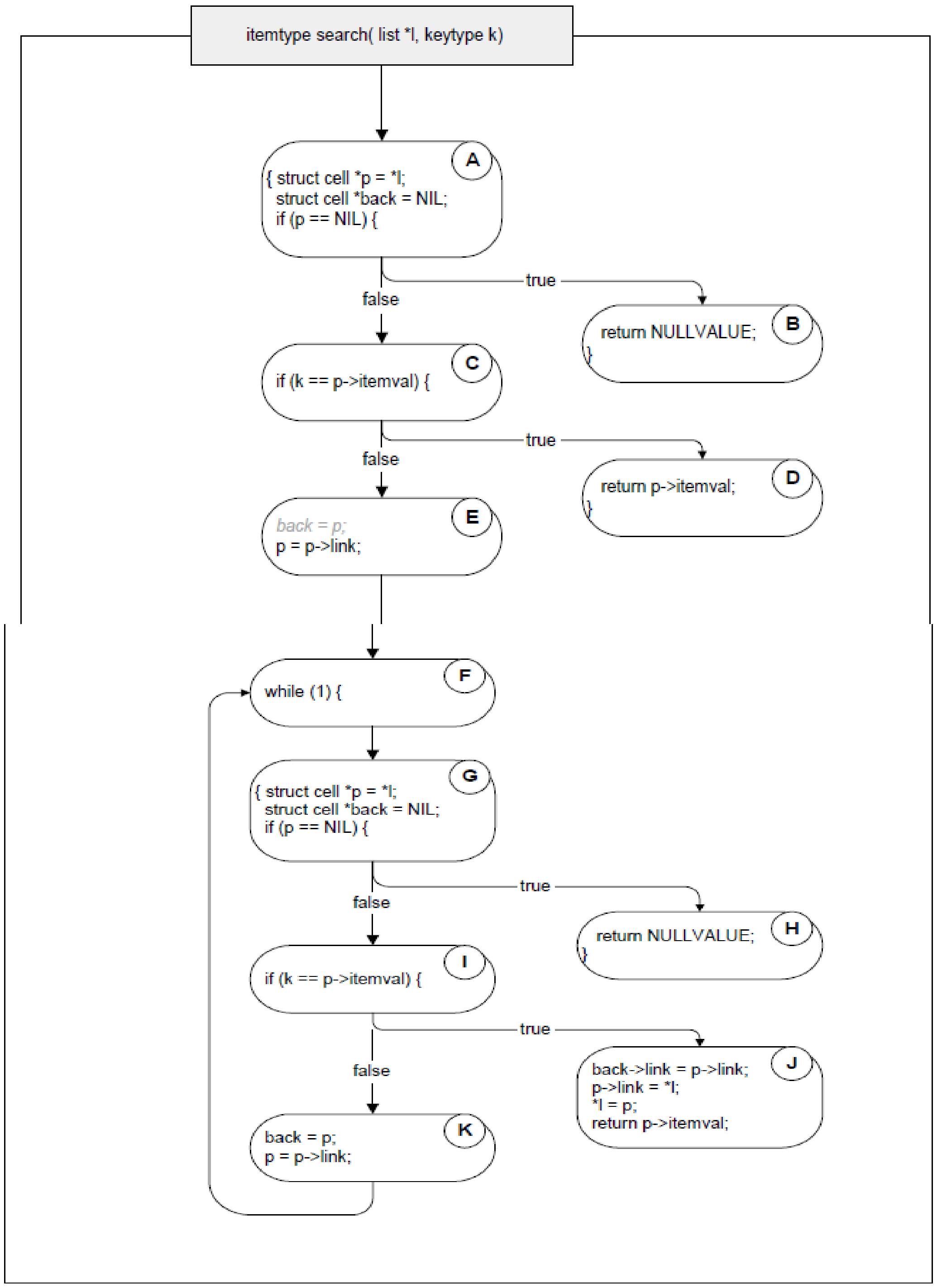


Figure 5.6: The control flow graph of C function search with move-to-front feature.

