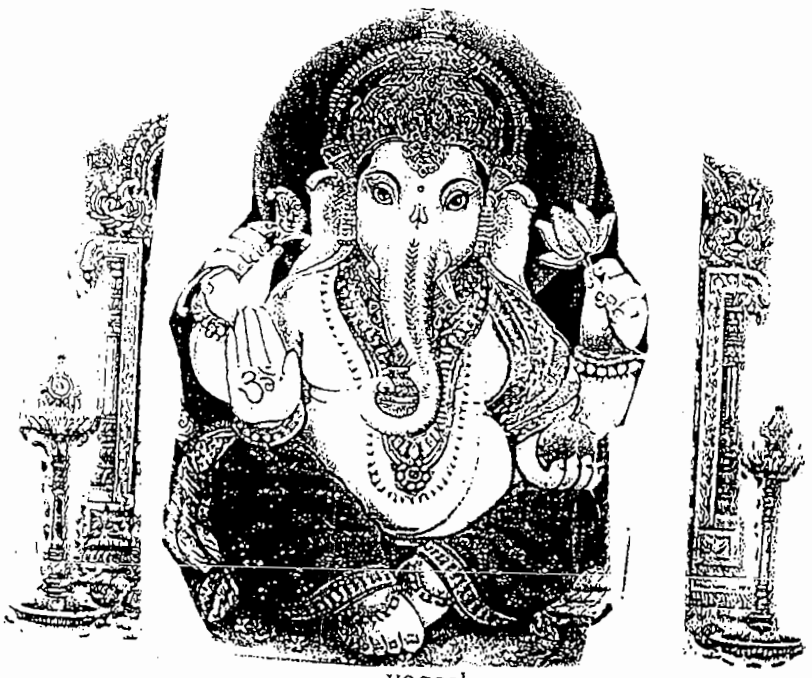
550=00

S.T



yogesh

Dealer:

SRI VENKATESHWARA XEROX CENTER

Contact:
VENKATESH
Mob: 9448926729

ENGINEERING NOTES FOR ALL SUBJECTS (ALL SEM AND ALL BRANCHES) ARE AVAILABLE IN THIS SHOP

T.S

8TH SEM CSE/ISE

SOFTWARE TESTING

06IS81

ASHOK KUMAR K
VIVEKANANDA INSTITUTE OF TECHNOLOGY
Mob: 9742024066
e-mail: celestialcluster@gmail.com

S.V. XEROX
No. 34/A, Near RNS IT College,
Uttarahalli-Kengeri Main Road,
Channasandra, Bengaluru - 560 061.
Mob: 9611148853, 9886552700

Dealer:

SRI VENKATESHWARA XEROX CENTER

Contact:
VENKATESH
Mob: 9448926729

ENGINEERING NOTES FOR ALL SUBJECTS (ALL SEM AND ALL BRANCHES) ARE AVAILABLE IN THIS SHOP

SPECIAL THANKS

I would also like to thank some of the students for their valuable feedback on my previous notes.

RAJESH RN, *JVIT, Ramanagaram*
SHWETHA, *DBIT, Bangalore*
JANU KHANDARI, *SKIT, Bangalore*
BIPIN, *JSSATE, Bangalore*
SHILPA, *SRSIT, Bangalore*
LAKSHMI, *SKIT, Bangalore*
MANASA, *JSSATE, Bangalore*
RAMYA, *RNSIT, Bangalore*
BRADLEY, *Mangalore*
SUMANTH, *JSSATE, Bangalore*
APEKSHA, *RNSIT, Bangalore*
SRIKANTH, *Tumkur*
RUKMINI, *Hassan*
KARTHIK RAO, *PESSE, Bangalore*
NISHITHA, *PESCE, Mandya*
SHIVU, *BNMIT, Bangalore*
KARTHIKA, *AMCEC, Bangalore*
BHARATH, *JVIT, Ramanagaram*
JAGANNATH, *BMS Evening College, Bangalore*
HEMANTH, *Shimoga*
NACHAPPA, *JSSATE, Bangalore*
PRADEEP, *DSCE, Bangalore*
DARSHINI, *JVIT, Ramanagaram*
MRUDULA, *GAT, Bangalore*
DEEPAK, *RNSIT, Bangalore*
SUHAS K.M, *Bellary*
JAYPRADEEP, *Sapthagiri College of Engineering, Bangalore*
ASHRITHA ALVA, *NMIT, Bangalore*
SANDEEP PAI, *HKBKCE, Bangalore*
HARISH, *Chikmaglur*
GIRISH, *JVIT, Ramanagaram*

Dedicated To:

DEEPIKA N

Sri Venkateshwara College of Engineering, Bangalore

NAYANA M C

Sri Venkateshwara College of Engineering, Bangalore

DEEPIKA C

Global Academy of Technology, Bangalore

MALA B C

BGSIT, Nagamangala, Mandya

UNIT 1:

BASICS OF SOFTWARE TESTING - 1

HUMANS, ERRORS, AND TESTING

- Errors are a part of our daily life. Humans make errors in their thoughts, actions, and in the products that might result from their actions.
- Humans make errors in any field, for ex in observation, in speech, in medical prescription, in surgery, in love, and similarly in software development.

Area	Error
Hearing	Spoken: He has a garage for repairing *foreign* cars. Heard: He has a garage for repairing *falling* cars.
Medicine	Incorrect antibiotic prescribed.
Music performance	Incorrect note played.
Numerical analysis	Incorrect algorithm for matrix inversion.
Observation	Operator fails to recognize that a relief valve is stuck open.
Software	Operator used: $\neq$, correct operator: $>$. Identifier used: new_line, correct identifier: next_line. Expression used: $a \wedge (b \vee c)$, correct expression: $(a \wedge b) \vee c$. Data conversion from 64-bit floating point to 16-bit integer not protected (resulting in a software exception).
Speech	Spoken: *maple walnut*, intent: *maple walnut*. Spoken: *We need a new refrigerator*, intent: *We need a new washing machine*.
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of *pans* did you use? Intent: What kind of *pants* did you use?

Table: Examples of errors in various fields of human endeavor.

- To determine whether there are any errors in our thought, actions, and the process generated, we resort to the process of **testing**. The primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is they conform to the requirements.
  - Testing of thoughts is usually designed to determine if a concept or method has been understood satisfactorily.

ASHOK KUMAR K
mob: 9742024066
Email: celestialcluster@gmail.com

- Testing of actions is designed to check if a skill that results in the actions has been acquired satisfactorily.
- Testing of a product is designed to check if the product behaves as desired.

Errors, Faults, and Failures

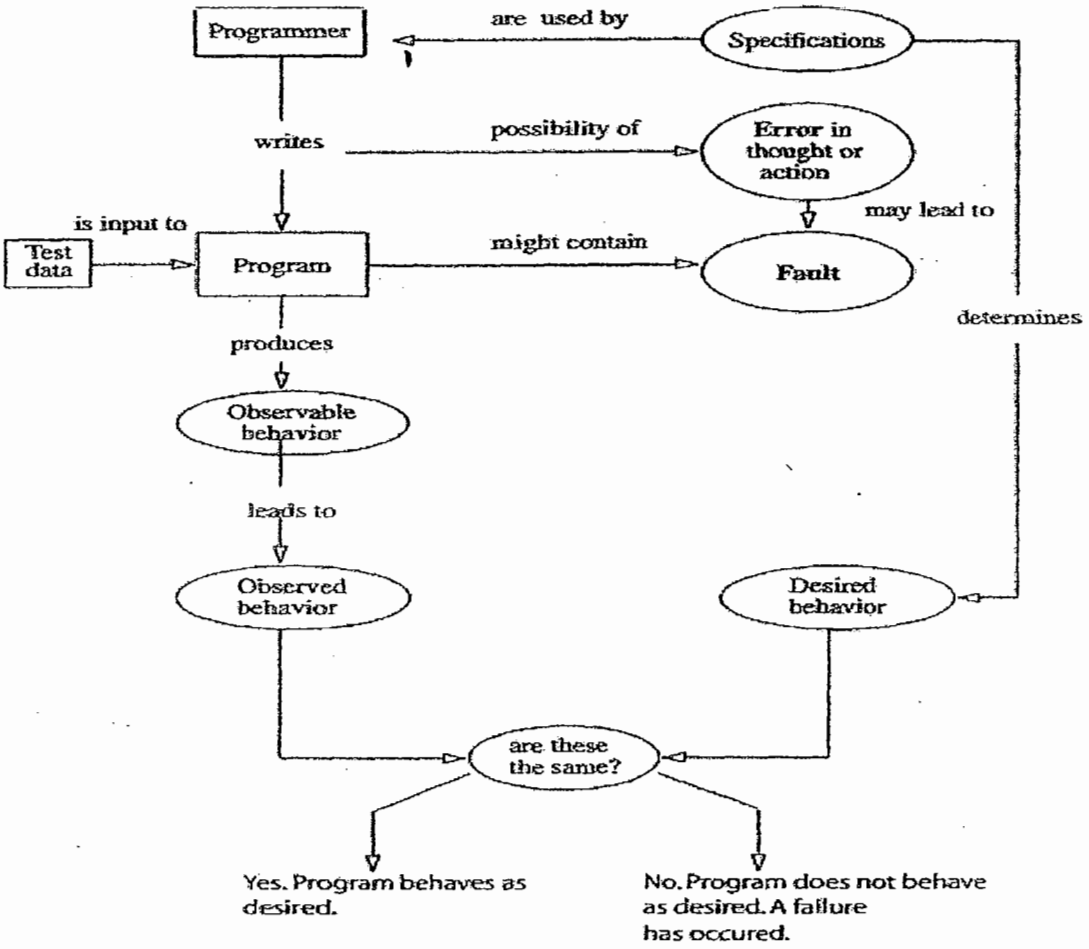


Figure: Errors, Faults, and Failures in the process of programming and testing.

- **Error:** Programmer writes a program. An error occurs in the process of writing a program.
- **Fault (or bug or defect):** It is the manifestation of one or more errors.

- **Failure**: Failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to the program's output.

Test Automation

- Execution of many tests can be tiring as well as error-prone. Hence there is a tremendous need for automating testing tasks.
- Many Software development organizations automate test-related tasks such as regression testing, graphical user interface testing, I/O device driver testing
- Tools used:
  - **GUI testing**: Eggplant, Marathon, & pounder
  - **Performance or load testing**: eLoadExpert, DBMonster,JMeter, Dieseltest,WAPT, LoadRunner, Grinder
  - **Regression testing**: Echelon, TestTube, WinRunner, XTest
- **AETG** is an automated test generator that can be used in variety of applications.

Developer and tester as two roles

- A tester refers to the role someone assumes when testing a program. Such an individual could be a developer testing a class he/she has coded, or a tester who is testing a fully integrated set of components.
- A programmer refers to an individual who engages in software development and often assumes the role of a tester, at least temporarily.

SOFTWARE QUALITY

- Software quality is a multidimensional quantity and is measurable.

Quality Attributes

- There exist several measures of software quality. These can be divided into static and dynamic quality attributes.
- Static quality attributes: Refers to actual code and related documentation. It includes structured, maintainable, testable code as well as the availability of correct and complete documentation.

- Dynamic quality attributes: Relate to the behavior of the application while in use. It includes software reliability, correctness, completeness, consistency, usability, and performance.
  - **Reliability**: refers to the probability of failure-free operation.
  - **Correctness**: refers to the correct operation of an application and is always with reference to some artifact.
  - **Completeness**: refers to the availability of all the features listed in the requirements or in the user manual.
  - **Consistency**: refers to adherence to a common set of conventions and assumptions.
  - **Usability (User-centric testing)**: refers to the ease with which an application can be used.
  - **Performance**: refers to the time the application takes to perform a requested task. It is considered as a non-functional requirement. It is specified in terms such as "This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory".

Reliability

- There are several definitions of software reliability:
- **Definition 1 (ANSI/IEEE STD 729-1983)**: Software Reliability is the probability of failure-free operation of software over a given time interval and under given conditions.
  - Adv: Accurate estimate of the reliability can be found.
  - Disadv: requires the knowledge of the profile of its users that might be difficult or impossible to estimate accurately.
- **Definition 2**: Software Reliability is the probability of failure free-operation of software in its intended environment.
  - Adv: one needs only a single number to denote reliability of a software application that is applicable to all its users.
  - Disadv: Such estimates are difficult to arrive at.

REQUIREMENTS, BEHAVIOR, AND CORRECTNESS

- Products, software in particular, are designed in response to requirements. Requirements specify the functions that a product is expected to perform. Of course, during the development of the product, the requirements might have changed from what was stated originally.
- Example: Two requirements are given below, each of which leads to a different program.

  Requirement 1: It is required to write a program that inputs two integers 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.
- Here, Requirement 1 is incomplete and Requirement 2 is ambiguous.

