UNIT-4

LOGIC BASED TESTING:

This unit gives an indepth overview of logic based testing and its implementation.

At the end of this unit, the student will be able to:

- Understand the concept of Logic basedtesting.
- Learn about Decision Tables and their application
- Understand the use of decision tables in test-case design and know theirlimitations.
- Understand and interpret KV Charts and know theirlimitations.
- Learn how to transform specifications into sentences and map them into KVcharts.
- Understand the importance of dont-careconditions.

OVERVIEW OF LOGIC BASED TESTING :

- **INTRODUCTION:**
  - The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.
  - Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using**Karnaugh-Veitchcharts**.
  - "Logic" is one of the most often used words in programmers' vocabularies but one of their least usedtechniques.
  - Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate thosetechniques.
  - Logic has been, for several decades, the primary tool of hardware logic designers.
  - Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testingmethods.
  - As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest ofall.
  - The trouble with specifications is that they're hard toexpress.
  - Boolean algebra (also known as the sentential calculus) is the most basic of all logicsystems.
  - Higher-order logic systems are needed and used for formalspecifications.
  - Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on booleanalgebra.
  - **KNOWLEDGE BASEDSYSTEM:**
    - The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.
    - Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in thatdomain.
    - One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on thatdata.

- The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inferenceengine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formallogic.

- Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of theseprocessors.
- Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitchdiagram is that conceptualtool.

DECISION TABLES:

- Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the actionentry.
- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will takeplace.
- The condition stub is a list of names ofconditions.

CONDITION ENTRY

		RULE 1	RULE 2	RULE 3	RULE 4
CONDITION STUB	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
	CONDITION 4	NO	YES	NO	YES
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES

ACTION ENTRY

Figure 6.1 : Examples of Decision Table.

- A more general decision table can be asbelow:

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to thatrule.
- The action stub names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES", the action will take place; if "NO", the action will not takeplace.
- The table in Figure 6.1 can be translated as follows:

  Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule2).
- "Condition" is another word forpredicate.
- Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.
- Now the above translationsbecome:
  1. Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule2).
  2. Action 2 will be taken if the predicates are all false, (rule3).
  3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule4).

- In addition to the stated rules, we also need a **Default Rule** that specifies the default action to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure6.3

	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

- **DECISION-TABLEPROCESSORS:**
  - Decision tables can be automatically translated into code and, as such, are a higher-order language
  - If the rule is satisfied, the corresponding action takesplace
  - Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action istaken
  - Decision tables have become a useful tool in the programmers kit, in business dataprocessing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

0. The specification is given as a decision table or can be easily converted into one.
1. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takesplace.
2. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takesplace.
3. Once a rule is satisfied and an action selected, no other rule need be examined.
4. If several actions can result from satisfying a rule, the order in which the actions are executed doesn'tmatter

DECISION-TABLES AND STRUCTURE:

- Decision tables can also be used to examine a program'sstructure.
- Figure 6.4 shows a program segment that consists of a decisiontree.
- These decisions, in various combinations, can lead to actions 1, 2, or 3.

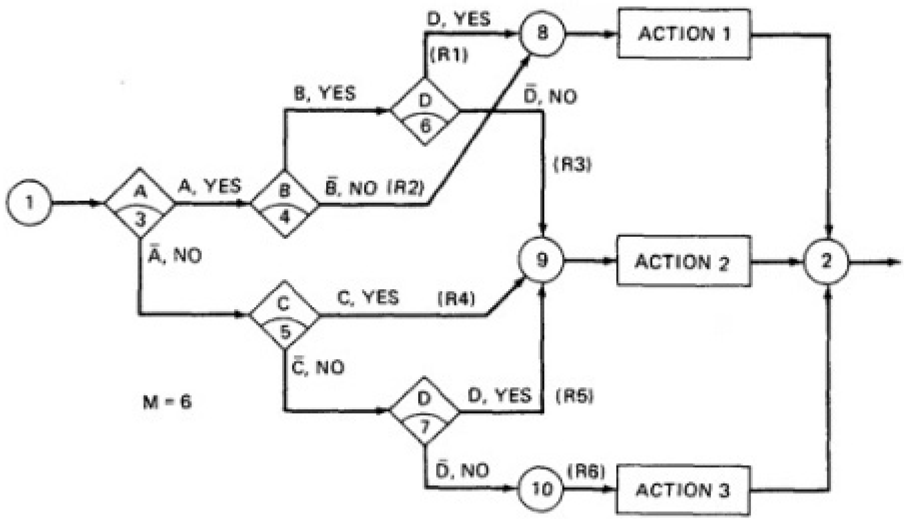


Figure 6.4 : A Sample Program

- If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I,YES.
- The corresponding decision table is shown in Table6.1

	RULE 1	**RULE 2**	**RULE 3**	**RULE 4**	**RULE 5**	**RULE 6**
CONDITION A	YES	YES	YES	NO	NO	NO
CONDITION B	YES	NO	YES	I	I	I
CONDITION C	I	I	I	YES	NO	NO
CONDITION D	YES	I	NO	I	YES	NO
ACTION 1	YES	YES	NO	NO	NO	NO
ACTION 2	NO	NO	YES	YES	YES	NO
ACTION 3	NO	NO	NO	NO	NO	YES

