CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - II

Sub:	Software Testing							Code:	10CS842
Date:	10/ 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	8-A,B	Branch:	CSE
Answer Any FIVE FULL Questions									

		Marks	OBE	
			CO	RBT
1	Explain Slice Based testing using Sales commission problem	[10]	CO3	L3
2	Define Predicate node, du-path, dc-path. Give du-path for stocks, locks, totallocks, sales and commission for commission sales problem	[10]	CO2	L3
3 (a)	Explain mutation analysis software fault based testing	[7]	CO2	L3
3 (b)	Define test oracle and partial oracle	[3]	CO2	L1
4	Explain verification trade off dimensions	[10]	CO3	L3
5 (a)	Briefly discuss the dependability properties in process framework	[7]	CO3	L3
5 (b)	List the fault based adequacy criteria	[3]	CO3	L3
6 (a)	Why organizational factors are needed in process framework?	[6]	CO3	L3
6 (b)	Define the terms Distinct, Distinguished, Alternate program & Expression,	[4]	CO1	L3
7	Explain the Six basic principles of Testing	[10]	CO3	L3
8 (a)	What is scaffolding? Describe generic & application specific scaffolding.	[6]	CO4	L3
(b)	Define All C-uses/Some P-uses, All P-uses/Some C-uses, All Defs	[4]	CO4	L3

Course Name / Code	Software Testing / 10CS842

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C842.1	Describe the Terminology & levels of testing	1	2	1	0	0	0	0	1	0	1	1	0
C842.1	Write the Test Document, Scenario, Case, Plan	1	1	2	1	0	0	0	1	2	1	0	0
C842.1	Explain the Software testing process, Techniques with examples	1	2	1	1	0	0	0	1	0	1	0	0
C842.1	Describe the process framework-Validation, Verification and Basic principles	1	1	1	1	0	0	0	1	0	1	0	0
C842.1	Explain the process by using Testing tools	1	1	1	1	2	0	0	1	1	1	2	0
C842.1	Demonstrate the process improvement in software testing	1	1	1	2	0	0	0	1	1	3	2	0

Revised Bloom's Taxonomy (RBT)		Programme Outcome
Cognitive level	KEYWORDS	PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.	
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend	
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.	
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.	
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.	

1. **Explain Slice Based testing with an example**.

Slice Based Testing

• The second type of data flow testing

• A program slice is a set of program statements that contribute to, or affect a value for a variable at some point in the program

• The idea of slicing is to divide a program into components that have some useful meaning

• Definition:

– Given a program P, and a program graph G(P) in which statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n,*** written S(V,n) is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.

• "prior to": a slice captures the execution time behaviour of a program with respect to the variable(s) in the slice

• "contribute": some declarative statements have an effect on the value of a variable (e.g. const, type)

– We will exclude those non-executable statements

The USE relationship pertains to five forms of usage:

– P-use: used in a predicate (decision)

– C-use: used in a computation

– O-use: used for output

– L-use: used for location (e.g. pointers)

– I-use: used for iteration (internal counters, loop indices)

• Definition nodes:

– I-def: defined by input

– A-def: defined by assignment

• Eventually we will develop a DAG (directed acyclic graph) of slices in which nodes are slices and edges correspond to the subset relationship

• **Guidelines for choosing slices**:

– If the value of v is the same whether the statement fragment is included or excluded, exclude the fragment.

– If statement fragment n is:

• a defining node for v, include n in the slice

• a usage node for v, exclude n from the slice

– O-use, L-use & I-use nodes are excluded from slices

Example of Slice-Based Testing: ***The Commission Program***

- S_1: S(locks,13) = {13} (defining node I-def)
- S_2: S(locks,14) = {13,14,19,20}
- S_3: S(locks,16) = {13,14,16,19,20}
- S_4: S(locks,19) = {19} (defining node I-def)
- S_5: S(stocks,15) = {13,14,15,19,20}
- S_6: S(stocks,17) = {13,14,15,17,19,20}
- S_7: S(barrels,15) = {13,14,15,19,20}
- S_8: S(barrels,18) = {13,14,15,18,19,20}
- S_9: S(totallocks,10) = {10} (A-def)
- S_{10}: S(totallocks,16) = {10,13,14,16,19,20} (A-def & C-use)