  Proof:

  Requirement 1: Incomplete

  Suppose that program *max* is developed to satisfy Requirement 1. 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, or on two separate lines with a carriage return typed in after each number. The requirements are stated above fails to provide an answer to this question. This example illustrates the incompleteness Requirement 1.

  Requirement 2: ambiguous

  It is not clear from this requirement whether the input sequence is to be sorted in ascending or in descending order. The behavior of *sort* program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing *sort*.
- Testers are often faced with incomplete and/or ambiguous requirements.

Input domain and program correctness

- A program is considered correct if it behaves as desired on all possible test inputs.
- Testing a program on all possible inputs is known as **exhaustive testing.**
- The set of all possible inputs to a program P is known as the **input domain** or input space, of P.

- Example:
  - Using Requirement 1 above we find the input domain of *max* to be the set of all pairs of integers where each element in the pair integers is in the range 32,768 till 32,767.
  - Using Requirement 2 it is not possible to find the input domain for the *sort* program. Therefore, we will modify this requirement as follows.

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 input first followed by the sequence of integers to be sorted; the sequence is terminated with a period.

- Based on the above modified requirement, the input domain for *sort* is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period.

<A -3 15 12 55.>

<D 23 77.>

- Definition of Program correctness: A program is considered correct if it behaves as expected on each element of its input domain.

Valid and Invalid Inputs

- The modified requirement for *sort* mentions that the request characters can be "A" and "D", but fails to answer the question **"What if the user types a different character?"**
- When using *sort* it is certainly possible for the user to type a character other than "A" and "D". Any character other than "A" and "D" is considered as invalid input to *sort*. The requirement for *sort* does not specify what action it should take when an invalid input is encountered.
- Testing a program against invalid inputs might reveal errors in the program. Example: Suppose we are testing the *sort* program. We execute it against the following input: <E 7 19.>. Suppose that upon execution on the above input, the *sort* program enters into an infinite loop and neither asks the user for any

input nor responds to anything typed by the user. This observed behavior points to a possible error in sort.

- Instead of typing an integer the user types in a character such as "?", program should inform the user that the input is invalid.
- Input domain is partitioned into two sub domains: **valid and invalid inputs.**

Example:

<D 7 9F 19.> ---------- Invalid

<A 7 19.> ---------- Valid

CORRECTNESS VERSUS RELIABILITY

Correctness

- 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. In most cases that are encountered in practice, this 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.

Testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.

- Example: This example illustrates why the probability of program failure might not change upon error removal.

```
integer x, y
input x, y
if(x < y) <---- This condition should be x<=y
{ print f(x, y) }
else
{ print g(x, y) }
```

Reliability

- Reliability of a program P is the probability of its successful execution on a randomly selected element from its input domain
- 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; its reliability can be anywhere between 0 and 1.
- Example: How to compute program reliability in a simplistic manner.
  Consider a program P which takes a pair of integers as input. The input domain of this program is the set of all pairs of integers. Suppose now that in actual use there are only three pairs that will be input to P. These are as follows:
  { < (0,0) (-1,1) (1, -1) > }. If it is known that P fails on exactly one of the three possible input pairs then the frequency with which P will function correctly is 2/3. This number is an estimate of the probability of the successful operation of P and hence is the reliability of P.

Program use and the operational profile

- 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.
- Consider a *sort* program which, on any given execution, allows any one of two types of input sequences. Sample operational profiles for *sort* is specified as follows:

Operational profile #1		Operational profile #2	
Sequence	Probability	Sequence	Probability
Numbers only	0.9	Numbers only	0.1
Alphanumeric strings	0.1	Alphanumeric strings	0.9

Operational profile 1 is used for sorting sequences of numbers
Operational profile 2 is used for sorting alphanumeric strings

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 below figure, testing and debugging are often used as two related activities in a cyclic manner.

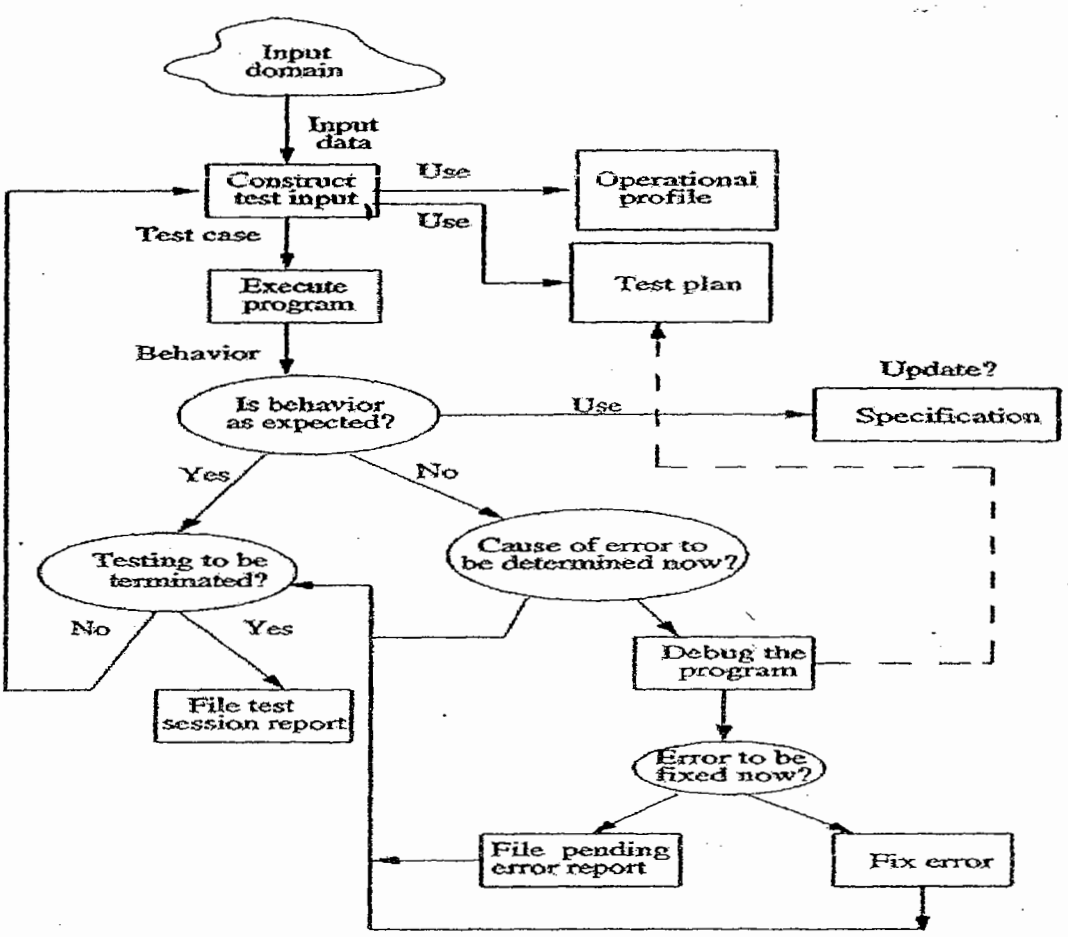


Figure: A test and Debug cycle.

Preparing a test plan

- A test cycle is often guided by a **test plan.**
- **Example:** Test plan for *sort*

The sort program is to be tested to meet the requirements given earlier. Specifically, the following needs to be done: 1. Execute sort on at least two input sequences, 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 inputs 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 (test case)

- 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 more test cases.

Example: The following test cases are generated for the *sort* program using the test plan given above.

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 testing
- Tester constructs test harness to aid in 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.

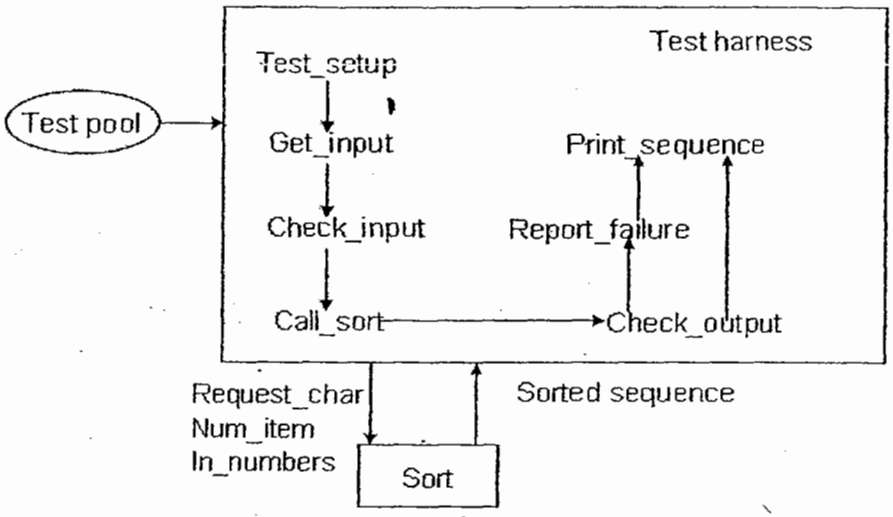


Figure: A simple test harness to test the *sort* program.

Specifying program behavior

- Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.
- Program ***state*** can be used to define program behavior, and ***state transition diagram*** (or ***state diagram***) can be used to specify program behavior
- The **state** of a program is the set of current values of all its variables and an indication of which statement in the program is to be executed next. One way to encode the state is by collecting the current values of program variables into a vector known as the **state vector.**
- A **state diagram** specifies program states and how the program changes its state on an input sequence.
- Indication of where the control of execution is at any instant of time is given by an **identifier** associated with the next program statement. In assembly language program counter is used for this purpose

Example: Program p1.1

The state vector for this program consists of four elements. The first element is the statement identifier where control of execution is currently positioned. The next three elements are, respectively, the values of the three variables X, Y, Z.

The letter u is an element of the state vector stands for an *undefined* value.

The notation s_i --→ s_j is an abbreviation for "The program moves from state s_i to s_j". The movement from s_i to s_j is caused by the execution of the statement whose identifier is listed as the first element of state s_i.

```
1    Integer x, y, z;
2    Input (x, y);
3    If(x<y)
4    {z=y;}
5    Else
6    {z=x;}
7    endif
8    output(z);
9    end
```

The possible sequence of states that the *max* program may go through is given below:

[2 u u u] --→ [3 3 15 u] --→ [4 3 15 15] --→ [5 3 15 15] --→
[8 3 15 15] --→ [9 3 15 15]

- For *max* program final state is sufficient for determining the maximum of two numbers & also for the tester.
- Consider a menu-driven application named *myapp*. Figure shows the menu bar for this application.

When started, the application enters the initial state.

ASHOK KUMAR K
mob: 9742024066
Email: celestialcluster@gmail.com

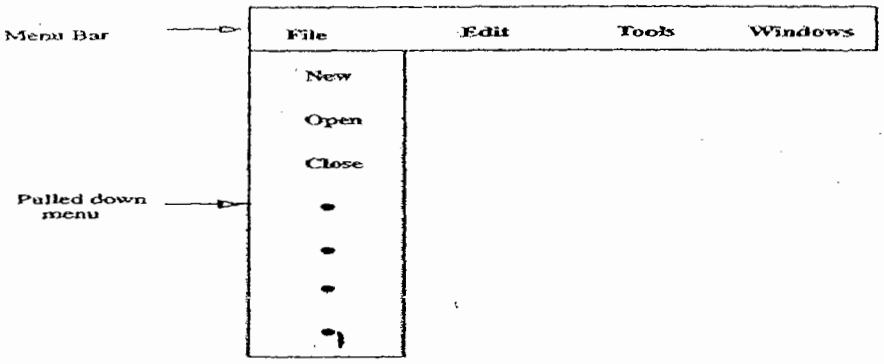


Figure: Menu Bar displaying four menu items when application *myapp* is started

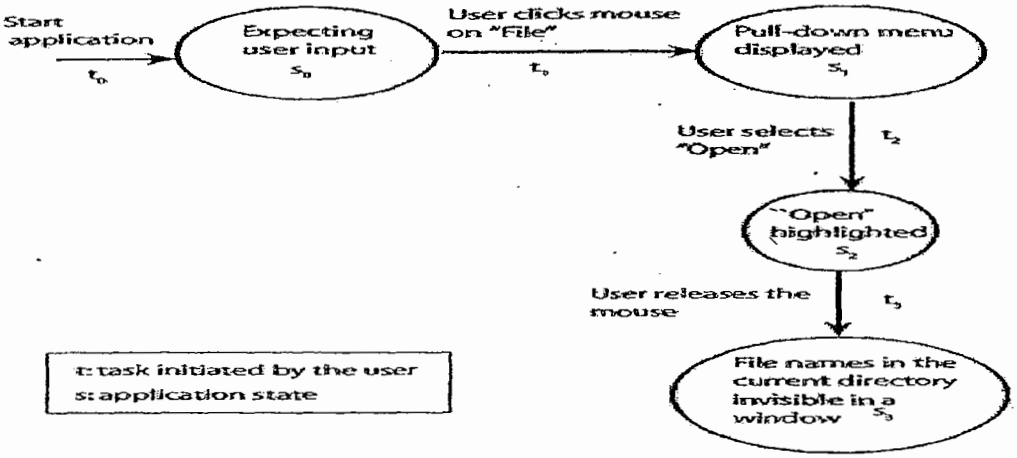


Figure: A state sequence for *myapp* showing how the application is expected to behave when the user selects the `open` option under the `file` menu

Assessing the correctness of program behavior

- An important step in testing a program is the one wherein the tester determines if the observed behavior of the program under test is correct or not
- This step is divided into two smaller steps
    1. In the first step one observes the behavior.
    2. In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.
- The entity that performs the task of checking the correctness of the observed behavior is known as an oracle

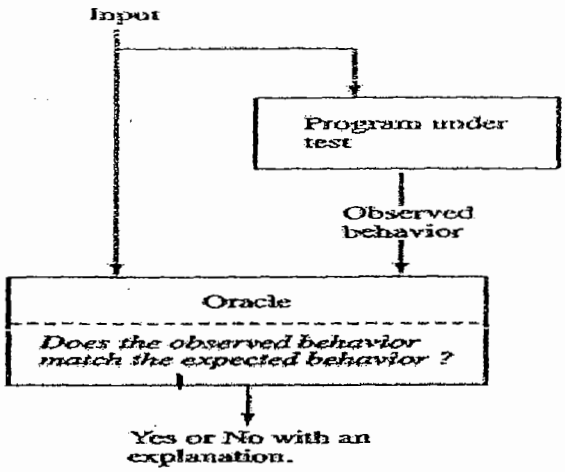


Figure: Relationship between the program under test and the oracle.