- ***Table 6.1 : Decision Table corresponding toFigure 6.4***
- As an example, expanding the immaterial cases results asbelow:

	RULE 1	RULE 2
CONDITION 1	YES	YES
CONDITION 2	I	NO
CONDITION 3	YES	I
CONDITION 4	NO	NO
ACTION 1	YES	NO
ACTION 2	NO	YES

⇨

RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
YES	YES	YES	YES
YES	NO	NO	NO
YES	YES	YES	NO
NO	NO	NO	NO
YES	YES	NO	NO
NO	NO	YES	YES

- Similalrly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 asbelow:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

- ***Table 6.2 : Expansion of Table6.1***

- Sixteen cases are represented in Table 6.1, and no case appearstwice.
- Consequently, the flowgraph appears to be complete andconsistent.
- As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

- Consider the following specification whose putative flowgraph is shown in Figure 6.5:
    1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions aremet.
    2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions aremet.
    3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions aremet.
    4. If none ofthe conditions is met, then do processes A1, A2, andA3.
    5. When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and(A1,A2,A3).
- Figure 6.5 shows a sample program with abug.

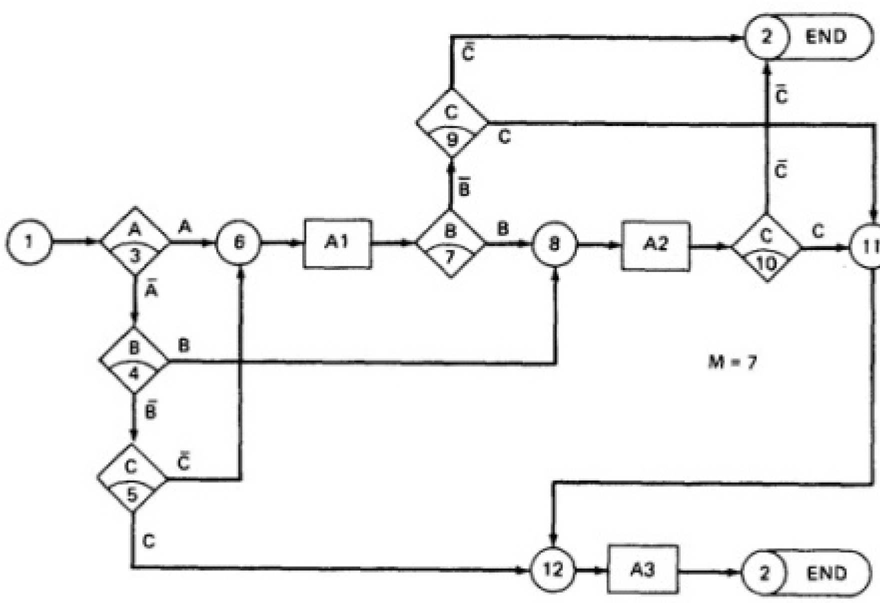


Figure 6.5 : A Troublesome Program

- The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$ cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 andA3.
- Table 6.3 shows the conversion of this flowgraph into a decision table after expansion.

RULES

	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$\bar{A}B\bar{C}$	$AB\bar{C}$	ABC	$A\bar{B}C$	$A\bar{B}\bar{C}$
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3 : Decision Table for Figure 6.5

PATH EXPRESSIONS:

- **GENERAL:**
    - Logic-based testing is structural testing when it's applied to structure (e.g., control flowgraph of an implementation); it's functional testing when it's applied to aspecification.
    - In logic-based testing we focus on the truth values of control flowpredicates.
    - A **predicate** is implemented as a process whose outcome is a truth-functional value.
    - For our purpose, logic-based testing is restricted to binarypredicates.
    - We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\bar{A}$) asweights.

BOOLEAN ALGEBRA:

- **STEPS:**
    1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say $\bar{A}$).
    2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\bar{C}$.
    3. Iftwo or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

