CMR INSTITUTE OF TECHNOLOGY	USN											CMRIT

Internal Assesment Test - III

Sub:	Software Testing and Practices							Code:	16MCA43
Date:	23.05.2018	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA

Answer Any **FIVE FULL** Questions

		Marks	OBE	
			CO	RBT
1	Briefly explain about functional testing and structural testing.	[10]	CO1	L4
	OR			
	Explain about quality attributes. Differentiate between testing and debugging.	[5+5]	CO2	L1
2	Generate the BOR – Constraint set and construct an abstract syntax tree of predicate Pr=(a+b)<c^!Pv(r>s). Write an algorithm to generate a minimal BOR – constraint set from an abstract syntax tree of a predicate Pr.	[10]	CO3	L1
	OR			
	Discuss the six basic principles of software testing	[10]	CO1	L2
3	State and explain the data flow diagram for the triangle problem	[10]	CO1	L4
	OR			
	Describe about SATM screens with the problem statement.	[10]	CO4	L2
4	Explain Boundary Value Analysis with an example	[10]	CO1	L2
	OR			
	Define decision table and explain with an example	[10]	CO2	L1
5	Explain about mutation analysis and fault based adequacy criteria	[10]	CO2	L2
	OR			
	Explain 'self-checks as oracles' and 'capture and replay'.	[10]	CO1	L4

1. A. Briefly explain about functional testing and structural testing.

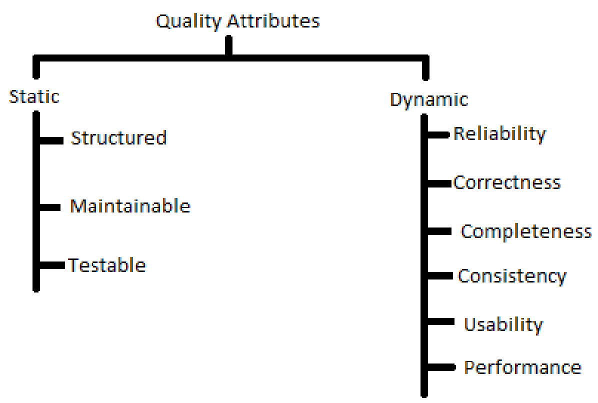
Structural and functional testing differences:

- Both Structural and Functional Technique is used to ensure adequate testing
- Structural analysis basically test the uncover error occur during the coding of the program.
- Functional analysis basically test he uncover occur during implementing requirements and design specifications.

- Functional testing basically concern about the results but not the processing.
- Structural testing is basically concern both the results and also the process.
- Structural testing is used in all the phases where design , requirements and algorithm is discussed.
- The main objective of the Structural testing to ensure that the functionality is working fine and the product is technically good enough to implement in the real environment.
- Functional testing is some times called as black box testing, no need to know about the coding of the program.
- Structural testing is some times called as white box testing because knowledge of code is very much essential. We need the understand the code written by other users.
- Various Structural Testing are

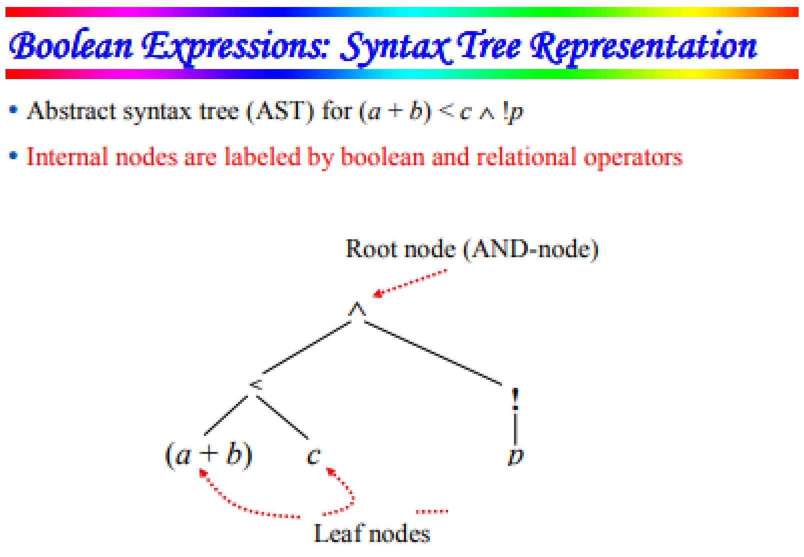
  1. **Stress Testing**
  2. **Execution Testing**
  3. **Operations Testing**
  4. **Recovery Testing**
  5. **Compliance Testing**
  6. **Security Testing**