- A tester often assumes the role of an oracle and thus serves as a human oracle. Example: To verify if the output of a matrix multiplication program is correct or not, a tester might input two 2X2 matrices and check if the output produced by the program matches the result of hand calculation.
- Even though a human oracle is often the best available oracle, it has several disadvantages:
  - Error prone
  - Slower
  - Result in checking of only trivial I/O behaviors.
- Oracles can also be programs designed to check the behavior of other programs. Example: one might use a matrix multiplication program to check if a matrix inverse program has produced the correct output.
- Advantages of using programs as oracles are:
  - Speed
  - Accuracy
  - Ease with which complex computations can be checked

Construction of oracles

- Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires the determination of input-output relationship.
- In general, the construction of automated oracles is a complex undertaking.
- Example for oracle construction:

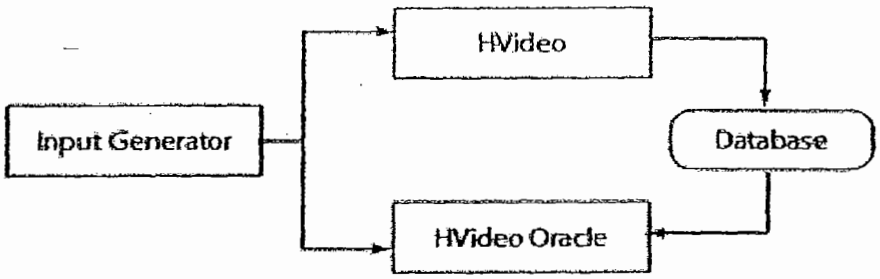


Figure: Relationship between an input generator, `HVideo`, and its oracle

TEST METRICS

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

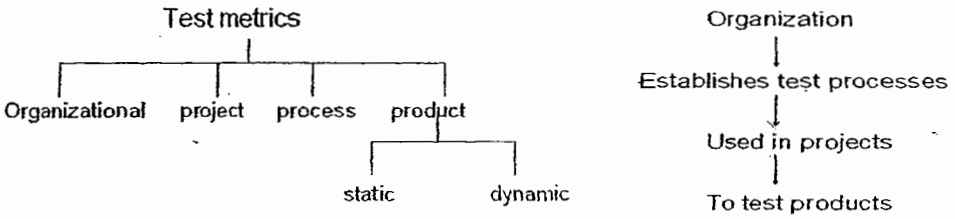


Figure: Types of metrics used in software testing and their relationships

- Regardless of level at which metrics are defined and collected, there exist four general core areas that assist in the design of metrics. These are:
  1. **Schedule**: related metrics measure actual completion times of various activities and compare with estimated time to completion
  2. **Quality**: related metrics measure quality of a product or process
  3. **Resource**: related metrics measure items such as cost in dollars, manpower and tests executed
  4. **Size**: related metrics measure size of various objects such as the source code and number of tests in a test suite.

Organizational metrics

- At organizational level metrics are useful in overall project plan & management
- Average over a set of products developed and marketed by an organization is the product quality

- Computing product quality at regular intervals and overall products released over a given duration shows the quality trend across the organization
- Organizational-level metrics allow senior management in setting new goals and plan for resources needed to realize the goals

Project metrics

- Project metrics relate to a specific project, for ex: the I/O device testing project or a compiler project.
- The ratio of actual-to-planned system test effort (tester-man-months) is one project metric.
- Another project metric is the ratio of number of successful tests to the total number of tests in the system test phase.

Process metrics

- Every project uses some test process.
- The Big-bang approach is one process used in relatively small single-person projects
- Goal of process metric is to assess the goodness of the process
- Later a defect is found the costlier it is to fix. Hence, a metric that classifies defects according to the phase in which they are found assists in evaluating the process itself.

Product metrics: Generic

- Product metrics relate to a specific product such as a compiler for a programming language. These are useful in making decisions related to the product. For ex: "should this product be released for use by the customer?"
- Product complexity-related metrics are of two types
    1. Cyclomatic complexity
    2. Halstead Metrics
- Cyclomatic complexity
    - Proposed by Thomas McCabe in 1976 is based on the control flow of a program.
    - Given control flow graph G of a program P containing N nodes, E edges, and p connected procedures, the cyclomatic complexity V(G) is computed as follows:

V(G)=E-N+2p

- **Halstead metrics**
  - It was published by Prof Maurice Halstead in a book titled Elements of software Science.
  - Using program size (S) and effort (E) , the proposed number of errors (B) found during a software development effort:

    B=7.6E0.667S0.333
  - Advantage of using an estimator such as B is that it allows the management to plan for testing resources.

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

Metric	Meaning
Reliability	Probability of failure of a s/w wrt a given operational profile in a given environment

Defect density	Number of defects per KLOC
Defect severity	Distribution of defects by their level of severity
Test coverage	Fraction of testable items e.g. basic blocks
Cyclomatic complexity	Measures complexity of a program based on its CFG
Weighted methods per class	$\sum_{n=1}^{n} C$
Class coupling	Measures the number of classes to which a given class is coupled
Response set	Set of all methods that can be invoked, directly or indirectly , when a message is sent to object o
Number of children	Number of immediate descendants of a class in the class hierarchy

Table: A sample of product metrics

Static and Dynamic Metrics

- Static metrics are those computed without having to execute the product Ex: Number of testers working on a project.
- Dynamic metrics require code execution Ex: Number of defects remaining to be fixed.

Testability

- Testability is the degree to which a system or components facilitates the establishment of test criteria & the performance of tests to determine whether those criteria have been met.
- Testability of a product can be categorized into static and dynamic testability metrics:
    - Static testability metrics: Ex: Software complexity - more complex an application , lower the testability that is , higher the effort required to test it

- Dynamic metrics: Ex: code-based coverage criteria- program which is difficult to generate tests

- Testability is concern in both hardware and software:
  - In hardware testability detects fault with respect to a fault model in finished product.
  - In software it focuses on verification of design & implementation.

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.
  It is obvious that a **software application does not degrade over time** (any fault present in the application will remain and no new faults will creep in unless the applications changed), whereas **hardware degrades over time** (ex: VLSI chip may fail over time).
- This difference leads to **BIST (Built in self test)** techniques applied to hardware and software designs.
- **Fault models**: Hardware testers generate tests based on fault models. For ex: using a *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 ex: to test whether there is a memory leak in an application, one performs a combination of stress testing and code inspection.
- Hardware testers use a variety of fault models at different levels of abstraction. For ex: at the lower level there are transistor level faults. At higher levels there are gate level, circuit level, and function-level fault models. Software testers might or might not use fault models during test generation even though the models exist.
- **Test domain**: A major difference between tests for hardware and software is in the domain of tests.
  - For hardware domain of the test input involves bit patterns.
  - For software domain of the test input involves tuples consisting of one or more basic data types.
- **Test coverage**:
  - Tough in hardware
  - Easy in software

ASHOK KUMAR K
mob: 9742024066
Email: celestialcluster@gmail.com

Software Testing	Hardware Testing
software application does not degrade over time	hardware degrades over time
Hardware testers generate tests based on fault models	Software testers generate tests to test for correct functionality. Sometimes such tests do not correspond to any general fault model.
Hardware testers use a variety of fault models at different levels of abstraction	Software testers might or might not use fault models during test generation even though the models exist.
For hardware, domain of the test input involves bit patterns	For software, domain of the test input involves tuples consisting of one or more basic data types
Test coverage is tough in hardware	Test coverage is easy in software

TESTING AND VERIFICATION

- Program verification aims at proving the correctness of programs by showing that it contains no errors. i.e, it aims at showing that a given program works for all possible inputs that satisfy a set of conditions.
  Testing aims at uncovering errors in a program. i.e, it aims to show that the given program is reliable in that no errors of any significance were found
- Program verification and testing are best considered as complementary techniques. In practice, one can shed program verification, but not testing.
- 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 perfect technique as it promises to verify that a program is free from errors. However, the person who verified a program might have made mistake 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, and so on.

- Thus, **Neither verification nor testing is a perfect technique for proving the correctness of programs.**

DEFECT MANAGEMENT

- Defect management is an integral part of a development and test process in many software development organizations. It is a subprocess of development process.
- Defect management entails the following: defect prevention, discovery, recording and reporting, classification, resolution, and prediction.
  - **Defect prevention** is achieved through a variety of processes and tools Ex: Good coding techniques, unit test plans, code inspections.
  - **Defect discovery** is the identification of defects in response to failures observed during dynamic testing or found during static testing. Discovering a defect often involves debugging the code under test.
  - Defects found are **classified** and **recorded** in a database. Classification becomes important in dealing with the defects. For ex: defects classified as *high severity* are likely to be attended to first by the developers than those classified as *low severity*.
  - Each defect, when recorded, is marked as *open* indicating that it needs to be resolved. One or more developers are assigned to resolve the defect. Resolution requires careful scrutiny of the defect, identifying a fix if needed, implementing the fix, testing the fix, and finally closing the defect indicating that it has been resolved.
  - Organizations often do source code analysis to **predict** how many defects an application might contain before it enters the testing phase
  - Several tools exist for recording defects, and computing and reporting defect related statistics. Ex: **BugZilla** (open source), and **FogBugz** (commercially available).