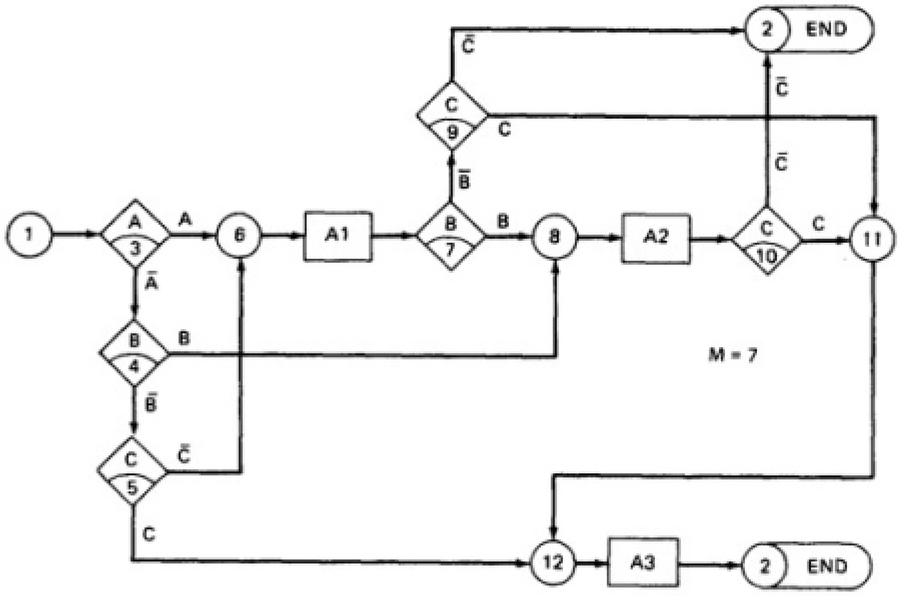


Figure 6.5 : A Troublesome Program

- Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify thepoint.

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\,\overline{C} \\
N8 &= (N6)B + \overline{A}B = AB + \overline{A}\,\overline{B}\,\overline{C}B + \overline{A}B \\
N11 &= (N8)C + (N6)\overline{B}C \\
N12 &= N11 + \overline{A}\,\overline{B}C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C}
\end{aligned}
$$

- There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "alwaysfalse".

- **RULES OF BOOLEANALGEBRA:**
  - Boolean algebra has three operators: X (AND), +(OR)and $\overline{A}$(NOT)
  - **X :**meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinaryalgebra.
  - **+ :**meaning OR. "A + B" means "either A is true or B is true or both".
  - $\overline{A}$meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.
  - The following are the laws of booleanalgebra:

1. $A + A$ $\overline{A} + \overline{A}$	$= A$ $= \overline{A}$	If something is true, saying it twice doesn't make it truer, ditto for falsehoods.
2. $A + 1$	$= 1$	If something is always true, then "either A or true or both" must also be universally true.
3. $A + 0$	$= A$	
4. $A + B$	$= B + A$	Commutative law.
5. $A + \overline{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6. AA $\overline{A}\overline{A}$	$= A$ $= \overline{A}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\overline{A}$	$= 0$	A statement can't be simultaneously true and false.
11. $\overline{\overline{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12. $\overline{0}$	$= 1$	
13. $\overline{1}$	$= 0$	
14. $\overline{A + B}$	$= \overline{A}\,\overline{B}$	Called "De Morgan's theorem or law."
15. $\overline{AB}$	$= \overline{A} + \overline{B}$	
16. $A(B + C)$	$= AB + AC$	Distributive law.
17. $(AB)C$	$= A(BC)$	Multiplication is associative.
18. $(A + B) + C$	$= A + (B + C)$	So is addition.
19. $A + \overline{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebraexpression.

Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B)

The product of several literals is called a **product term** (e.g., ABC, DE).

An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.

The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB + DEF reduces by rule 20 to AB + DEF; that is, AB and DEF are prime implicants.

The path expressions of Figure 6.5 can now be simplified by applying the rules.

The following are the laws of boolean algebra:

N6	$= A + \overline{A}\,\overline{B}\,\overline{C}$	
	$= A + \overline{B}\,\overline{C}$	: Use rule 19, with "B" = $\overline{B}\,\overline{C}$.
N8	$= (N6)B + \overline{A}B$	
	$= (A + \overline{B}\,\overline{C})B + \overline{A}B$	: Substitution.
	$= AB + \overline{B}\,\overline{C}B + \overline{A}\,\overline{B}$	: Rule 16 (distributive law).
	$= AB + B\overline{B}C + \overline{A}B$	: Rule 9 (commutative multiplication).
	$= AB + 0C + \overline{A}B$	: Rule 10.
	$= AB + 0 + \overline{A}B$	: Rule 8.
	$= AB + \overline{A}B$	: Rule 3.
	$= (A + \overline{A})B$	: Rule 16 (distributive law).
	$= 1 \times B$	: Rule 5.
	$= B$	: Rules 7, 9.

Similarly,

$$
\begin{aligned}
N11 &= (N8)C + (N6)\overline{B}C \\
&= BC + (A + \overline{B}\,\overline{C})\overline{B}C && \text{: Substitution.} \\
&= BC + A\overline{B}C && \text{: Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B}A) && \text{: Rules 9, 16.} \\
&= C(B + A) && \text{: Rule 19.} \\
&= AC + BC && \text{: Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A}\,\overline{B}C \\
&= AC + BC + \overline{A}\,\overline{B}C \\
&= C(B + \overline{A}\,\overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C} \\
&= C + B\overline{C} + (A + \overline{B}\,\overline{C})\overline{B}\,\overline{C} \\
&= C + B\overline{C} + \overline{B}\,\overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}
$$

The deviation from the specification is now clear. The functions should have been:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\,\overline{C} = A + \overline{B}\,\overline{C} && \text{: correct.} \\
N8 &= B + \overline{A}\,\overline{B}\,\overline{C} = B + \overline{A}\,\overline{C} && \text{: wrong, was just B.} \\
N12 &= C + \overline{A}\,\overline{B}\,\overline{C} = C + \overline{A}\,\overline{B} && \text{: wrong, was just C.}
\end{aligned}
$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate-value combinations lead to where.