S(totallocks,21)= {10,13,14,16,19,20} 21 is an O-Use of totallocks, excluded

- S_{11}: S(totallocks,24) = {10,13,14,16,19,20} (24 is a C-use of total locks)
- S_{12}: S(totalstocks,11) = {11} (A-def)
- S_{13}: S(totalstocks,17) = {11,13,14,15,17,19,20} (A-def & C-use)
- S_{14}: S(totalstocks,22) = {11,13,14,15,17,19,20}(22 is an O-Use of totalstocks)
- S_{15}: S(totalbarrels,12) = {12}
- S_{16}: S(totalbarrels,18) = {12,13,14,15,18,19,20} (A-def & C-use)
- S_{17}: S(totalbarrels,23) = {12,13,14,15,18,19,20} (23 is an O-Use of totalbarrels)
- S_{18}: S(lock_price,24) = {7} (A-def)
- S_{19}: S(stock_price,25) = {8}
- S_{20}: S(barrel_price,26) = {9}
- S_{21}: S(lock_sales,24) = {7,10,13,14,16,19,20,24}
- S_{22}: S(Stock_sales,25) = {8,11,13,14,15,17,19,20,25}
- S_{23}: S(barrel_sales,26) = {9,12,13,14,15,18,19,20,26}
- S_{24}: S(sales,27) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{25}: S(sales,28) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{26}: S(sales,29) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{27}: S(sales,33) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{28}: S(sales,34) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{29}: S(sales,37) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}
- S_{30}: S(sales,38) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25,26,27}

– $S_{24} = S_{10} \cup S_{13} \cup S_{16} \cup S_{21} \cup S_{22} \cup S_{23}$

– If the value of sales is wrong, we first look at how it is computed, and if this is OK, we check how the components are computed

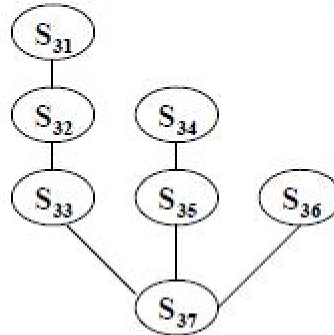
- S_{31}: S(commission,31) = {31}
- S_{32}: S(commission,32) = {31,32}
- S_{33}: S(commission,33) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25, 26,27,29,30,31,32,33}
- S_{34}: S(commission,36) = {36}

• S_{35}: S(commission,37) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25, 26,27,36,37}

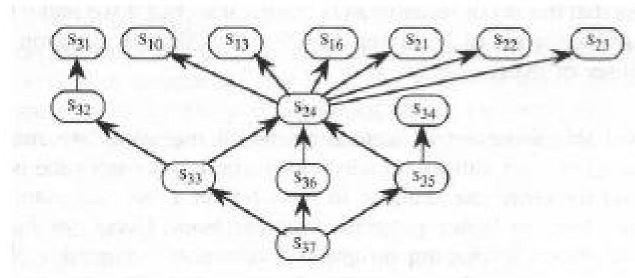
• S_{36}: S(commission,38) = {7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25, 26,27,29,34,38}

• S_{37}: S(commission,41) ={7,8,9,10,11,12,13,14,15,16,17,18,19,20,24,25, 26,27,29,30,31,32,33,34,35,36,37,38}

- Lattice of slices in the commission problem



- The lattice is a DAG in which slices are nodes and an edge represents the proper subset relationship



2. **Defining node Usage node , Predicate node, Du-path, Dc-path**

Definition:

a) – Node n ÎG(P) is a ***defining node*** of the variable v Î V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

• For example: input , assignment, loop control statements (for int i=0;i<10;i++) and procedure calls are defining nodes

• When the code corresponding to such statements executes,
contents of the memory location associated with v is changed

b) – Node n ÎG(P) is a ***usage node*** of the variable v Î V, written as USE(v,n), iff the value of the variable v is used at the statement fragment corresponding to node n.