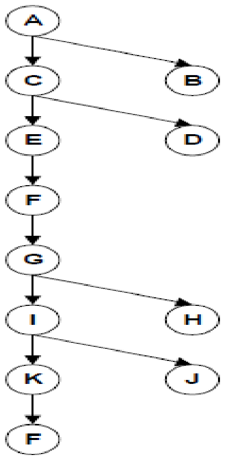


Figure 5.7: The boundary/interior sub-paths for C function search.

PROCEDURE CALL TESTING

- The criteria considered to this point measure coverage of control flow within individual procedures.
- They are not well suited to integration testing or system testing.
- Moreover, if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details.
- In some programming languages (FORTRAN, for example), a single procedure may have multiple entry points, and one would want to test invocation through each of the entry points.
- More common are procedures with multiple exit points.
- Exercising all the entry points of a procedure is not the same as exercising all the calls
- For example, procedure A may call procedure C from two distinct points, and procedure B may also call procedure C. In this case, coverage of calls of C means exercising all three of the points of calls.
- Commonly available testing tools can measure coverage of entry and exit points.
- Coverage of calls requires exercising each statement in which the parser and scanner access the symbol table, but this would almost certainly be satisfied by a set of test cases exercising each production in the grammar accepted by the parser.
- In object-oriented programming, local state is manipulated by procedures called *methods*, and systematic testing necessarily concerns sequences of method calls on the same object.

COMPARING STRUCTURAL TESTING CRITERIA

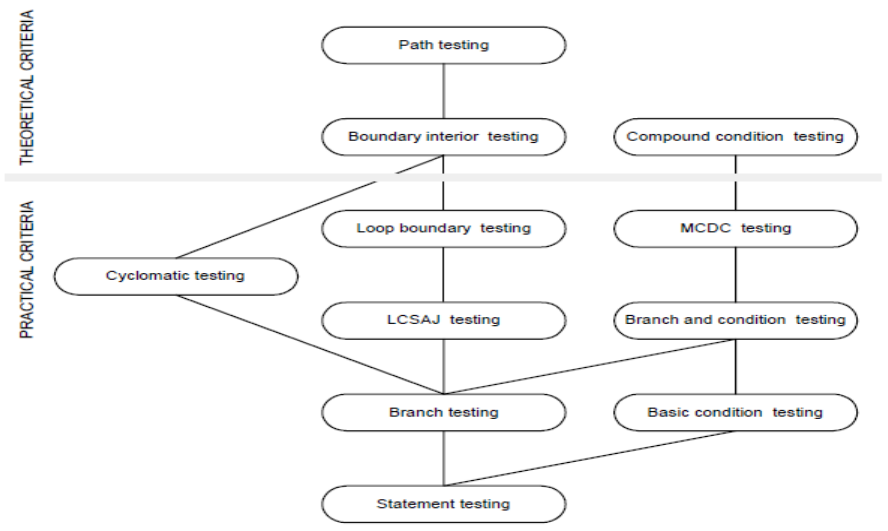


Figure 5.8: The subsumption relation among structural test adequacy criteria

- ♣ Power and cost of structural test adequacy criteria described earlier can be formally compared using the subsumes relation.
- ♣ The relations among these criteria are illustrated in the above figure.
- ♣ They are divide into two broad categories
  - Practical criteria
  - Theoretical criteria

  [Explain more looking at the diagram]