B. Explain about quality attributes. Differentiate between testing and debugging.



Static quality attributes: structured, maintainable, testable code as well as the availability of correct and complete documentation.
Dynamic quality attributes: software reliability, correctness, completeness, consistency, usability, and performance
Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time.
Correctness will be established via requirement specification and the program text to prove that software is behaving as expected. 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. 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.
Completeness refers to the availability of all features listed in the requirements, or in the user manual. Incomplete software is one that does not fully implement all features required.
Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database.
Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing.
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."

2. A. Generate the BOR – Constraint set and construct an abstract syntax tree of predicate Pr=(a+b)<c^!Pv(r>s). Write an algorithm to generate a minimal BOR – constraint set from an abstract syntax tree of a predicate Pr.



Boolean Expressions: Syntax Tree Representation

- Abstract syntax tree (AST) for $(a + b) < c \wedge !p$
- Internal nodes are labeled by boolean and relational operators

B. Discuss the six basic principles of software testing

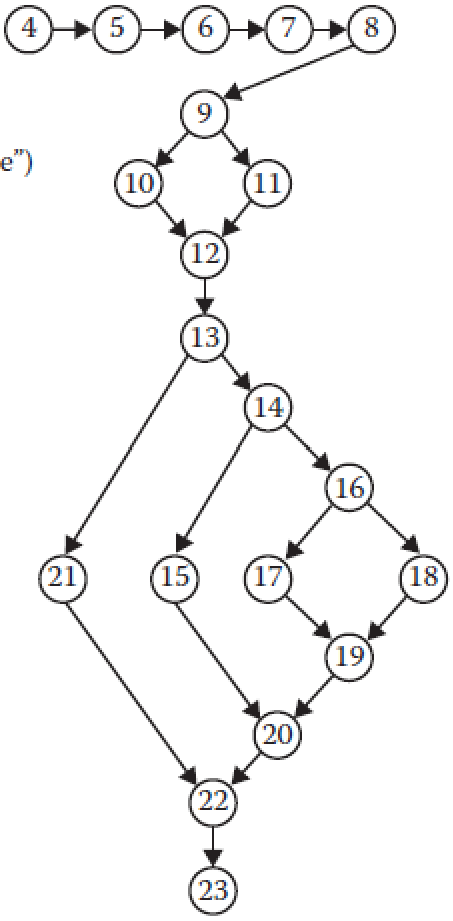
The six basic principles of software testing are:
- General engineering principles:
  - Partition: divide and conquer
  - Visibility: making information accessible
  - Feedback: tuning the development process

- Specific A&T principles:
  - Sensitivity: better to fail every time than sometimes
  - Redundancy: making intentions explicit
  - Restriction: making the problem easier

Partition: Hardware testing and verification problems can be handled by suitably partitioning the input space
Visibility: The ability to measure progress or status against goals. X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?"
Feedback: The ability to measure progress or status against goals
X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?"
Sensitivity: A test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria). Run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine
Redundancy: Redundant checks can increase the capabilities of catching specific faults early or more efficiently.
e.g, Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.
Restriction: Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems

3. A. State and explain the data flow diagram for the triangle problem

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14    Then  If (a = b) AND (b = c)
15          Then Output ("Equilateral")
16          Else  If (a≠b) AND (a≠c) AND (b≠c)
17                Then Output ("Scalene")
18                Else Output ("Isosceles")
19             EndIf
20          EndIf
21   Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```