• For example: output , assignment(i:=i+1), condition, loop control statements and procedure calls are usage nodes
• When the code corresponding to such statements executes, contents of the memory location associated with v is not changed

c) – A usage node USE(v,n) is a ***predicate use*** (denoted as P-use), iff the statement n is a predicate statement; otherwise USE(v,n) is a *computation use*, (denoted C-use)
d) – A ***definition-use (sub) path*** with respect to a variable v (denoted du-path) is a (sub) path in PATHS(P) such that for some v Î V, there are define and usage nodes DEF(v,m) & USE(v,n) such that m & n are the initial and final nodes of the (sub) path.
e) – A ***definition-clear (sub) path*** with respect to a variable v (denoted dc-path) is a definition-use(sub) path in PATHS(P) with initial and final nodes DEF(v,m) & USE(v,n) such that no other node in the (sub) path is a defining node of v
  • Du-paths that are not definition-clear are potential trouble spots!

Explain the Du-path for Commission problem. (Or) Define/Use Testing

Variable ***stocks***:
– We have DEF(stocks,15) & USE (stocks,17)
• Hence path <15,17> is a du-path with respect to stocks
• p0 = <25,27> is also decision-clear, a dc-path

• Variable ***locks***:
– We have DEF(locks,13), DEF(locks,19),
USE(locks,14) & USE(locks,16): 4 du-paths(dc too!)
• p1 = <13,14>
• p2 = <13,14,15,16>
• p3 = <19,20,14>
• p4 = < 19,20,14,15,16>
• We could extend p1 and p3 to include node 21:
• p1‘ = <13,14,21> & p3‘ = <19,20,14,21>
– Then p1`,p2,p3` & p4 form a very complete set of test cases for the while loop
» Bypass the loop » Begin the loop » Repeat the loop » Exit the loop

Variable ***totallocks***:
– We have: • DEF(totallocks,10) & DEF(totallocks,16)
• USE(totallocks,16), USE(totallocks,21),USE(totallocks,24)
• We could expect 6 du-path, why?
• There are only 5:
– p5 = <10,11,12,13,14,15,16> (also a dc-path)
– p6 = <10,11,12,13,14,15,16,17,18,19,20,21>
» The subpath: <16,17,18.19.20,14,15> might be traversed several times

» This is not a dc-path, if there is a problem with the value of totallocks at node 21, we should look at the defining node, 16

– p7 = **< 10,11,12,13,14,15,16,17,18,** 19,20,21,22,23,24**>**

» p7 = <p6, 22,23,24 >, not a dc-path

– <16,16> is degenerate, disregard it

– p8 = < 16,17,18,19,20,21> this is a dc-path

– p9 = < 16,17,18,19,20,21,23,24> this is a dc-path

• p8 & p9 have the same loop iteration problem as p6

Variable ***sales***:

– DEF(sales,27), USE(sales,28), USE(sales,29),
USE(sales,33), USE(sales,34) , USE(sales,37) ,USE(sales,38)

– p10 = <27,28>

– p11 = <27,28,29>

– p12 = < 27,28,29, 30,31,32,33>

» This is a dc-path covering p10 & p11 » If we test with p12, we will cover p10 & p11 too

– p13 = < 27,28,29,34>

– p14 = < 27,28,29,34,35,36,37>

– p15 = < 27,28,29,38>

• Variable ***commission***:

• Since 31,32 & 33 could be replaced by:

– Commission := 220 + 0.2 * (sales - 1800), then 33 could be considered the defining node

• Same for 36,37 could be considered the defining node

– DEF(commission, 33), DEF(commission, 37),
DEF(commission, 38), USE(commission, 41)

– p16 = <33,41> (a dc-path) – p17 = <37,41> (a dc-path) – p18 = <38,41> (a dc-path)