THE INFEASIBILITY PROBLEM

- Sometimes no set of test cases is capable of satisfying some test coverage criteria for a particular program, because the criterion requires the execution of a program elements that can never be executed
- Ex:
  - Execution of statements that cannot be executed as a result of
    - Defensive programming
    - Code reuse
  - Execution of conditions that cannot be satisfied as a result of interdependent conditions
  - Paths that cannot be executed as a result of interdependent decisions.
- Large amount of "fossil" code may indicate serous maintainability problems, but some unreachable code is common even in well designed well maintained systems.
- Solutions to the infeasibility problem
  - ➔ Make allowances for it by setting a coverage goal less than 100%
    Ex: 90% coverage of basic blocks, 10% allowance for infeasible blocks
  - ➔ Require justification of each element left uncovered. This approach is taken in some quality standards, like RTCA/DO-178B & EUROCAE ED-12B for MC/DC
- However, it is more expensive (because it requires manual inspection and understanding of each element left uncovered) and is unlikely to be cost-effective for criteria that impose test obligations for large numbers of infeasible paths.
- This problem, even more than the large number of test cases that may be required, leads us to conclude that stringent path-oriented coverage criteria are seldom useful.

UNIT 5 QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the branch testing, with an example.	June 10	4
2	Explain the following: i)procedure call testing ii)path testing	June 10	8
3	Explain in detail, condition testing and the infeasibility problem associated with it.	June 10	8
4	Describe the following with an example: i)Statement testing ii)Branch testing	June 11	10
5	Explain the path testing for C-function for searching to nearly and dynamically re-arranging a linked list. Also describe the control flow graph for the above C-function.	June 11	10
6	What is structural testing? Explain statement testing and branch testing with examples.	Dec 11	10
7	Distinguish between white box and black box testing categories.	Dec 11	4
8	What is path testing? Draw a flow graph for the biggest of three numbers program and calculate the cyclomatic complexity.	Dec 11	6

UNIT 7
TEST CASE SELECTION AND ADEQUACY, TEST EXECUTION

OVERVIEW

- The key problem in software testing is selecting and evaluating test cases
- Ideally we should like an "adequate" test suite to be one that ensures correctness of the product. Unfortunately, the goal is not attainable.
- The difficulty of proving that some set of test cases is adequate in this sense is equivalent to the difficulty of proving that the program is correct. In other words, we could have "adequate" testing in this sense only if we could establish correctness without any testing at all.
- So, in practice we settle for criteria that identify inadequacies in test suites.
- If no test in the test suite executes a particular program statement, we might similarly conclude that the test suite is inadequate to guard against faults in that statement.

TEST SPECIFICATIONS AND CASES

- A Test Case Includes input, the expected output, pass/fail criteria and the environment in which the test is being conducted. Here the term input means all the data that is required to make a test case.
- A Test Case specification is a requirement to be satisfied by one or more test cases.
- Specification-based testing uses the specification of the program as the point of reference for test input data selection and adequacy.
- A test specification can be drawn from system, interface or program.
- The distinction between a test case and test specification is similar to the distinction between program specification and program.
- Software Test cases derived from specifications of interface and programs are generally termed as glass box or white box testing.

Test cases should uncover errors like:

- ✓ Comparison of different data types
- ✓ Incorrect logical operators are precedence
- ✓ Expectation of equality when precision error makes equality unlikely
- ✓ Incorrect comparison or variables
- ✓ Improper or non-existent loop termination.
- ✓ Failure to exit when divergent iteration is encountered
- ✓ Improperly modified loop variables.

A test specification drawn from system, program and module interface specification often describes program inputs, but they can just as well specify any observable behavior that could appear in specifications.

Testing Terms

Test case
A test case is a set of inputs, execution conditions, and a pass/fail criterion.

Test case specification
A test case specification is a requirement to be satisfied by one or more actual test cases.

Test obligation

A test obligation is a partial test case specification, requiring some property deemed important to thorough testing. We use the term obligation to distinguish the requirements imposed by a test adequacy criterion from more complete test case specifications.

Test suite
A test suite is a set of test cases. Typically, a method for functional testing is concerned with creating a test suite. A test suite for a program, system, or individual unit may be made up of several test suites for individual modules, subsystems or features.