EXECUTION HISTORY

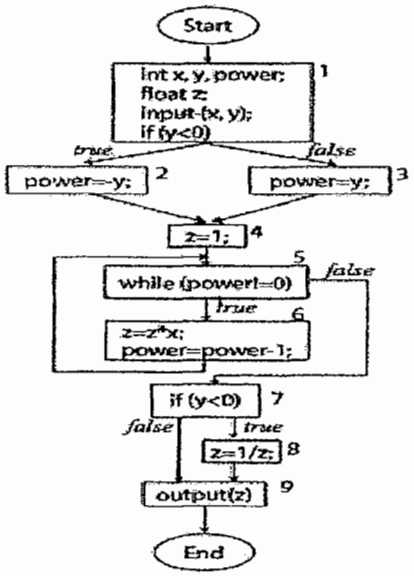
- **Execution history (execution trace)** of a program is an organized collection of information about various elements of a program during a given execution.
- An **Execution slice** is an executable subsequence of execution history.
- There are several ways to represent an execution history:
  - Sequence in which functions in a program are executed against a given test input.
  - Sequence in which program blocks are executed.
  - For a program written in object-oriented language such as java, execution history is represented as a sequence of objects and the corresponding methods accessed.
- Example: Consider the 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
```

A list of all basic blocks in program P1.2 is given below:

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

Control Flow Graph (CFG) for P1.2 is given below:



We are interested in finding the sequence in which the basic blocks are executed when the program P1.2 is executed with the test input `t1: < x=2, y=3 >`. A straight forward examination of its CFG reveals the following sequence: `1, 3, 4, 5, 6, 5, 6, 5, 6, 7, 9`. This sequence of blocks represents an execution history of program P1.2 against test `t1`.

Another test t2: < x=1, y=0 > generates the following execution history expressed in terms of blocks: 1, 3, 4, 5, 9.

- The More the information in the execution history, the larger the space required to save it.
- For debugging a function, one might want to know the sequence of blocks executed as well as values of one or more variables used in the function.
- For selecting a subset of tests to run during regression testing, and for performance analysis, one might be satisfied with only a sequence of function calls or blocks executed.
- A complete execution history recorded from the start of a program's execution until its termination represents a single execution path through the program. Partial execution history of program can also be recorded.

TEST-GENERATION STRATEGIES

- One of the key tasks in any software test activity is the generation of test cases. The program under test is executed against the test cases to determine whether it conforms to the requirements.
- Any form of 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.
- In most commercial environments, the process is a bit more formal. The tests are generated using a mix of formal and informal methods often directly from the requirements document serving as the source. In more advanced test processes, requirements serve as a source for the development of formal models.
- Test strategies:
  - Model-based test generation: require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri net, etc.

- **Specification based**: require that a subset of the requirements be modeled using a formal mathematical notation. Examples: B, Z, and Larch.
- **Code-based test generation**: generate tests directly from the code.

- Figure summarizes several strategies for test generation:

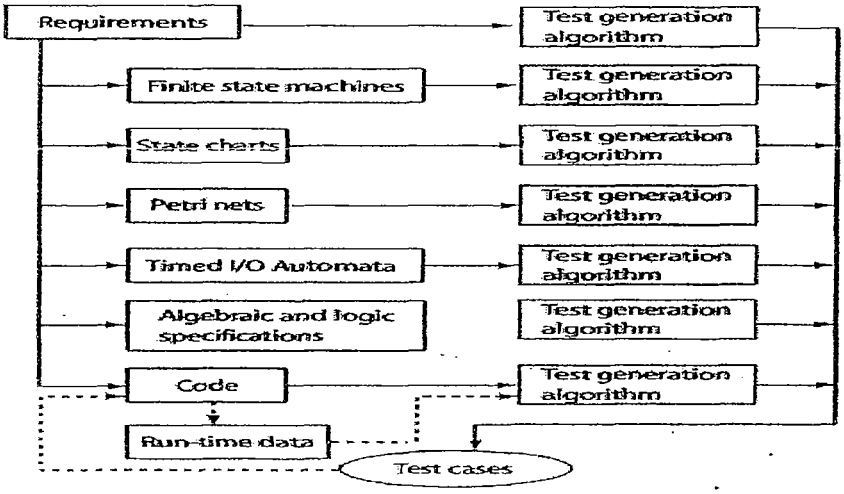


Figure: Requirements, models, and test generation algorithms.

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 applications under test.
- Static testing is useful in that it may lead to **discovery of faults** in the application, as well as **ambiguities** and **errors** in requirements and other application-related documents, at a relatively **low cost.**
- Static testing is best **carried out by** an individual who did not write the code, or by a team of individuals.
- **Test team has access** to requirements document, application, and all associated documents such as design document and user manuals. They also have access to one or more **static testing tools**, which takes the application code as input and generates a variety of data useful in the test process.
- A sample process of static testing is illustrated in below fig:

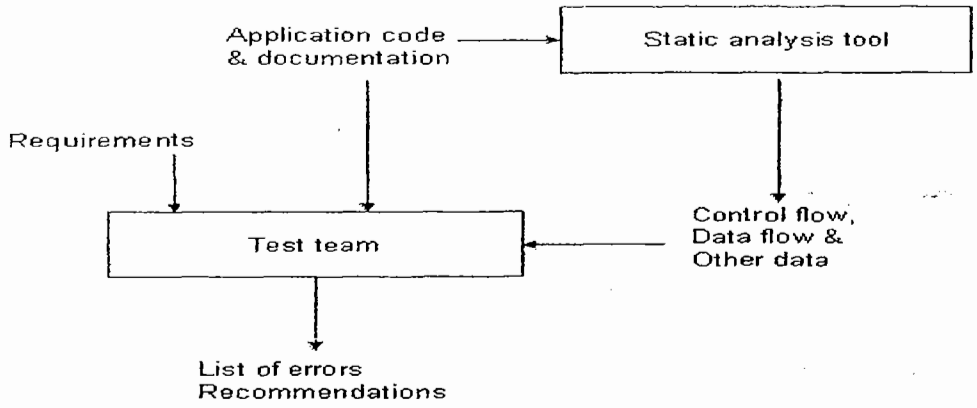


Figure: Elements of Static testing

- **Walkthroughs** and **Inspections** are an integral part of static testing.

Walkthroughs

- It is an **informal process** to review any application-related document.
  For ex:
  - Requirements are reviewed using a process termed **requirements walkthrough**
  - Code is reviewed using code **walkthrough (peer code review)**
- It begins with a **review plan** agreed upon by all members of the team. Each item of the document (ex: source code module) is reviewed with clearly stated objectives in view. A **detailed report** is generated that lists item of concern regarding the document reviewed.
- In **requirements walkthrough**, the test team must review the **Requirement document** to ensure that the requirements match the user needs, and are free from ambiguities and inconsistencies. Both functional and non-functional requirements are reviewed.

Inspections

- It is a more **formally defined process** usually **associated with code.**
- Code inspection **increases productivity and software quality.**
- Code inspection is **carried out by a team** and works according to the **inspection plan consisting of**
  1. Statement of purpose
  2. Work product to be inspected

3. Team information, roles, and tasks to be performed
4. Rate at which inspection task is to be completed
5. 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 are assigned the **roles** of moderator, reader, recorder, and author.
  - **Moderator** is in charge of the process and leads the review.
  - Actual code is read in by the **reader**, perhaps with the help of code browser and with large monitors for all in the team to view the code.
  - The **recorder** records any errors discovered or issues to be looked into.
  - The **author** is the actual developer whose primary task is to help others understand code.

Use of static code analysis tools in static testing

- **Static code analysis tool** can provide control-flow and data-flow information.
- Control flow information, presented in the form of CFG, helps the inspection team to determine **flow of control under different conditions.**
- CFG can be annotated with data-flow information to make a data flow graph (for ex: append each node of a CFG with a list of variables). This information is valuable to the inspection team in **understanding the code** as well as **pointing out possible defects.**
- **Commercially available static analysis tools for C and java programs:** Purify from IBM rational, Klockwork from Klockwork, Inc.

  Open source tool for the analysis of java programs: Lightweight analysis for program security in Eclipse (LAPSE)

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.
- Figure below illustrates the process of model-checking.

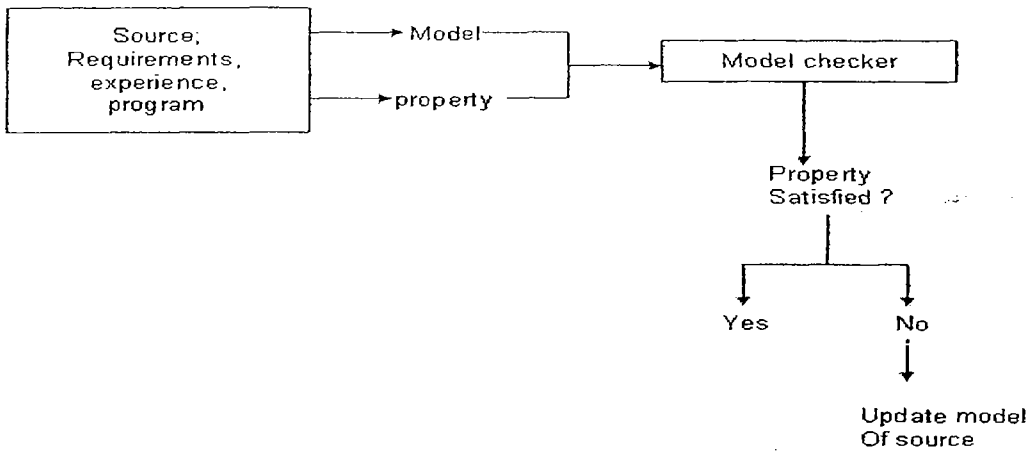


Figure: Elements of Model checking

- A **model**, usually finite-state, is extracted from some source. The source could be requirements, and in some cases, the application code itself. Each state of the finite-state model is prefixed with one or more **properties** that must hold when the application is in that state. For es: a property could be as simple as `x < 0`, indicating that variable x must hold a negative value in this state.
- One or more desired properties are then **coded** in a formal specification language.
- For each property, the checker could come up with one of the **three possible answers:**
  - Property is satisfied
  - Property is not satisfied
  - Unable to determine

  In the second case, the model checker provides a counterexample 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.

CONTROL-FLOW GRAPH (CFG) or Flow Graph or Program Graph

- A CFG captures the flow of control within a program. It assists testers in the analysis of a program to understand its behavior in terms of the flow of control.

Basic Block

- A basic block, or simply a **block**, in program 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.

  These points are the first and the last statements within a basic block. Control always enters a basic block at its entry point and exits from its exit points.

- Example: This program takes two integers `x` and `y`, and outputs 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 line 5 containing an `if` statement. Considering that there is decision at line 5, control could go to one of two possible destinations at lines 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
```

A list of all basic blocks in program P1.2 is given below

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

CFG: Definition and pictorial representation

- **Definition:** A **control flow graph (or flow graph) G** is defined as a finite set **N** of nodes and a finite set **E** of directed edges. An edge (i, j) in E, with an arrow directed from i to j, connects nodes n_i and n_j in N. We often write G= (N, E) to denote a flow graph G with nodes in N and edges in E.
- `Start` and `End` are two special nodes in N and are known as distinguished nodes. Every other node in G is reachable from `Start`. Also, every node in N has a path terminating at `End`. The node `Start` has no incoming edge, and `End` has no outgoing edge.
- **Pictorial Representation:**
  - In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.
  - Blocks and nodes are labeled such that block b_i corresponds to node n_i. An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j.
  - Each node is represented by an oval or a rectangular box. These boxes are labeled by their corresponding block numbers. The boxes are connected by lines representing edges. Arrows are used to indicate the direction of flow. A block that ends in a decision has two edges going out of it. These edges are labeled `true` and `false`.
- **Example:**

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) }

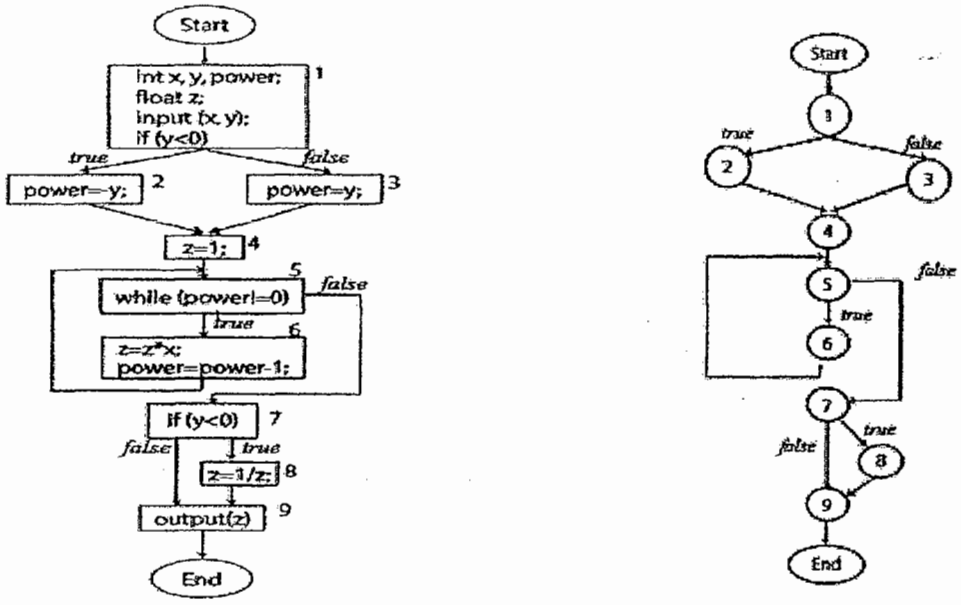


Figure: CFG for program P1.2

(a): Statements in each block are shown

(b): Statement within a block are omitted

Path

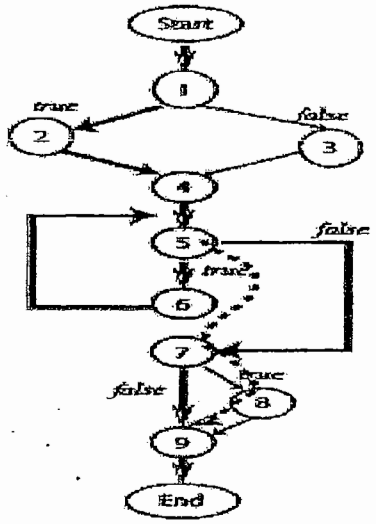
- Consider a flow graph G=(N, E). A sequence of k edges, k>0, (e_1, e_2, ... e_k), denotes a **path** of length k through the flow graph if the following sequence condition holds:

  Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and 0<i<k, if $e_i=(n_p, n_q)$ and $e_i+1=(n_r, n_s)$ then $n_q=n_r$

- Note:
  - **Complete path:** A path is said to be complete if the first node along the path is 'Start' and the terminating node is 'End'

- **Feasible path:** A path p through a program is said to be feasible, if there exists at least one test case which when input to P causes p to be traversed.
- If no test cases exists, then p is considered infeasible

- Example:



Feasible and complete paths:

p1 = (Start, 1, 2, 4, 5, 6, 5, 7, 8, 9, End)

p2 = (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)

Incomplete paths:

p3 = (5, 7, 8, 9)

p4 = (6, 5, 7, 9, End)

Complete but infeasible paths:

p5 = (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)

p6 = (Start, 1, 2, 4, 5, 7,9, End)

Invalid paths: because they do not satisfy the sequence condition stated earlier:

P7 = (Start, 1, 2, 4, 8, 9, End)

P8 = (Start, 1, 2, 4, 7, 9, End)

TYPES OF TESTING

- Here we present a framework consisting of a set of five classifiers that serve to classify testing techniques that fall under the dynamic testing category. Each

of the five classifiers is a mapping from a set of features to a set of testing techniques. Here are the five classifiers:

1. C1: Source of test generation.
2. C2: Lifecycle 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 models

Classifier C1: Source of test generation

Artifact	Technique	Example
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
Code	White-box	Adequacy assessment Coverage testing Data-flow testing Domain testing Mutation testing Path testing Structural testing Test minimization using coverage
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 interface	Interface testing	Interface mutation Pairwise testing

- **Black-box testing:** Tests can be generated from formally or informally specified requirements and without the aid of the code under test. Such form of testing is referred to as **black-box testing.**
- **Model-based or specification-based testing:** When the requirements are formally specified, then **model-based or specification based testing** is done. This is also a form of black-box testing.
- **White-box Testing: White-box testing** refers to the test activity wherein code is used in the generation of or the assessment of 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 this path. In the indirect case, tests generated using some black-box techniques are assessed against some code-based coverage criterion.