KV CHARTS:

- **INTRODUCTION:**
  - If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you'retesting.
  - **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
  - Beyond six variables these diagrams get cumbersome and may not be effective.
- **SINGLEVARIABLE:**
  - Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KVchart.

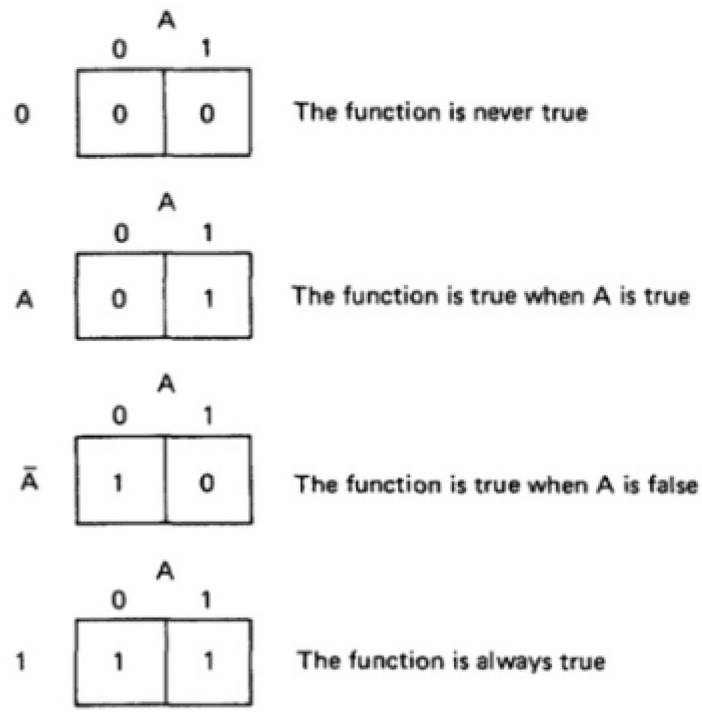


Figure 6.6 : KV Charts for Functions of a Single Variable.

- The charts show all possible truth values that the variable A canhave.
- A "1" means the variable’s value is "1" or TRUE. A "0" means that the variable's value is 0 orFALSE.
- The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of thevariable.
- We usually do not explicitly put in 0 entries but specify only the conditions under which the function istrue.

- **TWOVARIABLES:**
  - Figure 6.7 shows eight of the sixteen possible functions of twovariables.

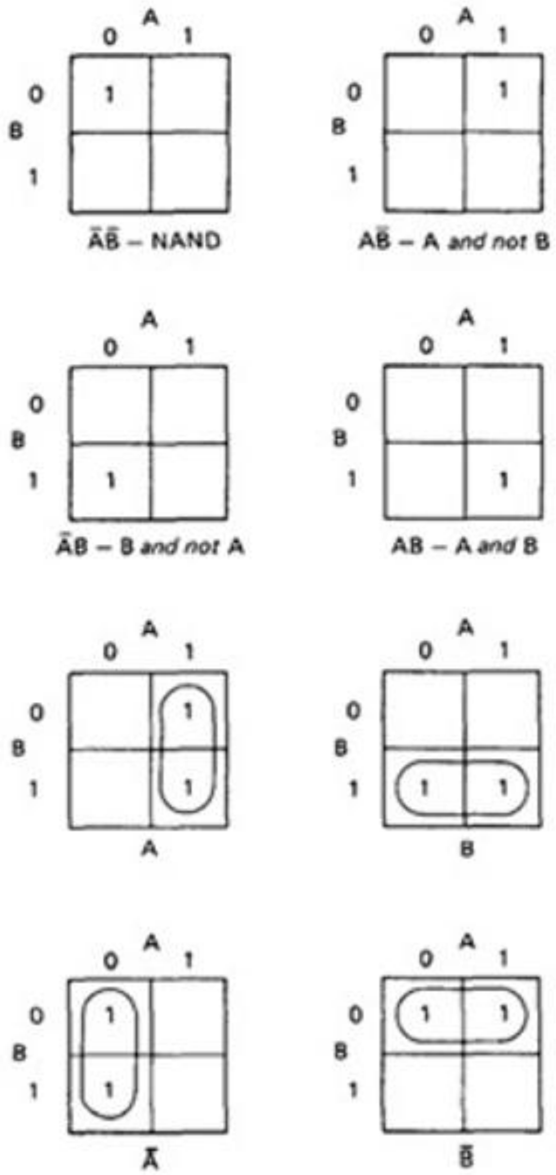


Figure 6.7 : KV Charts for Functions of Two Variables.

- Each box corresponds to the combination of values of the variables for the row and column of thatbox.
- A pair may be adjacent either horizontally or vertically but notdiagonally.
- Any variable that changes in either the horizontal or vertical direction does not appear in the expression.
- In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.
- Figure 6.8 shows the remaining eight functions of twovariables.