B. Describe about SATM screens with the problem statement.

The **SATM** system communicates with bank customers via the 15 screens shown in Figure 2.4. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account. When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept. At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept. On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount. If a problem occurs with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14. If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14. When the "No" button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from the card slot, screen 1 is displayed. When the "Yes" button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

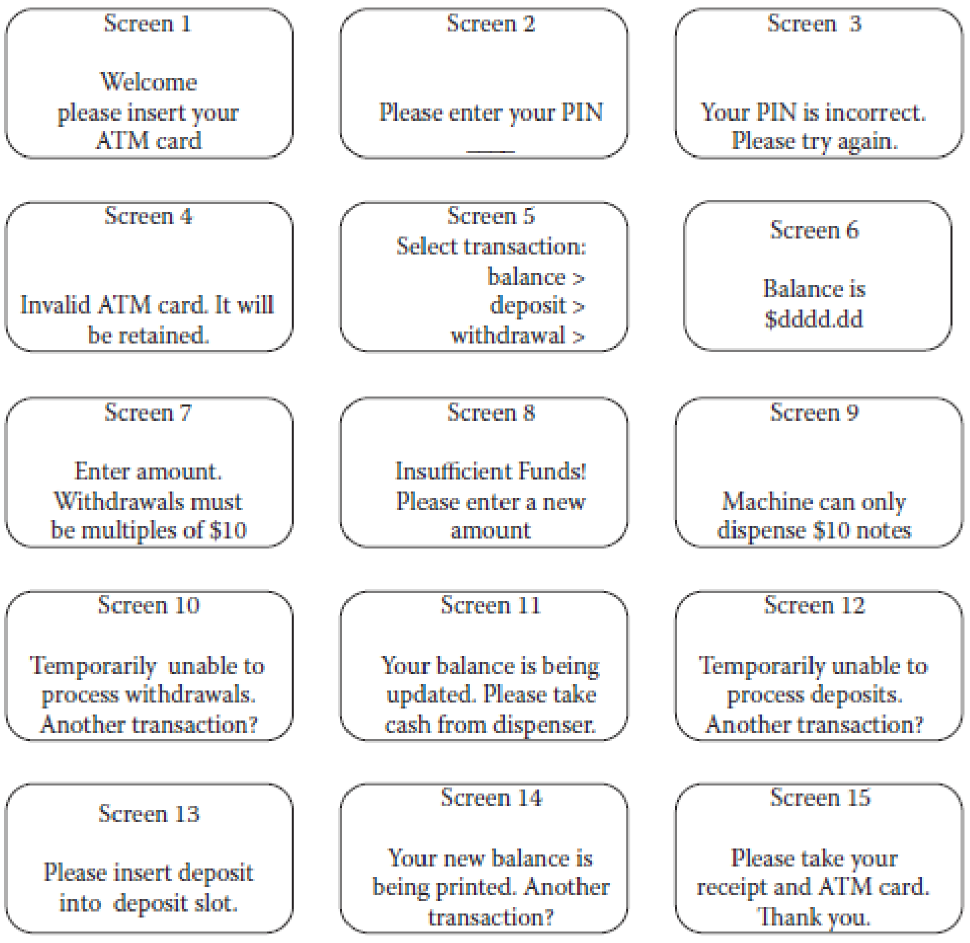
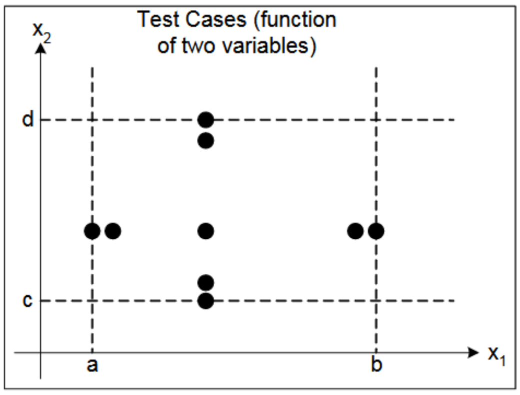