- **Interface testing:** Here, the tests are often generated using a component's interface. Certainly, the interface itself forms a part of the component's requirements and hence this form of testing is black-box testing.

Classifier C2: Life cycle phase

Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Post system, pre-release	Beta-testing

Testing activities take place throughout the software life cycle. Here, Testing is often categorized based on the phase in which the activity occurs.

- **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 **unit testing.**
- **Integration testing:** When units are integrated and a large component or a subsystem formed, programmers do **integration testing** of the subsystem.
- **System Testing:** When the entire system has been built, its testing is referred to as **system testing.**
- **Beta-testing:** Often a carefully selected set of customers is asked to test a system before commercialization. This form of testing is referred to as beta testing
- **Regression testing:** Errors reported by users of an application often lead to additional testing and debugging. In such situation, one performs a **regression** test. The goal of regression testing is to ensure that the modified system functions as per its specification.

Classifier C3: Goal-directed testing

Goal	Technique	Example
Advertised features	Functional testing	
Security	Security testing	
Invalid inputs	Robustness testing	
Vulnerabilities	Vulnerability testing	
Errors in GUI	GUI testing	Capture/plaback Event sequence graphs Complete Interaction Sequence
Operational correctness	Operational testing	Transactional-flow
Reliability assessment	Reliability testing	
Resistance to penetration	Penetration testing	
System performance	Performance testing	Stress testing
Customer acceptability	Acceptance testing	
Business compatibility	Compatibility testing	Interface testing Installation testing
Peripherals compatibility	Configuration testing	

Finding any hidden errors is the prime goal of testing, goal-oriented testing looks for specific types of failures.

- **Vulnerability testing:** It detects if there is any way by which the system under test can be penetrated by unauthorized users
- **Penetration testing:** It aims at evaluating how good the policies & measures are.
- **Robustness testing:** It is the task of testing an application for robustness against unintended inputs (invalid inputs)
- **Stress testing:** checks the behavior of an application under stress. i.e. It checks an application for conformance to its functional requirements as well as to its performance requirements when under stress.
- **Performance testing:** Here application is tested specifically with performance requirements in view.
- **Load testing:** It is a phase of testing in which an application is loaded with respect to one or more operations. Load testing is also stress testing, performance testing, robustness testing.

Classifier C4: Artifact under test

ASHOK KUMAR K
mob: 9742024066
Email: celestialcluster@gmail.com

Characteristics	Technique
Application component	Component testing
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	Operating system testing
Real-time software	Real-time testing
Requirements	Requirement testing
Software	Software testing
Web service	Web service testing

Testers often say "we do X-testing" where X corresponds to an artifact under test.

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 test process listed in below table.

Process	attributes
Testing in waterfall model	Usually done towards the end of the development cycle
Testing in V-model	Explicitly specifies activities in each phase of the development cycle
Spiral testing	Applied to software increments
Agile testing	Used in agile development methodologies such as eXtreme programming
Test-driven development(TDD)	Requirements specified as tests

THE SATURATION EFFECT

- The saturation effect is an abstraction of a phenomenon observed during the testing of complex software systems.

To understand this effect refer to the below figure.

Fig Plz refer text book

The horizontal axis in the figure refers to the test effort that increases 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.

UNIT 3: TEST GENERATION FROM REQUIREMENTS - 1

Syllabus

* Introduction
* The test selection problem
* Equivalence partitioning
* Boundary value Analysis
* Category partition method

– 7 Hours.

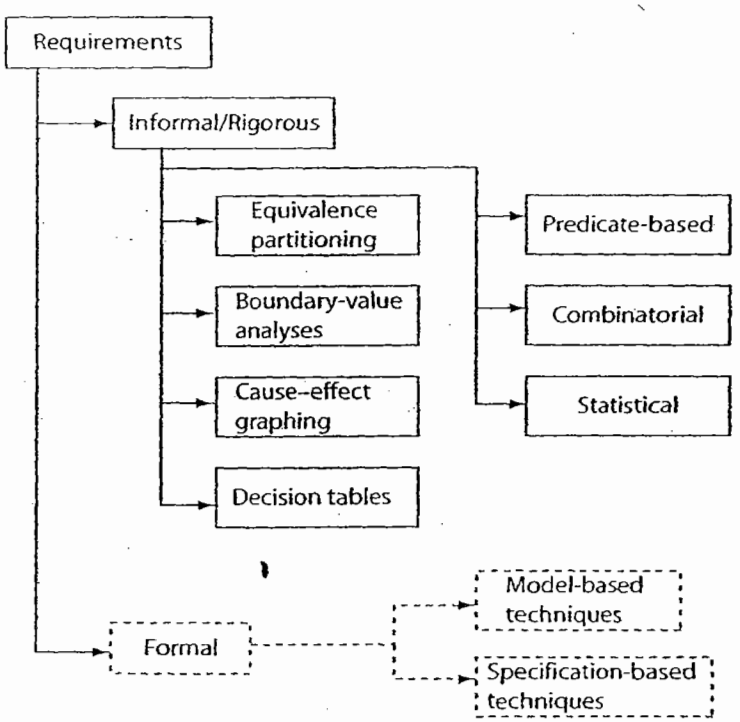
INTRODUCTION

* Requirements serve as a starting point for the generation of tests.

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

* Figure shows variety of test generation techniques:



THE TEST - SELECTION PROBLEM

* let D denote the input domain of program P.
  The test selection problem is to select a subset T 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 $T \subseteq D$ 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.

* Exaustive testing:

→ By exhaustive testing we mean testing the given program against every element in its input domain.

→ The complexity makes it harder to select individual tests.

→ The following two examples illustrate what is responsible for large and complex input domains.

ex 1: large input domain

- Consider a program p that is required to sort a sequence of integers into ascending order. Assuming that p will be executed on a machine in which integers range from -32768 to 32767, the input domain of p consists of all possible sequences of integers in the range [-3268, 32767]
- If the size of the input sequence is limited to say N, >1. Then the size of input domain depends on the value of N.

Size of input domain $S = \sum_{i=0}^{N} v^i$

V → no. of possible values each element of the input sequence may assume, ie 65536.

ex 2: Complex input domain

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

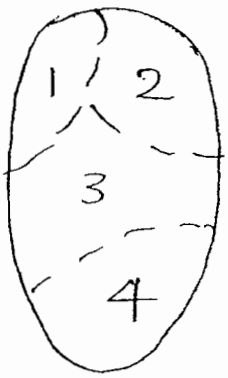
```
ID : int;
name : string;
rate : float;
hoursWorked : int;
```

} complex.

→ used in most of the situations

EQUIVALENCE PARTITIONING

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



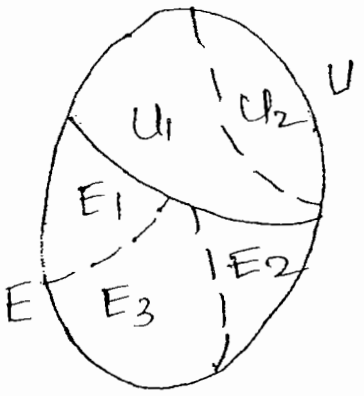
(a) Equivalence classes constitute a partition as they are disjoint

(b) Equivalence classes do not constitute a partition as they are not disjoint

+ one test is selected from each equivalence class.
+ when the equivalence classes created by two testers are identical, tests generated are different.

Faults Targeted.

+ the entire set of inputs can be divided into atleast two subsets
  - → one containing all expected (E) or legal inputs.
  - → other containing all unexpected (U) or illegal inputs



* Example:

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

→ Set of input values is now divided into E and U.

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

→ Further more, E is subdivided into

[1, 2 ... 61] and [62, ... 120]

↳ according to requirement R_1 ↳ According to requirement R_2

→ 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.

ex: → a method addList that returns the sum of elemem 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.

ex: $((1,5),6)$, and $((-3,14,3),14)$

→ the relation computed by addList is defined as follows:

$$addList : \mathcal{L} \rightarrow Z$$

$\mathcal{L}$ → set of all lists of integers

Z → set of integers.

Suppose that addList has an error (empty list) then,

$$addlist : \mathcal{L} \rightarrow Z \cup \{error\}$$

→ Relations that help a tester partition the input domain of a program are usually of the kind:

$$R : I \rightarrow I$$

I → input domain.

* Below ex shows a few ways to define equivalence classes based on the knowledge of requirements & the program text.

Example: the wordcount method takes a word w and a filename f as input and returns the no. of occurrences of w in the text contained in the file named 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 eq. 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
..	..	..

→ So we note that the no. of eq. classes without any knowledge of prog code is 2, whereas that with the knowledge of partial code is 6.

→ Eq classes based on program output

quest 1: Does the program ever generate a 0?
quest 2: what are the max & min possible values of the output?

These two questions leads to following eq. classes:

E1: Output value v is 0
E2: Output value v is the max. possible
E3: — " — min — " —
E4: All other output values.

Equivalence classes for variables

Table 2.1 Guidelines for generating equivalence classes for variables: range and strings

Kind	Equivalence classes	Example: Constraint	Example: 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}↓

Table 2.2 Guidelines for generating equivalence classes for variables: Enumeration and arrays

Kind	Equivalence classes	Example[a]: Constraint	Example[a]: 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]}↓}

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.

* Multi Dimensional partitioning

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

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

$3 \le x \le 7$

$5 \le y \le 9$.

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

$E_1: x < 3$ $E_4: y < 5$

→ For multidimensional partitioning, we consider the input domain to be the set product $X \times Y$. This leads to 9 eq. classes.

E1: $x < 3, y < 5$

E2: $x < 3, 5 \leq y \leq 9$

E3: $x < 3, y > 9$

E4: $3 \leq x \leq 7, y < 5$

E5: $3 \leq x \leq 7, 5 \leq y \leq 9$

E6: $3 \leq x \leq 7, y > 9$.

E7: $x > 7, y < 5$

E8: $x > 7, 5 \leq y \leq 9$

E9: $x > 7, y > 9$.

Systematic procedure for equivalence partitioning.

* Given the program requirements, the following steps are helpful in creating the equivalence classes.

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. Examine the set of values each variable can assume, an

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 behavior 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 not combining the equivalence classes, one misses the opportunity to generate useful tests.

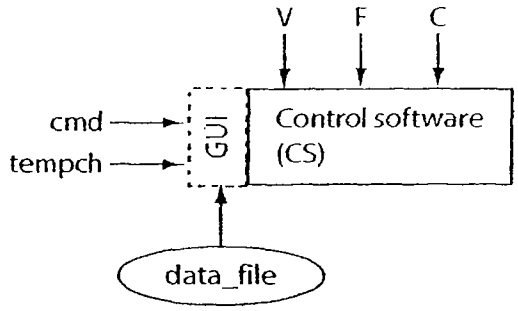
Eq. classes are combined using multidimensional approach.

4. Identify Infeasible Equivalence classes:

An infeasible Equivalence class is one that contains a combination of input data that can not 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 ~~four~~ three commands (cmd)
  - → change the boiler temperature (temp)
  - → shut down the boiler (shut)
  - → cancel the request (canc)
* 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. (0F is not allowed)
* Selection of option C forces the BCS to examine variable V. If V is set to 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.
* 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 F.