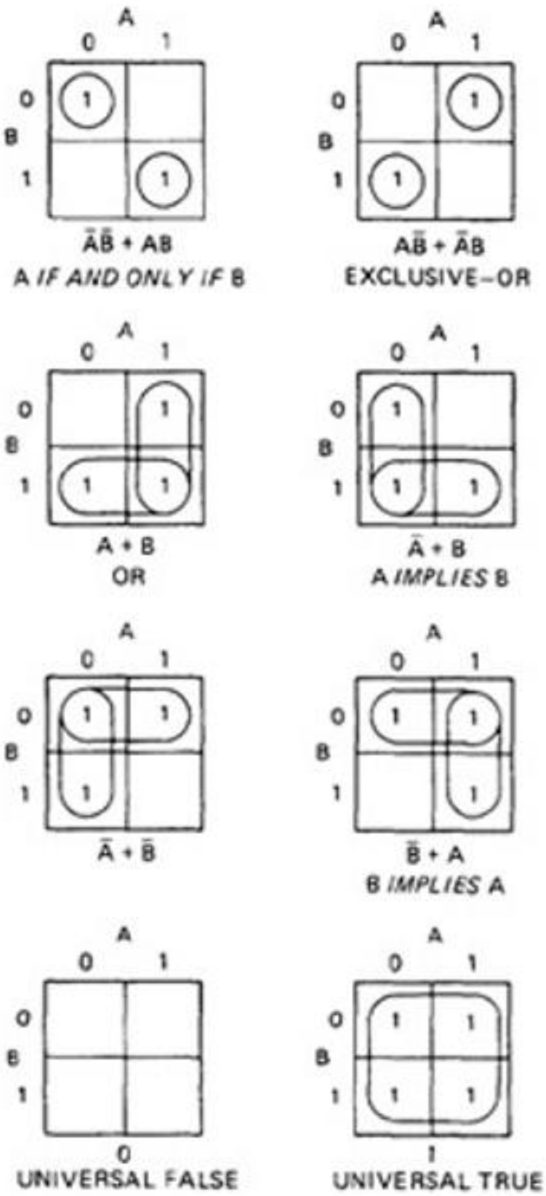
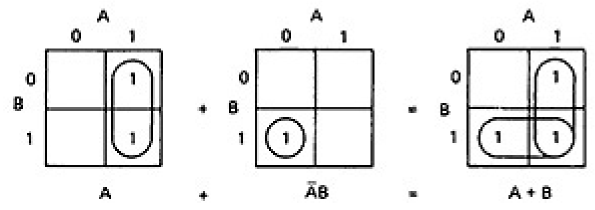


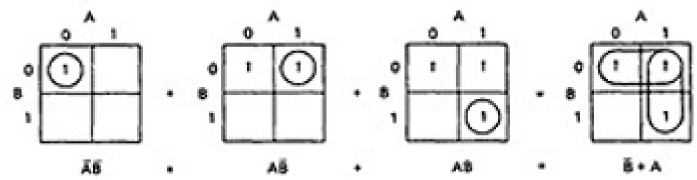
Figure 6.8 : More Functions of Two Variables.

- The first chart has two 1's in it, but because they are not adjacent, each must be takenseparately.
- They are written using a plussign.
- It is clear now why there are sixteen functions of twovariables.
- Each box in the KV chart corresponds to a combination of the variables' values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0entry).
- Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2n}ways of doing this.

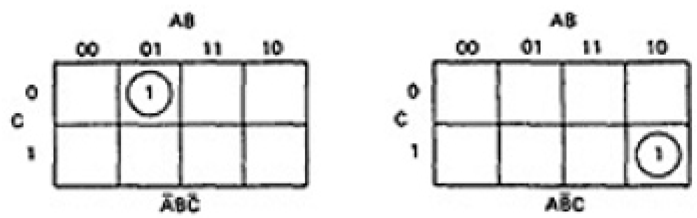
- Consequently for one variable there are 2^{21} = 4 functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, and so on.
- Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in thechart.

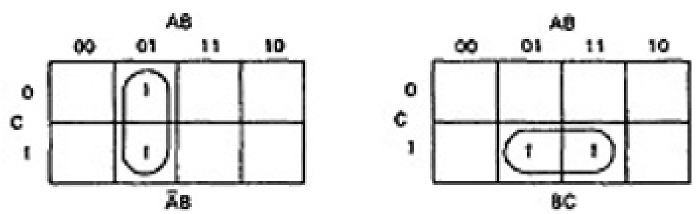


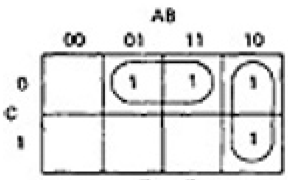
OR



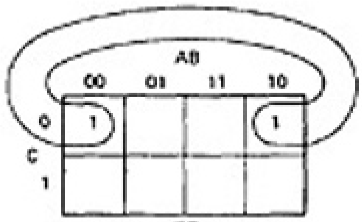
- **THREE VARIABLES:**
  - KV charts for three variables are shownbelow.
  - As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or1.
  - A three-variable chart can have groupings of 1, 2, 4, and 8boxes.
  - A few examples will illustrate theprinciples:



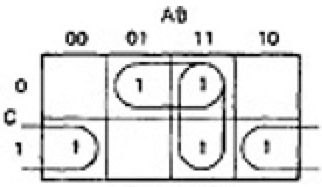




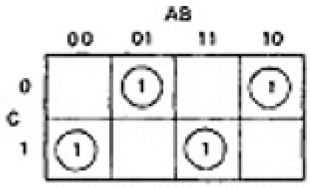
$B\bar{C} + A\bar{B}$



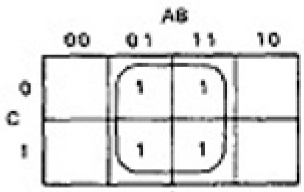
$\bar{B}\bar{C}$



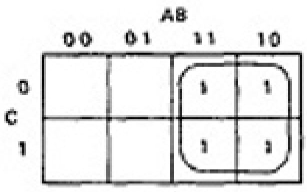
$B\bar{C} + AB + \bar{B}C$



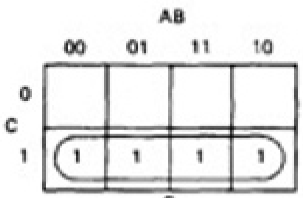
$\bar{A}\bar{B}C + \bar{A}B\bar{C} + ABC + A\bar{B}\bar{C}$



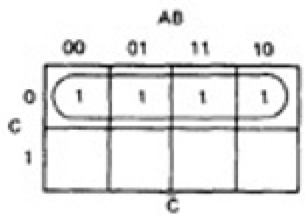
B



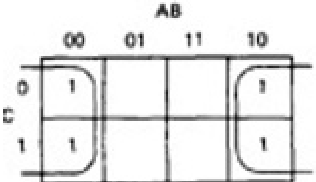
A



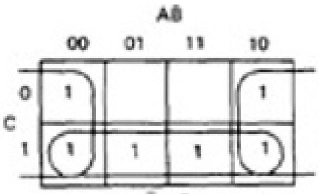
C



$\bar{C}$



$\bar{B}$



$\bar{B} + C$

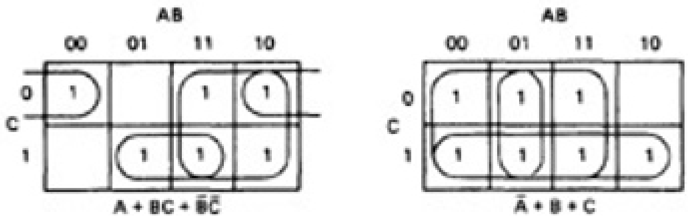


Figure 6.8 : KV Charts for Functions of Three Variables.

- You'll notice that there are several ways to circle the boxes into maximum- sized coveringgroups.

STATES, STATE GRAPHS, AND TRANSITION TESTING

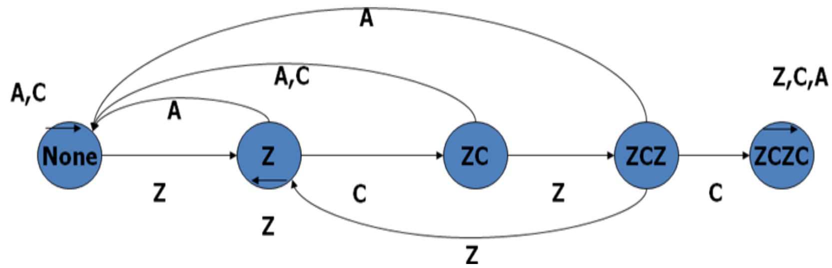
Introduction

- The finite state machine is as fundamental to software engineering as boolean algebra to logic.
- State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of softwarebehavior.
- Finite state machines can also be implemented as table-driven software, in which case they are a powerful designoption.

State Graphs

- A state is defined as: “A combination of circumstances or attributes belonging for the time being to a person orthing.”
- For example, a moving automobile whose engine is running can have the following states with respect to its transmission.
  - Reverse gear
  - Neutralgear
  - Firstgear
  - Secondgear
  - Thirdgear
  - Fourth gear

  State graph -
  Example
- For example, a program that detects the character sequence “ZCZC” can be in the followingstates.
- Neither ZCZC nor any part of it has beendetected.
  - Z has beendetected.
  - ZC has been detected.
  - ZCZ has been detected.
  - ZCZC has beendetected.



States are represented by Nodes. State are numbered or may identified by words or whatever else isconvenient.

Inputs and Transitions

- Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made aTransition.
- Transitions are denoted by links that join thestates.
- The input that causes the transition are marked on the link; that is, the inputs are link weights.
- There is one out link from every state for everyinput.

- If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: “input1, input2,input3………”.