Test or test execution
We use the term test or test execution to refer to the activity of executing test cases and evaluating their results. When we refer to "a test", we mean execution of a single test case, except where context makes it clear that the reference is to execution of a whole test suite.

Adequacy criterion
A test adequacy criterion is a predicate that is true (satisfied) or false (not satisfied) of a {program, test suite} pair. Usually a test adequacy criterion is expressed in the form of a rule for deriving a set of test obligations from another artefact, such as a program or specification. The adequacy criterion is then satisfied if every test obligation is satisfied by at least one test case in the suite.

ADEQUACY CRITERIA

- Adequacy criteria are the set of test obligations. We will use the term test obligation for test specifications imposed by adequacy criteria, to distinguish them from test specifications that are actually used to derive test cases.
- Where do test obligations come from?
  → Functional (black box, specification based): from software specifications.
  → Structural (white or glass box): from code.
  → Model based: from model of system.
  → Fault based: from hypothesized faults (common bugs).
- A test suite satisfies an adequacy criterion if
  → All the tests succeed (pass).
  → Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
  Example: A statement coverage adequacy criterion is satisfied by a particular test suite for a program if each executable statement in the program is executed by at least one test case in the test suite.
- Satisfiability:
  ➔ Sometimes no test suite can satisfy criteria for a given program.
  Example: if the program contains statements that can never be executed, then no test suite can satisfy the statement coverage criterion.
- Coping with unsatisfiability:
  ➔ Approach 1: Exclude any unsatisfiable obligation from the criterion.
  Example: modify statement coverage to require execution only of statements that can be executed.
  But we can't know for sure which are executable.
  ➔ Approach 2: Measure the extent to which a test suite approaches an adequacy criterion.
  Example: If a test suite satisfies 85 of 100 obligations, we have reached 85% coverage.
    - A coverage measure is the fraction of satisfied obligations.
    - Coverage can be a useful indicator.
      - Of progress toward a thorough test suite
    - Coverage can also be a dangerous seduction
      - Coverage is only a proxy for thoroughness or adequacy.
      - It is easy to improve coverage without improving a test suite.
      - The only measure that really matters is (cost) effectiveness.

COMPARING CRITERIA

Empirical approach: would be based on extensive studies of the effectiveness of different approaches to testing in industrial practice, including controlled studies to determine whether the relative effectiveness of different testing method, depends on the kind of software being tested, the kind of organization in which the software is developed & tested, and a myriad of other potential confounding factors.
Analytical approach: answers to questions of relative effectiveness would describe conditions under which one adequacy criterion is guaranteed to be more effective than another, or describe in statistical terms their relative effectiveness.

→Analytic comparisons of the strength of test coverage depend on a precise definition of what it means for one criterion to be “stronger” or “more effective” than another.
→A test suite T_a that does not include all the test cases of another test suite T_b may fail revealing the potential failure exposed by the test cases that are in T_b but not in T_a.
→Thus, if we consider only the guarantees that a test suite provides, the only way for one test suite T_a to be stronger than another suite T_b is to include all test cases of T_b plus additional ones

→To compare criteria, then, we consider all the possible ways of satisfying the criteria.
→ if every test suite that satisfies some criterion A is a superset of some test suite that satisfies criterion B, or equivalently, every suite that satisfies A also satisfies B, then we can say that A “subsumes” B

The subsumes relation
A test adequacy, a subsumes test coverage criterion B iff, for every program P, every test set satisfying A wrt P also satisfies B wrt P

→Empirical studies of particular test adequacy criteria do suggest that there is value in pursuing stronger criteria, particularly when the level of coverage attained is very high.
→Adequacy criteria do not provide useful guarantees for fault detection, so comparing guarantees is not a useful way to compare criteria

TEST EXECUTION – OVERVIEW

- Test execution must be sufficiently automated for frequent re-execution without little human involvement
- The purpose of run-time support for testing is to enable frequent hands-free re-execution of a test suite.
- A large suite of test data may be generated automatically from a more compact and abstract set of test case specifications