Screen 1
Welcome please insert your ATM card
Screen 2
Please enter your PIN ____
Screen 3
Your PIN is incorrect. Please try again.
Screen 4
Invalid ATM card. It will be retained.
Screen 5
Select transaction:
balance >
deposit >
withdrawal >
Screen 6
Balance is $dddd.dd
Screen 7
Enter amount. Withdrawals must be multiples of $10
Screen 8
Insufficient Funds! Please enter a new amount
Screen 9
Machine can only dispense $10 notes
Screen 10
Temporarily unable to process withdrawals. Another transaction?
Screen 11
Your balance is being updated. Please take cash from dispenser.
Screen 12
Temporarily unable to process deposits. Another transaction?
Screen 13
Please insert deposit into deposit slot.
Screen 14
Your new balance is being printed. Another transaction?
Screen 15
Please take your receipt and ATM card. Thank you.

Figure 2.4 SATM screens.

4. A. Explain Boundary Value Analysis with an example

5. **BVA test case for two variables functions**
6. In the general application of Boundary Value Analysis can be done in a uniform manner.
7. The basic form of implementation is to maintain all but one of the variables at their
8. nominal (normal or average) values and allowing the remaining variable to take on its
9. extreme values. The values used to test the extremities are:
10. •Min ---------------------------------- - Minimal
11. •Min+ ---------------------------------- - Just above Minimal
12. •Nom ---------------------------------- - Average

13. •Max- ------------------------------------ - Just below Maximum
14. •Max ------------------------------------ - Maximum
15. $<x_{1nom}, x_{2min}>$ $<x_{1nom}, x_{2min+}>$ $<x_{1nom}, x_{2nom}>$ $<x_{1nom}, x_{2max-}>$ $<x_{1nom}, x_{2max}>$
$<x_{1min}, x_{2nom}>$ $<x_{1min+}, x_{2nom}>$ $<x_{1max-}, x_{2nom}>$ $<x_{1max}, x_{2nom}>$
16. 