Finite State Machine

- A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions betweenstates.
    - Outputs
- An output can be associated with anylink.
- Out puts are denoted by letters or words and are separated from inputs by a slash as follows:“input/output”.
- As always, output denotes anything of interest that’s observable and is not restricted to explicit outputs bydevices.
- Outputs are also linkweights.
- If every input associated with a transition causes the same output, then denoted itas:
    - “input1, input2,input3… ..../output”

State Tables

- Big state graphs are cluttered and hard tofollow.
- It’s more convenient to represent the state graph as a table (the state table or state transition table) that specifies the states, the inputs, the transitions and the outputs.
- The following conventions areused:
- Each row of the table corresponds to astate.
- Each column corresponds to an inputcondition.
- The box at the intersection of a row and a column specifies the next state (the transition) and the output, ifany.

State Table-Example

	inputs		
STATE	Z	C	A
NONE	Z	NONE	NONE
Z	Z	ZC	NONE
ZC	ZCZ	NONE	NONE
ZCZ	Z	ZCZC	NONE
ZCZC	ZCZC	ZCZC	ZCZC

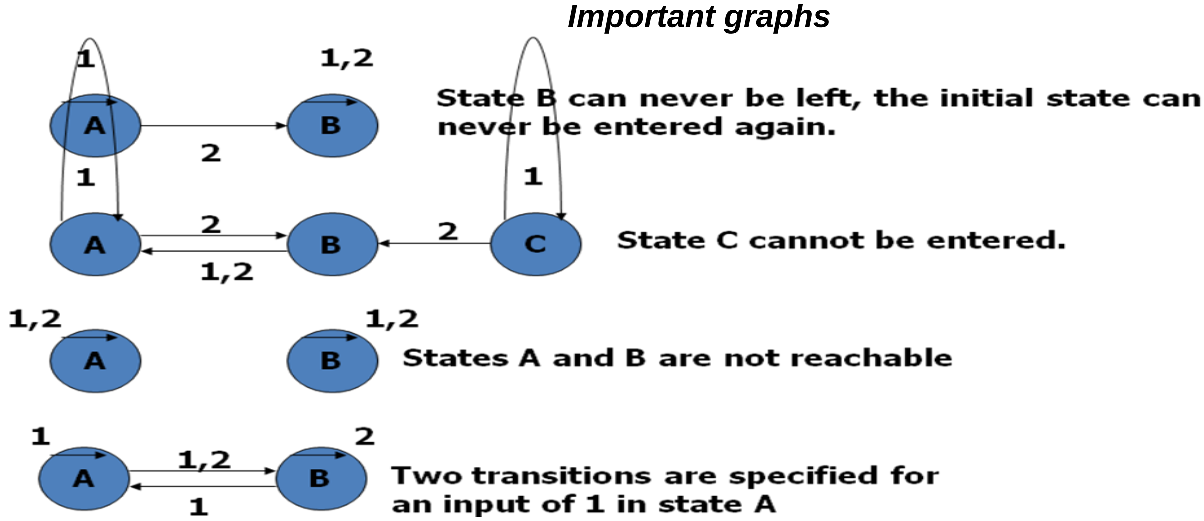
Time Versus Sequence

- State graphs don’t represent time-they represent sequence.
- A transition might take microseconds orcenturies;
- A system could be in one state for milliseconds and another for years- the state graph would be the same because it has no notion oftime.
- Although the finite state machines model can be elaborated to include notions of time in addition to sequence, such as time PetriNets.
    - Software implementation
- There is rarely a direct correspondence between programs and the behavior of a process described as a stategraph.
- The state graph represents, the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and soon.
- There is no simple correspondence between lines of code and states. The state table forms thebasis.

Good State Graphs and Bad

- What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test designcontext.
- Here are some principles forjudging.
    - The total number of states is equal to the product of the possibilities of factors that make up thestate.
    - For every state and input there is exactly one transition specified to exactly one, possibly the same,state.
    - For every transition there is one output action specified. The output could be trivial, but at least one output does somethingsensible.
    - For every state there is a sequence of inputs that will drive the system back to the same state.

Important graphs



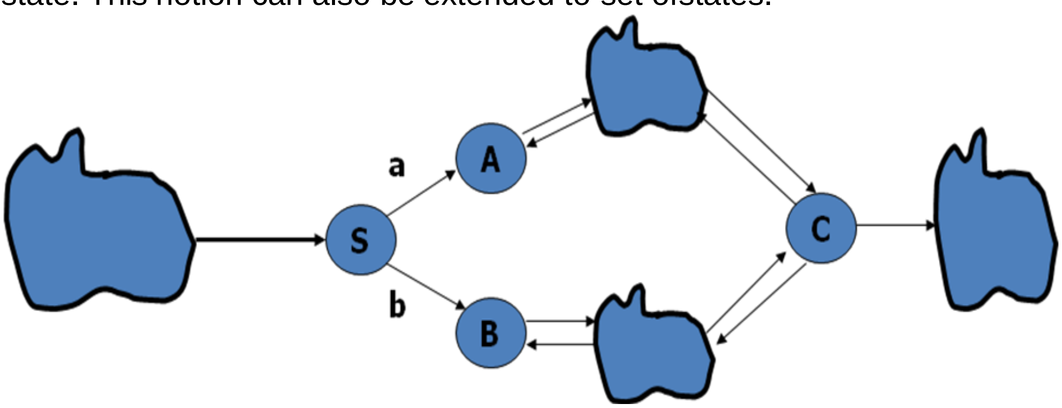
State Bugs-Number of States

- The number of states in a state graph is the number of states we choose to recognize or model.