FROM TEST CASE SPECIFICATION TO TEST CASES

- Test design often yields test case specifications, rather than concrete data.
- Example 1: “A large positive number”, not 420023
- Example 2: “a sorted sequence, length>2”, not “alpha, beta, chi, omega”
- A rule of thumb is that, while test case design involves judgement and creativity, test case generation should be a mechanical step.
- Automatic generation of concrete test cases from more abstract test case specifications reduce the impact of small interface changes in the course of development.
- Corresponding changes to the test suite are still required with each program change, but changes to test case specifications are likely to be smaller and more localized than changes to the concrete test cases.
- Instantiating test cases that satisfy several constraints may be simple if the constraints are independent, but becomes more difficult to automate when multiple constraints apply to the same item.

SCAFFOLDING

- Code developed to facilitate testing is called scaffolding, by analogy to the temporary structures erected around a building during construction or maintenance.
- Scaffoldings may include
  →Test drivers (substituting for a main or calling population)
  →Test harness (substituting for parts of the deployment environment)
  →Stubs (substituting for functionally called or used by the software under test)
- The purpose of scaffolding is to provide controllability to execute test cases and observability to judge the outcome of test execution.
- Sometimes scaffolding is required to simply make module executable, but even in incremental development with immediate integration of each module, scaffolding for controllability and observability may be required because the external interfaces of the system may not provide sufficient control to drive the module under test through test cases, or sufficient observability of the effect.
- Example: consider an interactive program that is normally driven through a GUI. Assume that each night the person goes through a fully automate and unattended cycle of integration compilation, and test execution.
- It is necessary to perform some testing through the interactive interface, but it is neither necessary nor efficient to execute all test cases that way. Small driver programs, independent of GUI can drive each module through large test suites in a short time.

GENERIC VERSUS SPECIFIC SCAFFOLDING

How general should scaffolding be? To answer

- We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit)
- ... or further factor out some common support code, to drive a large number of test cases from data... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
- Fully generic scaffolding may suffice for small numbers of hand-written test cases
- The simplest form of scaffolding is a driver program that runs a single, specific test case.
- It is worthwhile to write more generic test drivers that essentially interpret test case specifications.
- A large suite of automatically generated test cases and a smaller set of handwritten test cases can share the same underlying generic test scaffolding
- Scaffolding to replace portions of the system is somewhat more demanding and again both generic and application-specific approaches are possible
- A simplest stub – ***mock*** – can be generated automatically by analysis of the source code
- The balance of quality, scope and cost for a substantial piece of scaffolding software can be used in several projects
- The balance is altered in favour of simplicity and quick construction for the many small pieces of scaffolding that are typically produced during development to support unit and small-scale integration testing
- A question of costs and re-use – Just as for other kinds of software

TEST ORACLES

- In practice, the pass/fail criterion is usually imperfect.
- A test oracle may apply a pass/fail criterion that reflects only a part of the actual program specification, or is an approximation, and therefore passes some program executions it ought to fail
- Several partial test oracles may be more cost-effective than one that is more comprehensive
- A test oracle may also give false alarms, failing an execution that is ought to pass.
- False alarms in test execution are highly undesirable.
- The best oracle we can obtain is an oracle that detects deviations from expectation that may or may not be actual failure.