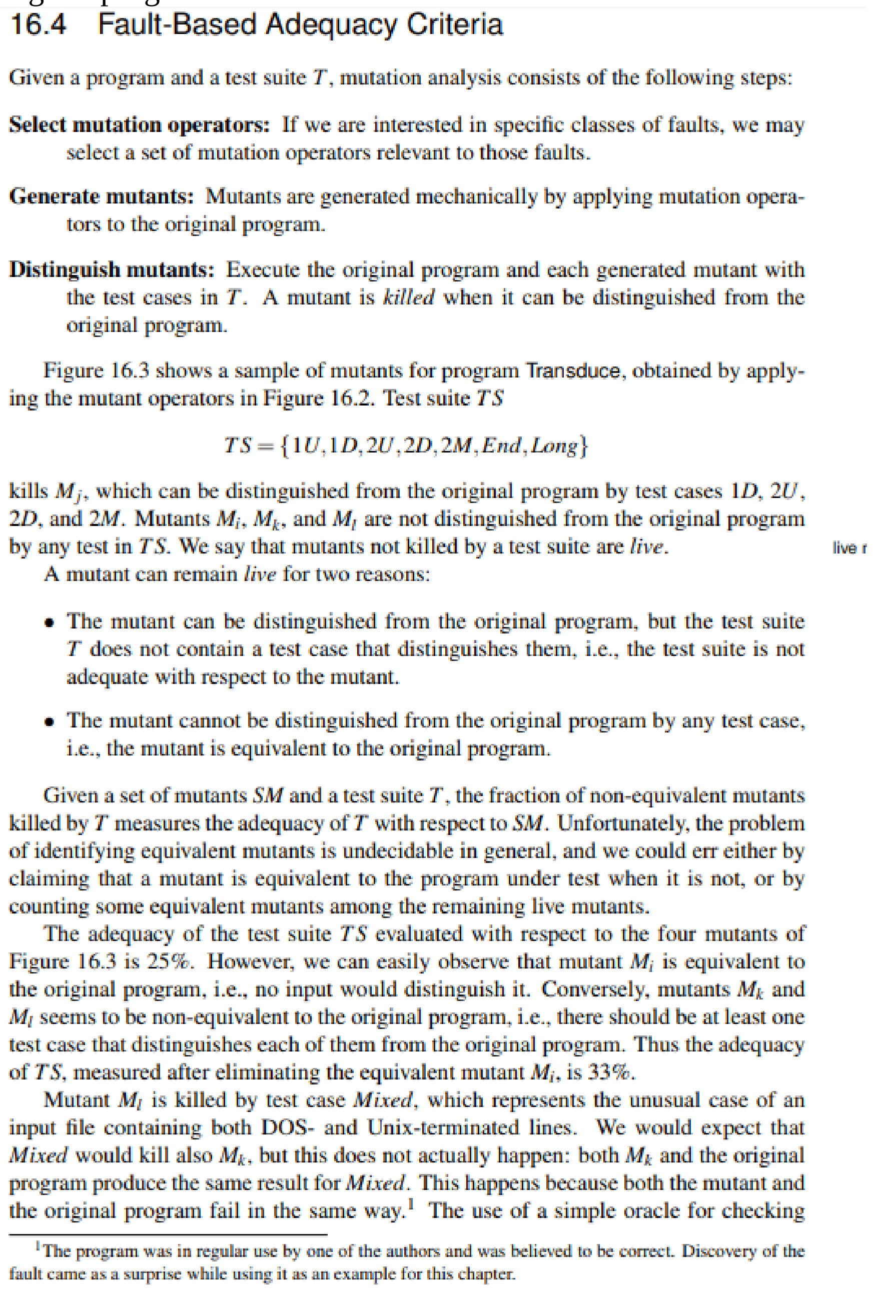
17. **<u>Limitations of BVA</u>**
18. Boundary Value Analysis works well when the Program Under Test (PUT) is a “function of several independent variables that represent bounded physical quantities” [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results. For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester’s intuition
19. and common sense shows that we require more emphasis towards the end of February or on leap years.
20. The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables. This lack of intuition or understanding for the variable nature means that BVA can be seen as quite rudimentary.

B. Define decision table and explain with an example

RULES		**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**
Conditions	C1: a < b + c	F	T	T	T	T	T	T	T	T	T	T
	C2 : b < a + c	-	F	T	T	T	T	T	T	T	T	T
	C3 : c < a + b	-	-	F	T	T	T	T	T	T	T	T
	C4 : a = b	-	-	-	T	T	T	T	F	F	F	F
	C5 : a = c	-	-	-	T	T	F	F	T	T	F	F
	C6 : b = c	-	-	-	T	F	T	F	T	F	T	F
Actions	a1 : Not a triangle	X	X	X								
	a2 : Scalene triangle											X
	a3 : Isosceles triangle							X		X	X	
	a4 : Equilateral triangle				X							
	a5 : Impossible					X	X		X			

5. a. Explain about mutation analysis and fault based adequacy criteria

Fault Based Testing: Terminology Original program: The program unit (e.g., C function or Java class) to be tested. Program location: A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and boolean expressions, and procedure calls. Alternate expression: Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors). Alternate program: A program obtained from the original program by substituting an alternate expression for the text at some program location. Distinct behavior of an alternate program R for a test t: The behavior of an alternate program R is distinct from the behavior of the original program P for a test t, if R and P produce a different result for t, or if the output of R is not defined for t. Distinguished set of alternate programs for a test suite T: A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in T.
Mutation Analysis: Terminology Original program under test: The program or procedure (function) to be tested. Mutant: A program that differs from the original program for one syntactic element, e.g., a statement, a condition, a variable, a label, etc. Distinguished mutant: A mutant that can be distinguished for the original program by executing at least one test case. Equivalent mutant: A mutant that cannot be distinguished from the original program. Mutation operator: A rule for producing a mutant program by syntactically modifying the original program.