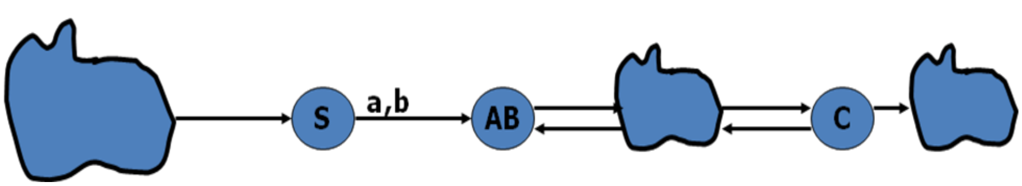
- The state is directly or indirectly recorded as a combination of values of variables that appear in the database.
- For example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of 2*2*10=40states.
- The number of states can be computed asfollows:
  - Identify all the component factors of thestate.
  - Identify all the allowable values for eachfactor.
  - The number of states is the product of the number of allowable values of all the factors.
- Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer thinks thereare.
- There is no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states thereare.
  - Impossible States
- Some times some combinations of factors may appear to be impossible.
- The discrepancy between the programmer's state count and the tester's state count isoften due to a difference of opinion concerning "impossiblestates".
- A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical condition handler when they appear to haveoccurred.

Equivalent States

- Two states are Equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to set ofstates.



Merging of Equivalent States



Recognizing Equivalent States

- Equivalent states can be recognized by the followingprocedures:
- The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state coulddiffer.
- There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can bemerged.

TransitionBugs-

unspecified and contradictory Transitions

- Every input-state combination must have a specified transition.
- If the transition is impossible, then there must be a mechanism that prevents the input from occurring in thatstate.
- Exactly one transition must be specified for every combination of input andstate.
- A program can't have contradictions orambiguities.
- Ambiguities are impossible because the program will do something for every input. Even the state does not change, by definition this is a transition to the same state.

Unreachable States

- An unreachable state is like unreachablecode.
- A state that no input sequence canreach.
- An unreachable state is not impossible, just as unreachable code is notimpossible
- There may be transitions from unreachable state to other states; there usually because the state became unreachable as a result of incorrecttransition.
- There are two possibilities for unreachablestates:
    - There is a bug; that is some transitions aremissing.
    - The transitions are there, but you don't know aboutit.

Dead States

- A dead state is a state that once entered cannot beleft.
- This is not necessarily a bug but it issuspicious.

Output Errors

- The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could beincorrect.
- Output actions must be verified independently of states and transitions. State Testing

  Impact of Bugs
- If a routine is specified as a state graph that has been verified as correct in all details. Program code or table or a combination of both must still beimplemented.
- A bug can manifest itself as one of the followingsymptoms:
- Wrong number ofstates.
- Wrong transitions for a given state-inputcombination.
- Wrong output for a giventransition.
- Pairs of states or sets of states that are inadvertently madeequivalent.
- States or set of states that are split to create in equivalentduplicates.

- States or sets of states that have becomedead.
- States or sets of states that have becomeunreachable.

Principles of State Testing

- The strategy for state testing is analogous to that used for path testing flowgraphs.
- Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a state graph.
- The notion of coverage is identical to that used for flowgraphs.
- Even though more state testing is done as a single case in a grand tour, it's impractical to do it that way for severalreasons.
- In the early phases of testing, you will never complete the grand tour because ofbugs.
- Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be tested. A grand tour is waste oftime.
- Theirs is no much history in a long test sequence and so much has happened that verification isdifficult.

Starting point of state testing

- Define a set of covering input sequences that get back to the initial state when starting from the initialstate.
- For each step in each input sequence, define the expected next state, the expected transition, and the expected outputcode.
- A set of tests, then, consists of three sets ofsequences:
    - Inputsequences
    - Corresponding transitions or next-statenames
    - Outputsequences

Limitations and Extensions

- State transition coverage in a state graph model does not guarantee completetesting.
- How defines a hierarchy of paths and methods for combining paths to produce covers of state graphs.
- The simplest is called a "0 switch" which corresponds to testing each transition individually.
- The next level consists of testing transitions sequences consisting of two transitions called "1switches".
- The maximum length switch is "n-1 switch" where there are n numbers ofstates.
    - Situations at which state testing isuseful
- Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs isimportant.
- Most protocols between systems, between humans and machines, between components of a system.
- Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on thestate.

Whenever a feature is directly and explicitly implemented as one or more state transition tables.