Two types

- **Comparison based oracle**

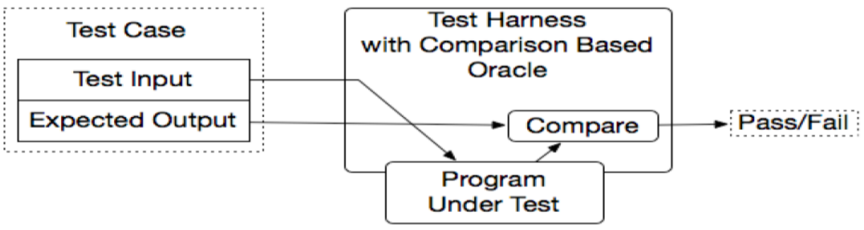


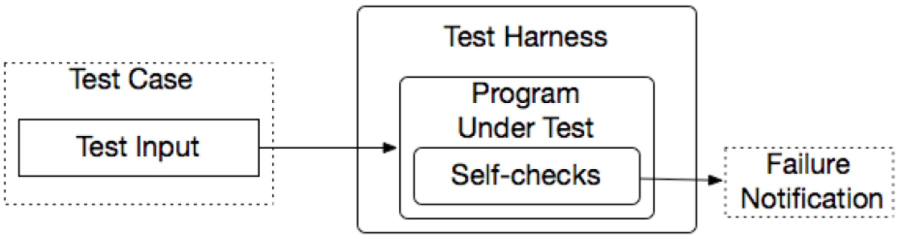
Fig: a test harness with a comparison based test oracle processes test cases consisting of (program input, predicted output) pairs.

- With a comparison based oracle , we need predicted output for each input
- Oracle compares actual to predicted output, and reports failure if they differ.
- It is best suited for small number of hand generated test cases example: for handwritten Junit test cases.
- They are used mainly for small, simple test cases
- Expected outputs can also be produced for complex test cases and large test suites
- Capture-replay testing, a special case in which the predicted output or behavior is preserved from an earlier execution
- Often possible to judge output or behavior without predicting it

- **Partial oracle**
  - Oracles that check results without references to predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others.
  - They check necessary but not sufficient conditions for correctness.
  - A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained
  - Specifications are often incomplete
  - Automatic derivations of test oracles are impossible

SELF-CHECKS AS ORACLES

- An oracle can also be written as self checks
  -Often possible to judge correctness without predicting results.
- Typically these self checks are in the form of assertions, but designed to be checked during execution.
- It is generally considered good design practice to make assertions and self checks to be free of side effects on program state.
- Self checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specification rather than all program behaviour.
- Devising the program assertions that correspond in a natural way to specifications poses two main challenges:
  - Bridging the gap between concrete execution values and abstractions used in specification
  - Dealing in a reasonable way with quantification over collection of values



- Structural invariants are good candidates for self checks implemented as assertions
- They pertain directly to the concrete data structure implementation
- It is sometimes straight-forward to translate quantification in a specification statement into iteration in a program assertion
- A run time assertion system must manage ghost variables
- They must retain "before" values
- They must ensure that they have no side effects outside assertion checking
- *Advantages:*
  -Usable with large, automatically generated test suites.
- *Limits:*
  -often it is only a partial check.
  -recognizes many or most failures, but not all.

CAPTURE AND REPLAY

- Sometimes it is difficult to either devise a precise description of expected behaviour or adequately characterize correct behaviour for effective self checks.
  Example: even if we separate testing program functionally from GUI, some testing of the GUI is required.
- If one cannot completely avoid human involvement test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error.
- The principle is simple:
  The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated testing.
- The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it, before it is invalidated by some change to the program.
- Mapping from concrete state to an abstract model of interacting sequences is some time possible but is generally quite limited.

No.	QUESTION	YEAR	MARKS
1.	Explain the following: i)Test Case ii)Test case Specification iii)Test Suite iv)Adequacy Criteria.	June 10	4
2.	Explain in detail, the scaffolding and test oracles, with reference to test execution.	June 10	8
3.	Discuss: i)Test case specification to test cases ii)capture and replay.	June 10	8
4.	Explain the adequacy criteria.	June 11	8
5.	Describe the test oracles with a neat diagram.	June 11	8
6.	What is scaffolding? Explain.	June 11	4
7.	Define the following testing terms: i)Test case ii)Test case specification iii)Test obligation iv)Test suite v)Smoke testing.	Dec 11	10
8.	What is scaffolding? Distinguish between generic and specific scaffolding. Briefly explain the differences.	Dec 11	10