Understand by comparing the program and Define Use Variables

Table 10.2 Define/Use Nodes for Variables in the Commission Problem

Variable	*Defined at Node*	*Used at Node*
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

```
1   Program Commission (INPUT,OUTPUT)
2       Dim locks, stocks, barrels As Integer
3       Dim lockPrice, stockPrice, barrelPrice As Real
4       Dim totalLocks, totalStocks, totalBarrels As Integer
5       Dim lockSales, stockSales, barrelSales As Real
6       Dim sales, commission As Real
7       lockPrice = 45.0
8       stockPrice = 30.0
9       barrelPrice = 25.0
10      totalLocks = 0
11      totalStocks = 0
12      totalBarrels = 0
13      Input(locks)
14      While NOT(locks = -1)    'loop condition uses -1 to indicate end of data
15          Input(stocks, barrels)
16          totalLocks = totalLocks + locks
17          totalStocks = totalStocks + stocks
18          totalBarrels = totalBarrels + barrels
19          Input(locks)
20      EndWhile
21      Output("Locks sold: ", totalLocks)
22      Output("Stocks sold: ", totalStocks)
23      Output("Barrels sold: ", totalBarrels)
24      lockSales = lockPrice*totalLocks
25      stockSales = stockPrice*totalStocks
26      barrelSales = barrelPrice * totalBarrels
27      sales = lockSales + stockSales + barrelSales
28      Output("Total sales: ", sales)
29      If (sales > 1800.0)
30          Then
31              commission = 0.10 * 1000.0
32              commission = commission + 0.15 * 800.0
33          commission = commission + 0.20*(sales-1800.0)
34      Else If (sales > 1000.0)
35          Then
36                  commission = 0.10 * 1000.0
37                  commission = commission + 0.15*(sales-1000.0)
38              Else commission = 0.10 * sales
39              EndIf
40      EndIf
41      Output("Commission is $", commission)
42      End Commission
```

3. Explain mutation analysis software fault based testing (Or) Explain mutation testing.

Mutation testing (or *Mutation analysis* or *Program mutation*) is used to design new software tests and evaluate the quality of existing software tests.

Mutation testing involves modifying a program in small ways.

A *mutant* is a copy of a program with a *mutation*
• A *mutation* is a syntactic change (a seeded bug)
– Example: change (i < 0) to (i <= 0)
• Run test suite on all the mutant programs
• A mutant is *killed* if it fails on at least one test case
• If many mutants are killed, infer that the test suite is also effective at finding real bugs

.

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.

The patterns mutation operator for changing program text is called mutation operators. We say a mutant is valid if it is syntactically **useful mutant** correct. We say a mutant is useful if, in addition to being **valid mutant**, its behavior differs from the behavior of the original program for no more than a small subset of program test cases.

Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by abs(e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

Write all together 10 modification irrespective of Operand, expression and statement.

b) Software that applies a pass/fail criterion to a program execution is called a **test oracle**, often shortened to oracle. Oracle that checks result without reference to a predicted output are often partial, which is called **partial oracle.**

4. Explain verification trade-off Dimensions. (Or) Degree of freedom

The activities for assuring the correctness of reactive systems reside within the Validation and Verification (V&V) process. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements.

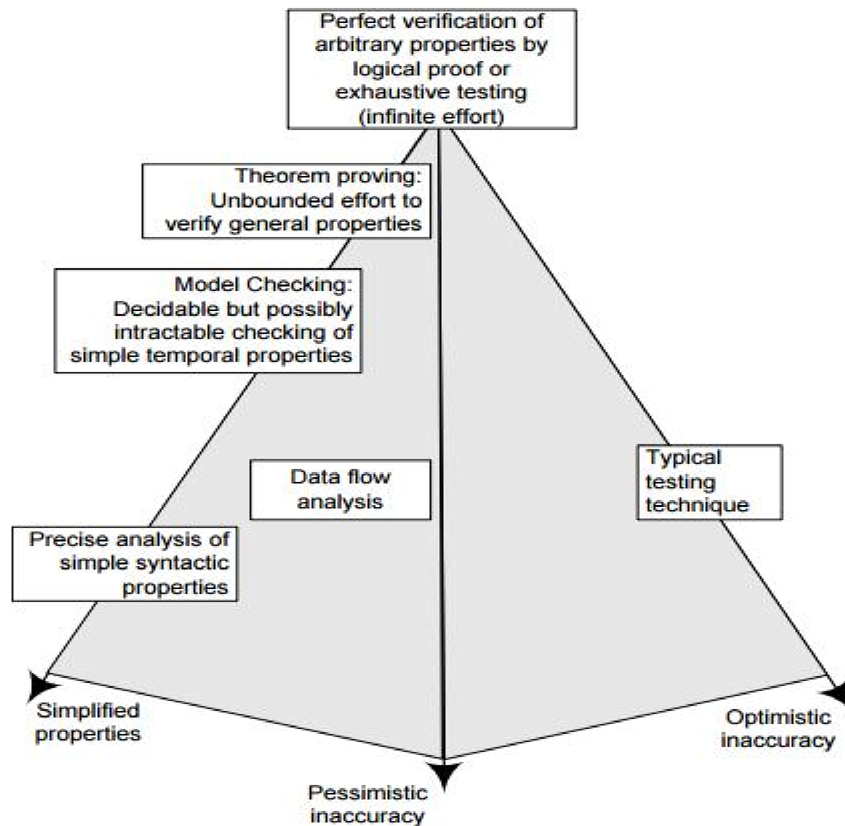
Given a precise specification and a program, it seems that one ought to be able to arrive at some logically sound argument or proof that a program satisfies the specified properties.