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.

1. Identify the input domain

* First we examine the requirements, identify input variables, their types, & 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}

∴ I/p domain $\subseteq S = V \times F \times cmd \times tempch$

→ eg: (GUI, –, temp, -5)
↳ dont care

2. Equivalence classing.

variable	partion
V	{{GUI}, {file}, {undefined}}
F	{valid}, {invalid}
Cmd	{{temp}, {cancel}, {shut}, {<_invalid>}}

3. Combine equivalence class

* note that tinvalid, tvalid, finvalid & fvalid denote sets of values. "undefined" denotes one value

4. 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.

{ CV, F, {cancel, shut, cinvalid}, tvalid ∪ tinvalid) }

Test selection based on Equivalence classes.

Table 2.3 Test data for the control software of a boiler control system ~~in Example 2.8~~.

ID	Equivalence class[a] {CV, F, cmd, tempch)}	Test data[b] (CV, 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)

GUI design and Equivalence classes

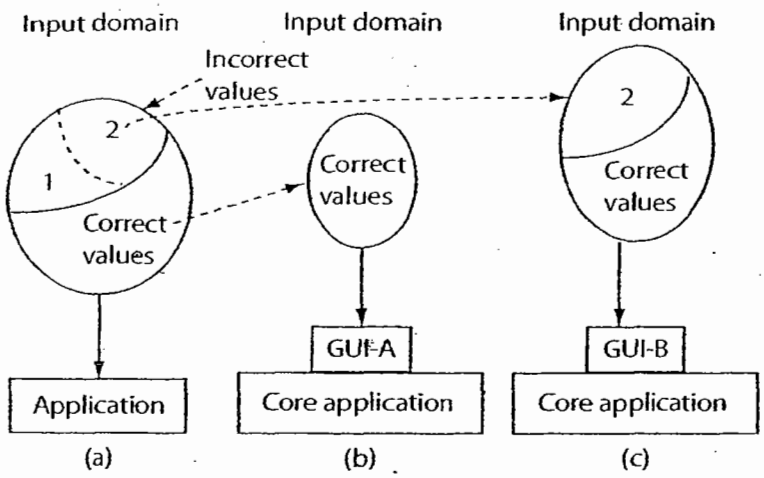


Fig. 2.6 Restriction of the input domain through careful design of the GUI. Partitioning of the input domain into equivalence classes must account for the presence of GUI as shown in (b) and (c). GUI-A protects all variables against incorrect input while GUI-B does allow the possibility of incorrect input for some variables.

BOUNDARY VALUE ANALYSIS (BVA)

* 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'
  - 'code' → integer
  - 'qty' → integer.

1. Create equivalence classes

→ Assuming that an item code must be in the range 99 to 9999 and qty in the range 1 to 100,

Equivalence classes for 'code':

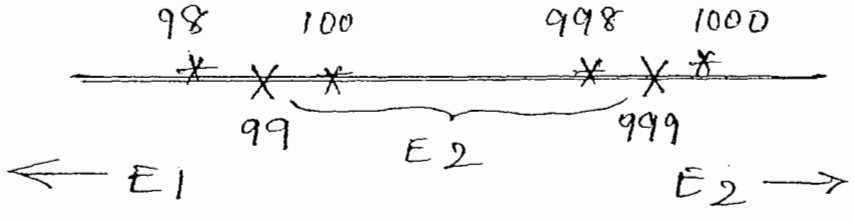
E1: Values less than 99
E2: values in the range
E3: Vales greater than 999

Equivalence classes for 'qty'

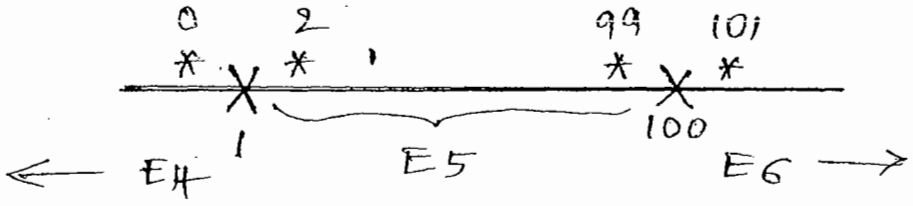
E4: Values less than 1
E5: values in the range
E6: Values greater than 999

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



(a)



(b)

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) $\}$

illegal values of 'code' and 'qty' are included

→ consider the following faulty code skeleton for method "fp"

```
1  public void fp(int code, qty)
2  {
3      if((code<99) && (code>999))
4      {display_error("invalid code"); return;}
5      // validity check for 'qty' is missing!
6      // Begin processing code and qty
7      :
8  }
```

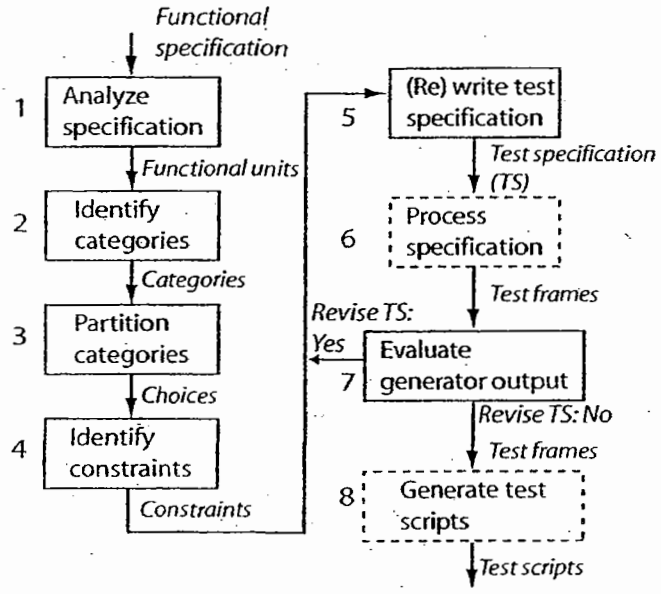
→ t_1 and t_6 tests indicate that value of 'code' is incorrect. But these two tests fails to check that the validity check on 'qty' is missing from the program.

→ none of the other tests will be able to reveal the missing-code error. By separating the correct and incorrect values of different input variables 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 a 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.



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: Analyze 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 analyzed and the inputs ~~are~~ isolated.
* Next we determine characteristics (or a category) of each parameter and environmental object.

Step 3: partition categories

* For each ~~categ~~ category, the tester determines 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 (Assign properties & selector expression to various choices)

* A constraint is specified using a property list and a selector expression

→ property list has the following form –

[property p1, p2 ...]

↳ keyword ↳ names of individual properties.

→ A selector expression takes one of the following forms –

[if p]

[: p and p2 and ...]

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 analyzed 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 test 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

UNIT 5 : STRUCTURAL TESTING

Syllabus

* Overview
* Statement testing
* Branch testing
* Condition testing
* Path testing
* procedure call testing
* comparing structural testing criteria
* the infeasibility problem

— 6 Hours.

ASHOK KUMAR K
VIVEKANANDA INSTITUTE OF TECHNOLOGY
Mob: 9742024066
e-mail: celestialcluster@gmail.com

OVERVIEW

* Definition:

→ Judging the test suite thoroughness based on the structure of the program / software itself is called structural testing / control flow testing ie the structure of the software itself is a valuable source of information for selecting test cases and determining whether a set of test cases has been sufficiently thorough.

→ structural Testing is also known as white box / Glass box / code-based Testing.

* Why structural (control flow) testing?

→ Structural testing is one way of answering the question "what is missing in our test suite?"

- If part of the program is not executed by any test case in the suite, faults in that part can not be exposed.
- A "part" may be
  - → A control flow element or combination
  - → statements (nodes) or Branches (edges)
  - → Fragments & combinations: Conditions, Paths.

→ Structural testing compliments Functional testing by including cases that may not be identified from specifications alone.

note:
+ Executing all control flow elements does not guarantee finding all faults because execution of a faulty statement may not always result (u

* Structural Testing complements Functional Testing : Justification

→ control flow testing complements functional testing by including cases that may not be identified from specifications alone.
A typical case is implementation of a single item of the specification by multiple parts of the program
ex:- HashTable collision (invisible in interface spec^n)

→ On the other hand, test suites satisfying control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria.
ex:- Missing path faults.

* Structural testing in practice :

→ Control flow testing criteria are used to evaluate the thoroughness of test-suites derived from functional testing criteria by identifying elements of the programs not adequately exercised.

→ unexecuted elements may be due to natural differences between specification and implementation or they may reveal flaws of the software or its development process:
  → inadequacy of the specifications that do not include cases present in the implementation
  → coding practice that radically diverges from the specification.
  → inadequate functional test suites.

→ control flow adequacy can be easily measured

```
 1  #include "hex_values.h"
 2  /**
 3   * @title cgi_decode
 4   * @desc
 5   *    Translate a string from the CGI encoding to plain ascii text
 6   *    '+' becomes space, %xx becomes byte with hex value xx,
 7   *    other alphanumeric characters map to themselves
 8   *
 9   *    returns 0 for success, positive for erroneous input
10   *          1 = bad hexadecimal digit
11   */
12  int cgi_decode(char *encoded, char *decoded) {
13    char *eptr = encoded;
14    char *dptr = decoded;
15    int ok=0;
16    while (*eptr) {
17      char c;
18      c = *eptr;
19      /* Case 1: '+' maps to blank */
20      if (c == '+') {
21        *dptr = ' ';
22      } else if (c == '%') {
23        /* Case 2: '%xx' is hex for character xx */
24        int   digit_high = Hex_Values[*(++eptr)];
25        int   digit_low  = Hex_Values[*(++eptr)];
26        /* Hex_Values maps illegal digits to -1 */
27        if (    digit_high == -1 || digit_low == -1 ) {
28          /* *dptr='?'; */
29          ok=1; /* Bad return code */
30        } else {
31          *dptr = 16* digit_high + digit_low;
32        }
33        /* Case 3: All other characters map to themselves */
34      } else {
35        *dptr = *eptr;
36      }
37      ++dptr;
38      ++eptr;
39    }
40    *dptr = '\0';              /* Null terminator for string */
41    return ok;
42  }
```

Figure 12.1: 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).*

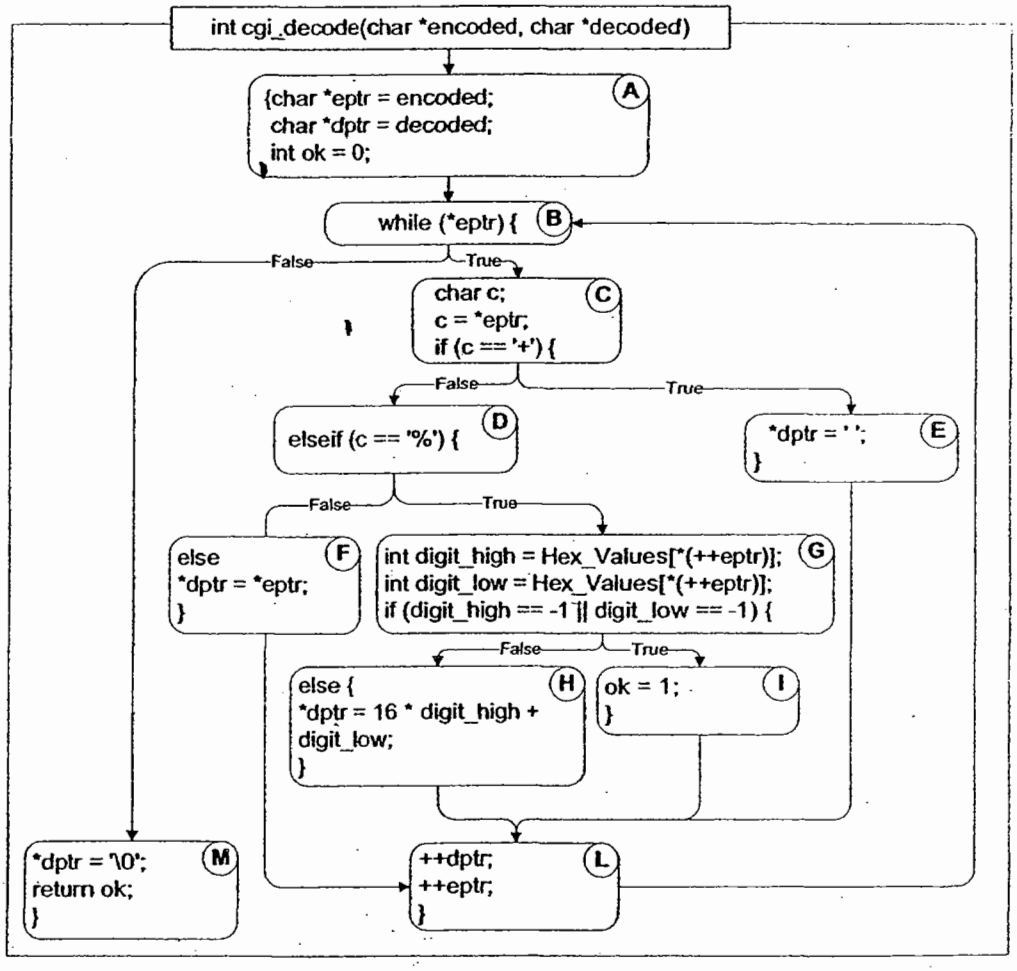


Figure 12.2: Control flow graph of function cgi_decode from Figure 12.1

T_0	=	{ " ", "test", "test+case%1Dadequacy" }
T_1	=	{ "adequate+test%0Dexecution%7U" }
T_2	=	{ "%3D", "%A", "a+b", "test" }
T_3	=	{ " ", "+%0D+%4J" }
T_4	=	{ "first+test%9Ktest%K9" }

Table 12.1: Sample test suites for C function cgi_decode *from Figure 12.1*

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 atleast 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 P 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.

$$C_{statement} = \frac{\text{number of executed statements}}{\text{number of statements}}$$

T satisfies the statement 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 refer to basic block coverage or node coverage.