16.4 Fault-Based Adequacy Criteria

Given a program and a test suite T, mutation analysis consists of the following steps:

Select mutation operators: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants: Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants: Execute the original program and each generated mutant with the test cases in T. A mutant is *killed* when it can be distinguished from the original program.

Figure 16.3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in Figure 16.2. Test suite TS

$$TS = \{1U, 1D, 2U, 2D, 2M, End, Long\}$$

kills M_j, which can be distinguished from the original program by test cases $1D$, $2U$, $2D$, and $2M$. Mutants M_i, M_k, and M_l are not distinguished from the original program by any test in TS. We say that mutants not killed by a test suite are *live*. live r

A mutant can remain *live* for two reasons:

- The mutant can be distinguished from the original program, but the test suite T does not contain a test case that distinguishes them, i.e., the test suite is not adequate with respect to the mutant.
- The mutant cannot be distinguished from the original program by any test case, i.e., the mutant is equivalent to the original program.

Given a set of mutants SM and a test suite T, the fraction of non-equivalent mutants killed by T measures the adequacy of T with respect to SM. Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not, or by counting some equivalent mutants among the remaining live mutants.

The adequacy of the test suite TS evaluated with respect to the four mutants of Figure 16.3 is 25%. However, we can easily observe that mutant M_i is equivalent to the original program, i.e., no input would distinguish it. Conversely, mutants M_k and M_l seems to be non-equivalent to the original program, i.e., there should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of TS, measured after eliminating the equivalent mutant M_i, is 33%.

Mutant M_l is killed by test case *Mixed*, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines. We would expect that *Mixed* would kill also M_k, but this does not actually happen: both M_k and the original program produce the same result for *Mixed*. This happens because both the mutant and the original program fail in the same way.[1] The use of a simple oracle for checking

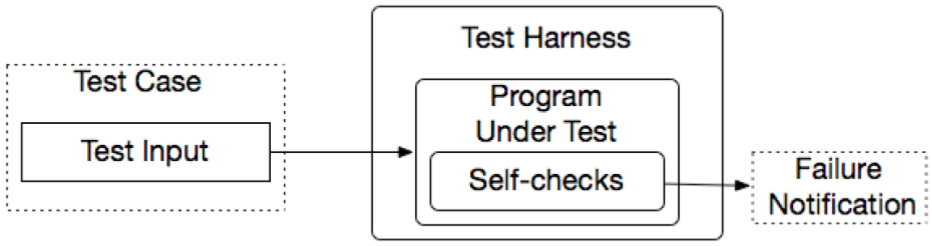
[1]The program was in regular use by one of the authors and was believed to be correct. Discovery of the fault came as a surprise while using it as an example for this chapter.

b. Explain ‘self-checks as oracles’ and ‘capture and replay’.

SELF-CHECKS AS ORACLES

 An oracle can also be written as self checks
-Often possible to judge correctness without predicting results.
 Typically these self checks are in the form of assertions, but designed to be checked during execution.
 It is generally considered good design practice to make assertions and self checks to be free of side effects on program state.
 Self checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specification rather than all program behaviour.
 Devising the program assertions that correspond in a natural way to specifications poses two main challenges:
Bridging the gap between concrete execution values and abstractions used in specification
Dealing in a reasonable way with quantification over collection of values
Structural invariants are good candidates for self checks implemented as assertions
 They pertain directly to the concrete data structure implementation
 It is sometimes straight-forward to translate quantification in a specification statement into iteration in a program assertion
 A run time assertion system must manage ghost variables
 They must retain “before” values
 They must ensure that they have no side effects outside assertion checking
 Advantages:
-Usable with large, automatically generated test suites.
 Limits:

-often it is only a partial check. -recognizes many or most failures, but not all.



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