Undeciability of Correctness Properties

- Correctness properties are not decidable.
  - Halting problem can be embedded in almost every property of interest.

For some properties and some very simple programs, it is in fact possible to obtain a logical correctness argument, albeit at high cost. In a few domains, logical correctness arguments may even be cost-effective for a few isolated, critical components (e.g., a safety interlock in a medical device). In general, though, one cannot produce a complete logical "proof" for the full specification of practical programs in full detail. This is not just a sign that technology for verification is immature. It is, rather, a consequence of one of the most fundamental properties of computation.

In theory, undecidability of a property S merely implies that for each verification technique for checking S, there is at least one "pathological" program for which that technique cannot obtain a correct answer in finite time. It does not imply that verification will always fail or even that it will usually fail, only that it will fail in at least one case.



Program testing is a verification technique and is as vulnerable to undecidability as other techniques. Exhaustive testing, that is, executing and checking every possible behaviour of a program, would be a "proof by cases," which is a perfectly legitimate way to construct a logical proof.

A technique for verifying a property can be inaccurate in one of two directions. It may be pessimistic, meaning that it is not guaranteed to accept a program optimistic even if the program does possess the property being analyzed, or it can be optimistic if it may accept some programs that do not possess the property (i.e., it may not detect all violations). Testing is the classic optimistic technique, because no finite number of tests can guarantee correctness. Many automated program analysis techniques for properties of program behaviors3 are pessimistic with respect to the properties they are designed to verify. Some analysis techniques may give a third possible answer, "don't know." We can consider these techniques to be either optimistic or pessimistic depending on how we interpret the "don't know" result. Perfection is unobtainable, but one can choose techniques that err in only a particular direction. A software verification technique that errs only in the pessimistic direction is called a conservative analysis. It might seem that a conservative analysis would always be preferable to one that could accept a faulty program.

5. Briefly discuss the dependability properties in process framework (or) Dependability properties

A program is **"correct"** if it is consistent with its specification, i.e., if it does exactly what the specification says it must do. It is therefore a consistency relation between two things, the specification and the program. It is impossible to say whether a program is correct in the absence of a specification, although the specification may be informal or implicit.

Reliability is a way of statistically approximating correctness. Reliability can be stated in different ways. Classical reliability is often stated in terms of time, e.g., **mean time between failures (MTBF) or availability (likelihood of correct functioning at any given time**).

Time-based reliability measures are often used for continuously functioning software (e.g., an operating system or network interface), but for other software "time" is often replaced by a usage-based measure (e.g., number of executions). For example, mean time between failures (MTBF) is a statement about the likelihood of failing before a given point in time (but "time" may be measured in number of uses or some other way).

Availability is the likelihood of correct functioning at any particular point in time. Reliability describes the behavior of a program, which may not be correlated to structural measures of quality.

A program is reliable but not correct when failures occur rarely. (A "failure" is any behaviour that is not permitted by the specification.)

A program may be correct without being safe or robust if the specification is inadequate, in the sense that the specification does not rule out some undesirable behaviours.

A particularly common way in which a program can be correct (or at least reliable) without being safe or robust is when the specification is only partly defined

Safety is a sub-category of robustness specifically concerned with avoiding certain very bad behaviors. The undesired outcomes are called **"hazards,"** and safety engineering is concerned with identifying and preventing hazards. Sometimes this literally means "human safety,"

A system is **robust** if it acts reasonably in severe or unusual conditions. It is not possible to give a precise definition of robustness, but one characteristic of robust systems is that their specifications include "desired reactions to undesirable situations". Robustness is often (but not always) concerned with partial functionality, also called "graceful degradation."

"Fail soft" is the same as the aforementioned "graceful degradation," i.e., maintaining some level of useful functionality despite partial system failure.