* Examples: In program 1, it contains 18 statem

→ A test suite $T_0 = \{$ "", "test", "test+case%0Dadequacy"

$C_{statement} = \frac{17}{18} = 94\%$ or node coverage $= \frac{10}{11} = 91\%$

so it does not satisfy the statement adequacy criter

→ A test suite $T_1 = \{$ "adequate+test%0Dexecution%0AU" $\}$

$C_{statement} = \frac{18}{18} = 1$ or 100%

so it satisfies the statement adequacy criterion.

→ A test suite $T_2 = \{$ "%3D", "%A", "a+b", "test" $\}$

$C_{statement} = \frac{18}{18} = 1$ or 100%

note 1: coverage does not depend on number of Test cases.

eg 1: $T_1 >$ coverage T_0

eg 2: $T_2 =$ coverage T_1.

note 2: Minimizing test suite size is seldom the goal because

→ small test cases make failure diagnosis easier

→ a ~~fail~~ failing test case in T_2 gives more information for fault localization than a failing test case in T_1.

~~BRANCH TESTING~~ → problem with statement testing

note 3: complete statement coverage may not imply executing all ~~programs in a~~ branches in a program.

eg $T_3 =$ " ", "+%00+%4J" $\}$ → Block F will be missing

$C_{statement} = 100\%$

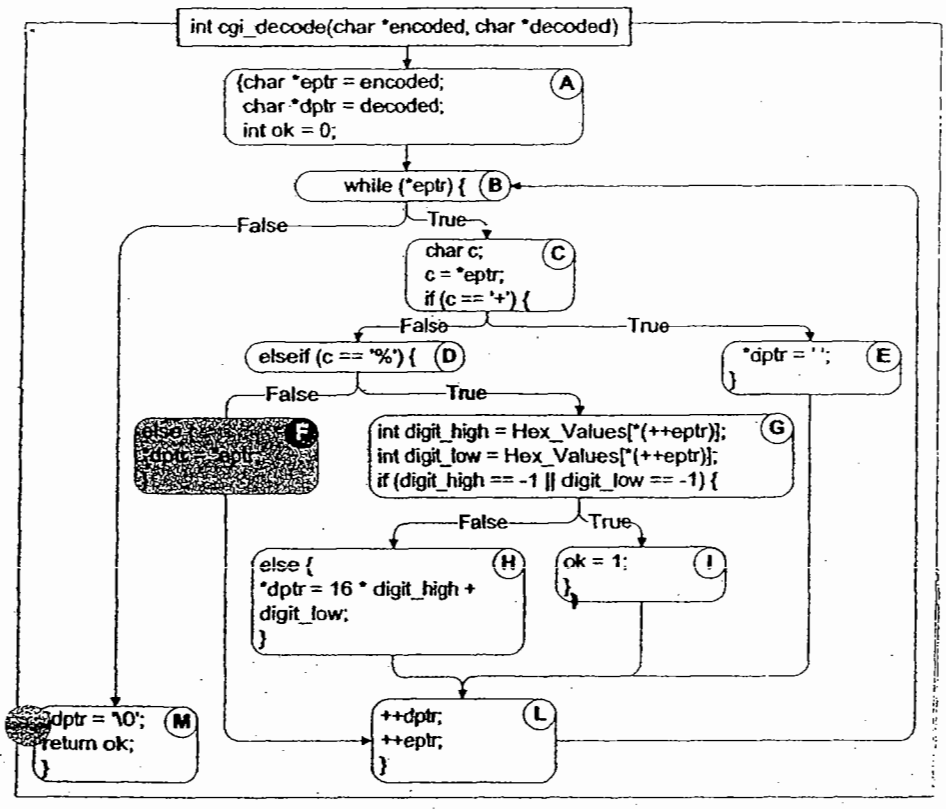


Figure 12.3: The control flow graph of C function cgi_decode' *which is obtained from the program of Figure 12.1 after removing node F.*

ie A test suite $T_3 = \{$" ", "+%0D+%4J" $\}$ satisfies the statement adequacy criterion for the program 1 but does not exercise the false branch from node D in the control flow graph model of the program.

Solution ? = Branch Testing.

BRANCH TESTING

+ Branch adequacy criterion requires each branch of the program to 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

This is equivalent to stating that every edge in 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.

$$C_{Branch} = \frac{\text{number of executed branches}}{\text{number of branches}}$$

T satisfies the branch adequacy criterion if $C_{Branch} = 1$.

* Examples:

→ $T_1 = \{$ "", "+%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 $C_{Branch} = 8/8 = 1$

note 1: Traversing all edges of a graph causes all nodes to be visited.

- so test suites that satisfy Branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program.

The converse is not true.

note 2: problem with branch testing

Assume we have forgotten the first operator '-' in the conditional statement at line 27 of program 1 resulting in the faulty expression

(digit_high == 1 || digit_low == -1)

Then the branch adequacy criterion can be satisfied (and both branches exercised) with test suites in which the first comparision evaluates always to false and only the second is varied.

Such test do not systematically exercise the first comparision and will not reveal the fault in that comparision.

solution? = Condition testing.

CONDITION TESTING

* Condition adequacy criteria overcome this problem by requiring different basic conditions of the decisions to be separately exercised.

* Basic condition adequacy criterion: requires ~~that~~ each basic condition to be covered.

definition: A test suite T for a program P covers all basic conditions of P, ie 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 atleast one test in T.

* Basic condition coverage ($C_{Basic\_condition}$) of T for P is the fraction of the total no. of truth values assumed by the basic conditions of program P during the execution of all test cases in T.

$$C_{Basic\,condition} = \frac{\left(\text{Total no. of truth values assumed by all basic conditions}\right)}{2 \times \text{no. of Basic conditions}}$$

* Basic conditions versus branches

→ Basic condition adequacy criterion can be satisfied without satisfying branch coverage

For ex;

The test suite $T_4 = \{$"first + test % 9K test % K9"$\}$ satisfies basic condition adequacy criterion, but not the branch ~~adequacy~~ condition adequacy criterion.

(∵ 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)

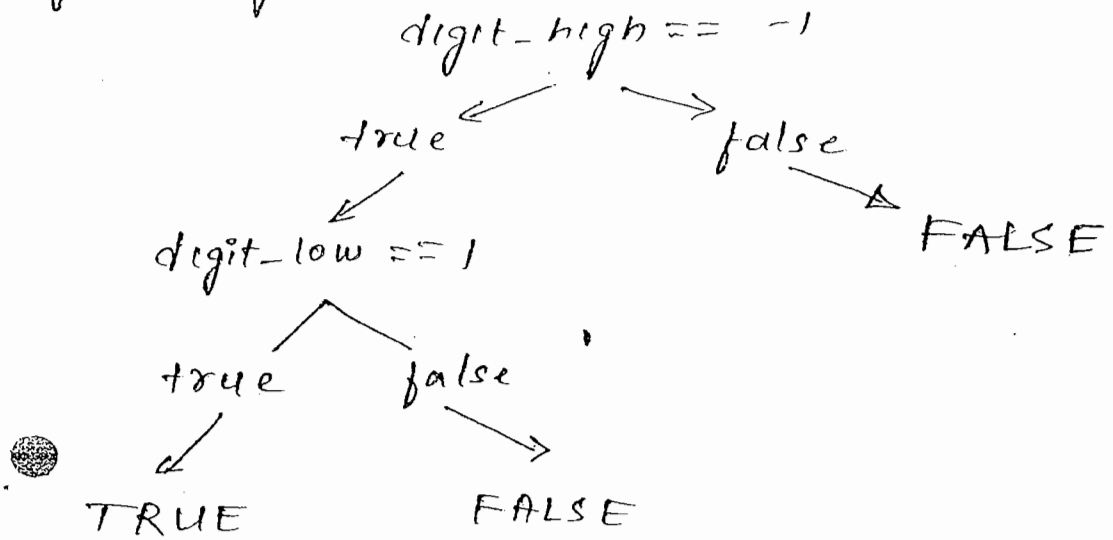
* Covering branches and conditions:

→ Branch and condition adequacy: A test suite satisfies the branch and condition adequacy criterion if it satisfies both the branch adequacy criterion and the condition adequacy criterion.

→ Compound condition adequacy:

- covers all possible evaluations of compound conditions.
- Cover all branches of a decision tree.

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



+ the no. of test cases required for compound condition adequacy grow exponentially with the no. of basic conditions in a decision, which would make the compound condition coverage impractical for programs with very complex conditions.

eg: For the expression, a && b && c && d && e, the compound condition coverage requires -

Test case	a	b	c	d	e
(1)	T	T	T	T	T
(2)	T	T	T	T	F
(3)	T	T	T	F	-
(4)	T	T	F	-	-
(5)	T	F	-	-	-
(6)	F	-	-	-	-

Modified condition / Decision coverage (MC/DC) or modified condition adequacy criterion

* MC/DC requires ~~that~~
  - → For each basic condition C, two test cases.
  - → Values of all evaluated conditions except C are the same
  - → Compound condition as a whole evaluates the true for one, and false for the other
* MC/DC can be satisfied with N+1 test cases, making it ~~attracting~~ attractive compromise b/w no. of required test cases & thoroughness of the test.

ex: For the expression (((a||b)&&c)||d)&&e

→ For compound condition adequacy

Test Case	a	b	c	d	e
(1)	True	–	True	–	True
(2)	False	True	True	–	True
(3)	True	–	False	True	True
(4)	False	True	False	True	True
(5)	False	False	–	True	True
(6)	True	–	True	–	False
(7)	False	True	True	–	False
(8)	True	–	False	True	False
(9)	False	True	False	True	False
(10)	False	False	–	True	False
(11)	True	–	False	False	–
(12)	False	True	False	False	–
(13)	False	False	–	False	–

→ For MC/DC

	a	b	c	d	e	Decision
(1)	True	–	True	–	True	True
(2)	False	True	True	–	True	True
(3)	True	–	False	True	True	True
(6)	True	–	True	–	False	False
(11)	True	–	False	False	–	False
(13)	False	False	–	False	–	False

underlined values independently affect the outcome of the decision.

Note: MC/DC is
- Basic condition coverage (CC)
- Branch coverage ~~(BC)~~ (DC)
- plus one additional condition (m)

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 paths of program P executed by atleast one test case in T

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

* Practical path coverage criteria

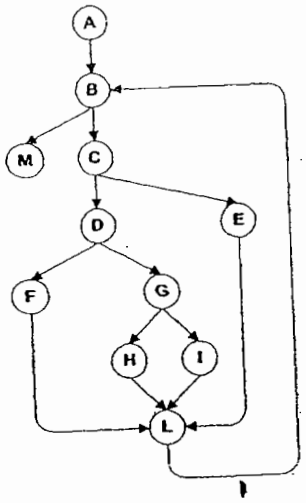
→ The no. of paths in a program with loops is unbounded, so the previously defined criterion cannot be satisfied for these programs.
For prog with loops, the denominator in the computation of path coverage is infinite, thus the path coverage becomes zero.

→ To obtain a practical criterion, it is necessary to partition the infinite set of paths into a finite number of classes and require only that representatives from each class be explored.

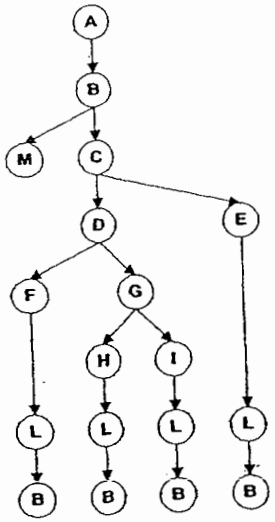
→ Useful criteria can be obtained by
- limiting the no. of paths to be covered. ie (no of traversals of loops)
- limiting the length of the paths to be traversed
- limiting the dependencies among selected paths.