"Fail safe" is avoidance of harmful behaviour, perhaps without providing any useful functionality at all. In many cases, this simply means shutting down to avoid doing harm.

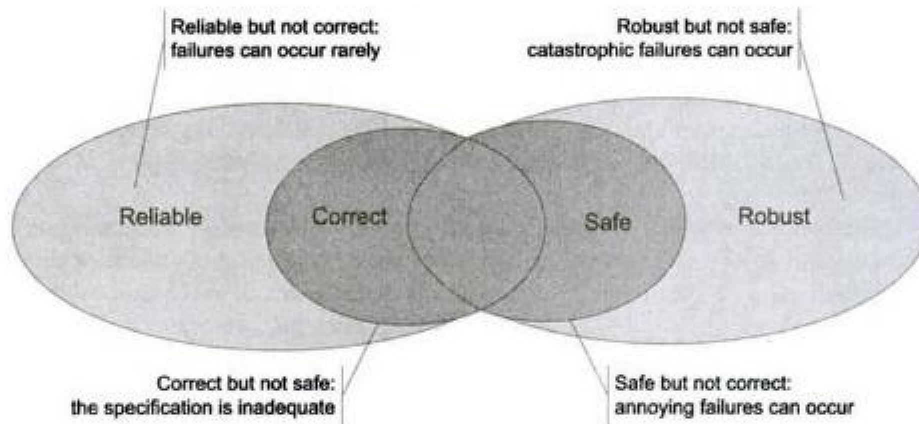


Figure 4.1: Relation among dependability properties

b) **List the fault based adequacy criteria**

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.

We say that mutants not killed by a test suite are live.
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.

6. **Why organizational factors are needed in process framework?**

Organizational factors Organizational factors
• Different teams for development and quality?
– separate development and quality teams is common in large organizations
– indistinguishable roles is postulated by some methodologies (extreme programming)

• Different roles for development and quality?
– test designer is a specific role in many organizations
– mobility of people and roles by rotating engineers over development and testing tasks among different projects is a possible option

Example of Allocation of Responsibilities • Allocating tasks and responsibilities is a complex job:

we can allocate

– Unit testing

• to the development team (requires detailed knowledge of the code)
• but the quality team may control the results (structural coverage)

– Integration, system and acceptance testing

• to the quality team
• but the development team may produce scaffolding and oracles

– Inspection and walk-through

• to mixed teams

– Regression testing

• to quality and maintenance teams

– Process improvement related activities

• to external specialists interacting with all teams

Allocation of Responsibilities and rewarding mechanisms: case A

• **allocation of responsibilities**
• Development team responsible development measured with LOC per person month
• Quality team responsible for quality
• **possible effect**
• Development team tries to maximize productivity, without considering quality
• Quality team will not have enough resources for bad quality products
• **result**
• product of bad quality and overall project failure

Allocation of Responsibilities and rewarding mechanisms: case B

• **allocation of responsibilities**
• Development team responsible for both development and quality control
• **possible effect**
• the problem of case A is solved
• but the team may delay testing for development without leaving enough resources for testing
• **result**
• delivery of a not fully tested product and overall project failure

b) Define the below terms

1. **Alternate Expression**
2. **Alternate Program**
3. **Distinct Behaviour**
4. **Distinguished Behaviour**

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*.

7. **Explain the basic six principles.**

- **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

1) **Sensitivity: better to fail every time than sometimes**

   Consistency helps:

   a. 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)

   b. 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.

2) **Redundancy: making intentions explicit**

   Redundant checks can increase the capabilities of catching specific faults early or more efficiently.

   a. Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.

   b. Validation of requirement specifications is redundant with respect to validation of the final software, but can reveal errors earlier and more efficiently.

c. Testing and proof of properties are redundant, but are often used together to increase confidence

3) **Partition: divide and conquer**
   Hard testing and verification problems can be handled by suitably partitioning the input space:
   a. both structural and functional test selection criteria identify suitable partitions of code or specifications (partitions drive the sampling of the input space)
   b. verification techniques fold the input space according to specific characteristics, grouping homogeneous data together and determining partitions
4) **Restriction: making the problem easier**
   Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems
   a. A weaker spec may be easier to check: it is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce.
   b. A stronger spec may be easier to check: it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.
5) **Visibility: Judging status**
   a. The ability to measure progress or status against goals
      i. 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?"
   b. Involves setting goals that can be assessed at each stage of development
      i. The biggest challenge is early assessment, e.g., assessing specifications and design with respect to product quality