* Boundary interior criterion. groups together paths that differ only in the subpath they follow when repeating the body of a loop.

* Fig below shows deriving a tree from CFG to derive subpaths for boundary/interior testing
(a) is the CFG of some 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 subpaths



(i)



(ii)

* limitations of boundary interior adequacy

```
if (a){
   S1;
}
if (b){
   S2;
}
if (c){
   S3;
}
....
if (x){
   Sn;
}
```

→ the subpaths through this control flow can include or exclude each of the statements S_i, so that in total N branches result in 2^N paths that must be traversed

→ choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent.

* 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.

Definition:
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 evaluates 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 atleast one execution, the body of the loop is repeated more than once.

* Linear code sequence and 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
$TER_{...}$ = coverage of n consequetive LCSAJs.

* Cyclomatic testing:

→ Cyclomatic number is the number of independent paths in the CFG.

- A path is representable as a bit vector, where each component of the vector represents an edge.
- "Dependence" is ordinary linear dependence b/w (bit) vectors.

- If e = number of edges,
  n = number of nodes
  c = no. 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 ~~area~~ actually been covered, and the coverage criterion is satisfied when this count reaches the cyclomatic complexity of the code under test.

PROCEDURE CALL TESTING

* The criteria considered to this point measure coverage of control flow within individual procedures. They are not well suited to integration or system testing.
* Usually it is more appropriate to choose a coverage granularity commensurate with the granularity of testing.
* procedure entry and exit testing:
  A procedure may have multiple entry points (eg FORTRAN) and multiple exit points.
* call coverage:
  The same entry point may be called from many points.

~~THE INFEASIBILITY PROBLEM~~

COMPARING STRUCTURAL TESTING CRITERIA

* power & cost of structural test adequacy criteria described earlier can be formally compared using the subsumes relation.
  The relations among these criteria are illustrated in below figure.
* They are divided into two broad categories
  → practical criteria

THE INFEASIBILITY PROBLEM

* Sometimes criteria may not be satisfiable because the criterion requires the execution of a program element that can never be executed.

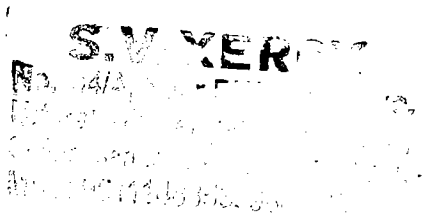
eg: - execution of statements that cannot be executed as a result of
  - Defensive programming
  - code Reuse.
- execution of conditions that cannot be satisfied as a result of interdependent conditions
- paths that cannot be executed as a result of interdependent decisions.

* large amount of "fossil" code may indicate serious 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%.
  eg 90% coverage of basic blocks, 10% allowance for infeasible blocks.

→ require justification of each element left uncovered.



UNIT 7:

TEST CASE SELECTION AND ADEQUACY, TEST EXECUTION

Syllabus

chapter 7A: Test case selection and Adequacy

* overview
* Test specifications and cases
* Adequacy criteria
* Comparing criteria

Chapter 7B: Test Execution

* overview
* From test case specifications to test cases
* Scaffolding
* Generic versus specific scaffolding.
* Test oracles
* self checks as oracles
* Capture and replay.

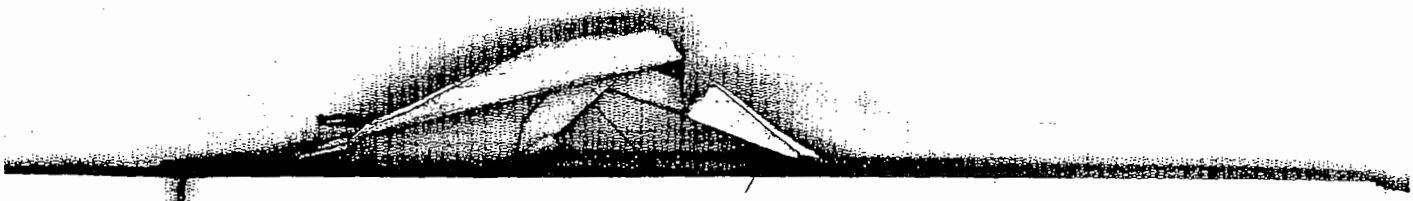
- 6 Hours.

* This unit introduces basic approaches to test case selection and corresponding adequacy criterion.

OVERVIEW

* 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.

TEST SPECIFICATIONS AND CASES



Testing Terms

While the informal meanings of words like "test" may be adequate for everyday conversation, in this context we must try to use terms in a more precise and consistent manner. Unfortunately, the terms we will need are not always used consistently in the literature, despite the existence of an IEEE standard that defines several of them. The terms we will use are defined as follows.

Test case: A *test case* is a set of inputs, execution conditions, and a pass/fail criterion. (This usage follows the IEEE standard.)

Test case specification: A test case specification is a requirement to be satisfied by one or more actual test cases. (This usage follows the IEEE standard.)

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. (This usage follows the IEEE standard.)

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. (The IEEE standard allows this and other definitions.)

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 artifact, 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.

For ex: 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 atleast one test case in the test suite.

* Satisfiability:

→ Sometimes no test suite can satisfy a criterion for a given program
ex: 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.
- Ex: modify statement coverage to require execution only of statements that can be executed.
- But we cant know for sure which are executable.

→ Approach 2: measure the extent to which a test suite approaches an adequacy criterion.
- Ex: 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.
  - Its easy to improve coverage without improving a test suite.
  - The only measure that really matters is (cost) effectiveness.

COMPARING CRITERIA

* Can we distinguish stronger from weaker adequacy criteria?
  - → Empirical approach
  - → Analytical approach
* 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 methods 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 answer to questions of relative effectiveness would describe conditions under which one adequacy criterion is guaranteed to be more effective than ~~other~~ another, or describe in statistical terms their relative effectiveness.
* The subsumes relation:
  A test adequacy criterion A subsumes test coverage criterion B iff, for every program P, every test set satisfying A w.r.t P also satisfies B w.r.t. P.

Ex: Exercising all program branches subsumes exercising all program statements.

* Uses of Adequacy criteria:

→ Test selection Approaches:

- Guidance in devising a thorough test suite
  ex: A specification-based criterion may suggest test cases covering representative combinations of values.

→ Revealing missing Tests:

- post hoc analysis: what might i have missed with this test suite?

→ often in combination:

- Ex: Design test suite from specifications, then use structural criterion (eg coverage of all branches) to highlight missed logic.

CHAPTER 7B:
TEST EXECUTION.

OVERVIEW

* Automating Test Execution

→ Designing test cases and test suites is creative.
  - like any design activity: A demanding intellectual activity, requiring human judgement.

→ Executing test cases should be automatic
  - Design once, execute many times.

→ Test automation separates the creative human process from the mechanical process of test ~~process~~ execution.

FROM TEST CASE SPECIFICATIONS TO TEST CASES

* Test design often yields test case specifications, rather than concrete data.
  - ex1: "A large positive number", not 4 20023
    ex2: "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

* Automatic generation of concrete test cases from more abstract test case specifications reduces 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 maintainence.

* Scaffoldings may include -
→ Test drivers (substituting for a main or calling population)
→ Test harness (substituting for parts of the ~~development~~ deployment environment)
→ Stubs (substituting for functionally called or used by

* The purpose of scaffolding are to provide controllability to execute test cases and observability to judge the outcome of test execution.

* Sometimes scaffolding is required to simply make a 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 thru' test cases, or sufficient observability of the effect.

~~GENERIC VERSUS SPECIFIC SCAFFOLDINGS~~

* Ex: consider an interactive program that is normally driven through a GUI. Assume that each night the person goes thru' a fully automated and unattended cycle of integration, compilation, and test execution.
It is necessary to perform some testing thru' the interactive interface, but it is neither necessary nor efficient to execute all test cases that way. small driver programs, indipendent of GUI can drive each module through large test suites in a short time.

GENERIC VERSUS SPECIFIC SCAFFOLDING

* How general should scaffolding be?
  - we could build a driver and stubs for each test case
  - or at least factor out some common code of the driver and test mgmt (eg: JUnit)
  - or further factor out some common support code, to drive a large no. of test cases from data (as in DDsteps)
  - or further, generate the data automatically from a more abstract model (eg: network traffic model)

TEST ORACLES

* Definition: The software that applies a pass/fail criterion to a program execution is called a test oracle, or simply a oracle.
* 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 it 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 failures.

* Comparision based oracle

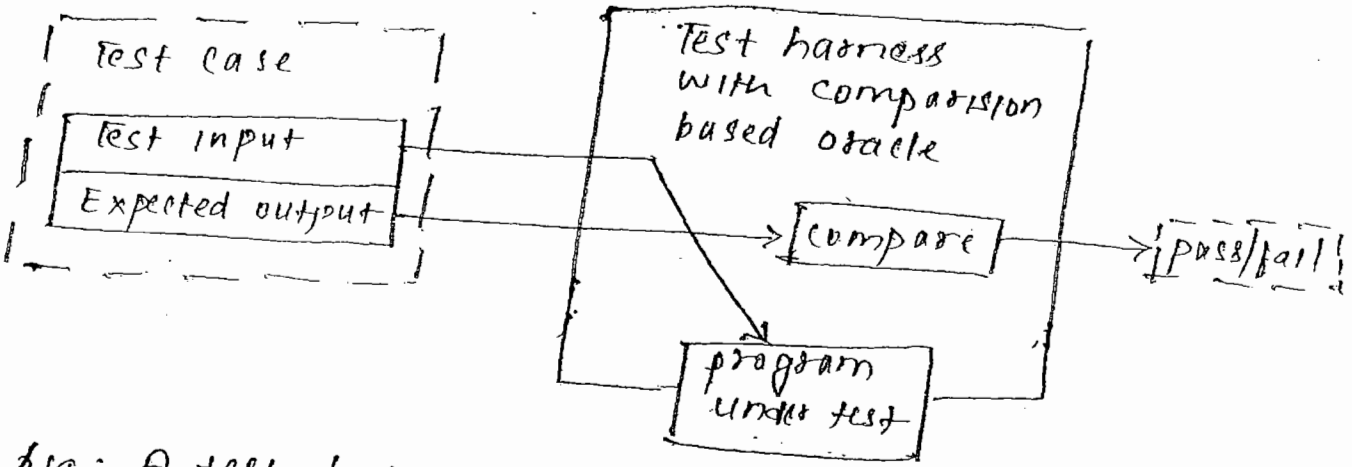


fig: A test harness with a comparision based test oracle processes test cases consisting of (program input, predicted output) pairs.

⇒ With a comparision based oracle, we need predicted output for each input
↳ (precomputed or derived)

→ oracle compares actual to predicted output, and reports failure if they differ.

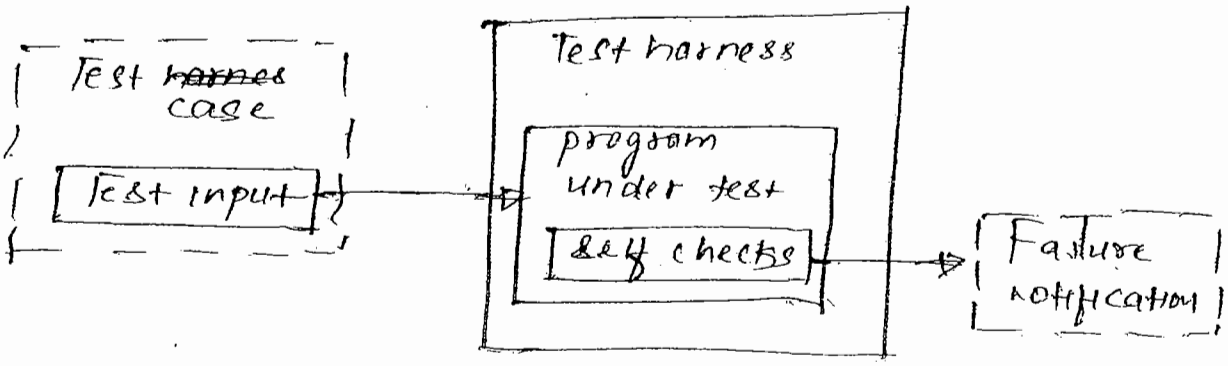
→ It is best suited for small no. of hand generated test cases. eg: for hand-[illegible]

* partial oracle

→ oracles that check results without reference to a predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others.

→ They checks necessary but not sufficient conditions for correctness

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 over all program behavior.

* Adv:
  - 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 behavior or adequately characterize correct behavior for effective self checks.
  eg: even if we separate testing program functionality from GUI, some testing of the GUI is required.
* If one cannot completely avoid human involvement in 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 (is replayed)

* 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 interaction sequences is some time possible but is generally quite limited.