   Related to observability
   c. Example: Choosing a simple or standard internal data format to facilitate unit testing.
6) **Feedback: tuning the development process**
   Learning from experience: Each project provides information to improve the next
   Examples
   a. Checklists are built on the basis of errors revealed in the past
   b. Error taxonomies can help in building better test selection criteria
   c. Design guidelines can avoid common pitfalls

8. **What is scaffolding? Describe generic & application specific scaffolding.**

Scaffolding
Code produced to support development activities (especially testing), Not part of the "product" as seen by the end user May be temporary (like scaffolding in construction of buildings Includes Test harnesses, drivers, and stubs.

• Test driver

– A “main” program for running a test

• May be produced before a “real” main program

• Provides more control than the “real” main program

– To driver program under test through test cases

• Test stubs

– Substitute for called functions/methods/objects

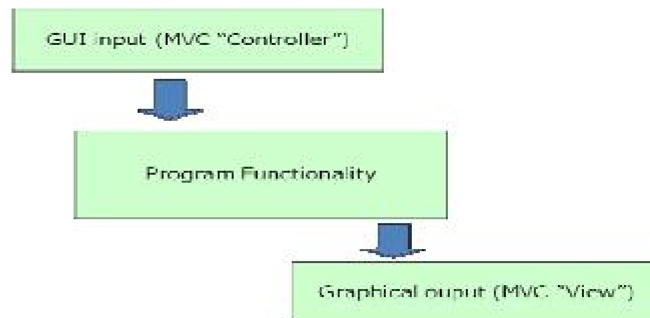
• Test harness

– Substitutes for other parts of the deployed environment

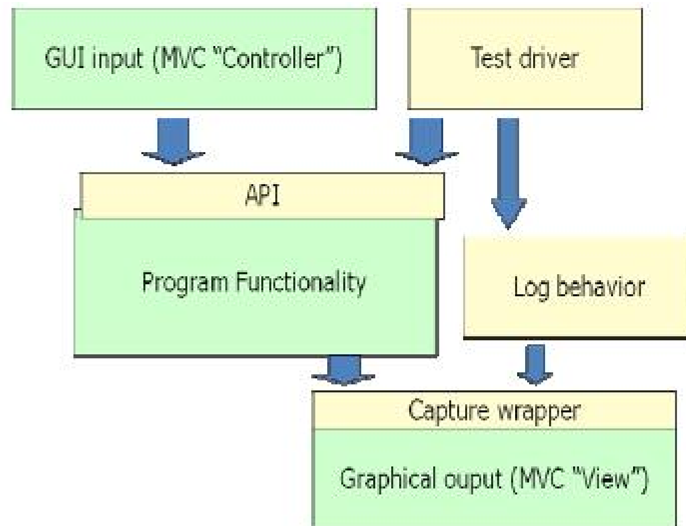
• Ex: Software simulation of a hardware device

The purpose of scaffolding is to provide controllability to execute test cases and observability to judge the outcome of test execution. Sometimes scaffolding is required to simply make a module executable.

Example: We want automated tests, but interactive input provides limited control and graphical output provides limited observability



A design for automated test includes provides interfaces for control (API) and observation (wrapper on ouput).



It should be generic or specific?

- How general should scaffolding be? To answer
  - We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit)
  - ... or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
  - ... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
- A question of costs and re-use
  - Just as for other kinds of software

b) **Define All C-uses/Some P-uses, All P-uses/Some C-uses, All Defs**

The set T satisfies the ***All P-Uses/Some C-Uses*** criterion for the program P iff for every variable v Î V, T contains definition clear (sub) paths from every defining node of v to every ***predicate*** use of v ***and if a definition of v has no P-uses, there is a definition-clear path to at least one computation use.***

– The set T satisfies the ***All C-Uses/Some P-Uses*** criterion for the program P iff for every variable v Î V, T contains definition clear (sub) paths from every defining node of v to every ***computation*** use of v ***and if a definition of v has no C-uses, there is a definition-clear path to at least one predicate use***

The set T satisfies the ***All-Defs*** criterion for the program P iff for every variable v Î V, T contains definition clear (sub) paths from every defining node of v ***to a use of v***