

UNIT-4

LOGIC BASED TESTING:

OVERVIEW OF LOGIC BASED TESTING :

- "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
- 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 testing methods.
- 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 of all.
- Higher-order logic systems are needed and used for formal specifications.
- 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 boolean algebra.

KNOWLEDGE BASED SYSTEM:

- 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 that domain.
- 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 that data.
- 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 **inference engine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.
- 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 these processors.

4.1 DECISION TABLES:

4.1.1 Definitions and notations:

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

		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

Figure 1 : Examples of Decision Table.

The table in Figure1 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 (rule 2).
- "Condition" is another word for predicate.
- Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.
- Now the above translations become:
 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 (rule 2).
 2. Action 2 will be taken if the predicates are all false, (rule 3).

3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).
- 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 1 is shown in Figure 2

	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 2 : The default rules of Table in Figure 1

4.1.2 DECISION-TABLE PROCESSORS:

- Decision tables can be automatically translated into code and, as such, are a higher-order language
- If the rule is satisfied, the corresponding action takes place
- 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 is taken
- Decision tables have become a useful tool in the programmers kit, in business data processing.

4.1.3 DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

1. The specification is given as a decision table or can be easily converted into one.
2. 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 takes place.
3. 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 takes place.
4. Once a rule is satisfied and an action selected, no other rule need be examined.

5. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter

4.1.4 Expansion of Immaterial Cases

- * **Immaterial entries (I)** cause most decision-table contradictions.
- * If a condition's truth value is immaterial in a rule, satisfying the rule does not depend on the condition. It doesn't mean that the case is impossible. For example,

For example

Rule 1: "if the persons are male and over 30, then they shall receive a 15% raise"


Rule 2: "but if the persons are female, then they shall receive a 10% raise"

	RULE 2		RULE 4			
	RULE 2.1	RULE 2.2	RULE 4.1	RULE 4.2	RULE 4.3	RULE 4.4
CONDITION 1	YES	YES	NO	NO	NO	NO
CONDITION 2	YES	NO	YES	YES	NO	NO
CONDITION 3	YES	YES	YES	NO	NO	YES
CONDITION 4	YES	YES	YES	YES	YES	YES

Table 3: Expansion of Immaterial cases for Rule 2 and 4 for figure 1

- Table 10.4 is an example of an inconsistent specification in which the expansion of two rules yields a contradiction.
- Rule 2 has been expanded by converting the I entry for condition 2 into a separate rule 2.1 for YES and Rule 2.2 for NO and similarly for Rule 4
- The key to test case design based on decision tables is to expand immaterial entries to generate tests that correspond to all the subrules that result.
- If some conditions are three-way an immaterial expand to 3 subrules . Similarly an immaterial n-way case statement expands to n-sub rules.
- If no default rules are given then all the cases not covered by explicit rules
- The specification is complete if and only if n (binary) conditions expand in 2^n sub unique rules
- The expansion of an inconsistent specification is shown in below diagram

	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
YES	NO	NO	NO	NO
YES	YES	YES	YES	NO
NO	NO	NO	NO	NO
YES	YES	NO	NO	NO
NO	NO	YES	YES	YES

Table 10.4 The expansion of an inconsistent specification

4.1.5 Test Case Design

1. If there are k rules over n binary predicates, there are at least k cases to consider and at most 2^n cases.
2. It is not usually possible to change the order in which the *predicates* are evaluated because that order is built into the program,*A tolerant implementation would allow these fields in any order.
3. It is not usually possible to change the order in which the *rules* are evaluated because that order is built into the program, but if the implementation allows the rule evaluation order to be modified, test different orders for the rules by pairwise interchanges.
4. Identify the places in the routine where rules are invoked or where the processors that evaluate the rules are called. Identify the places where actions are initiated.

4.1.6 DECISION-TABLES AND STRUCTURE:

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

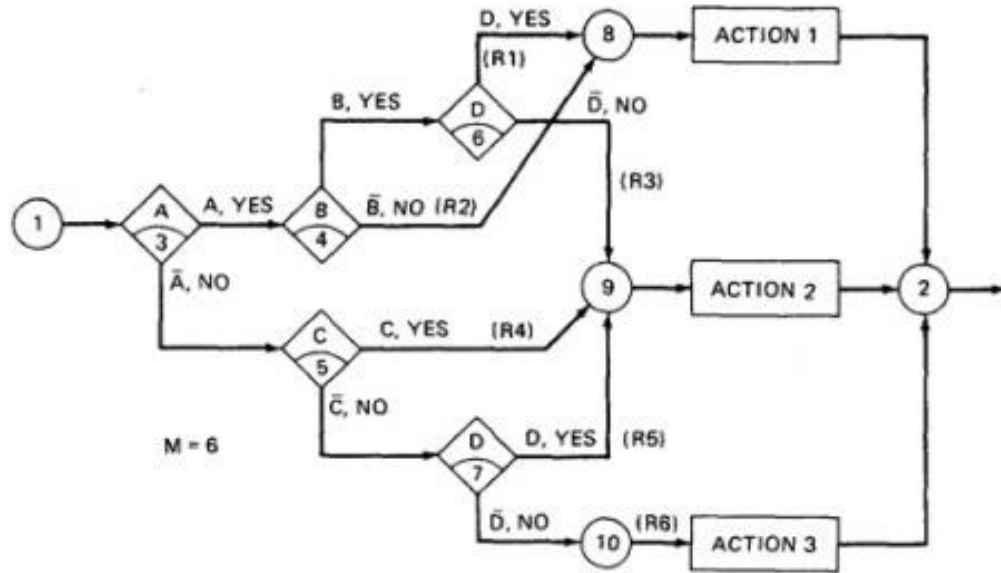


Figure 3 : A Sample Program

- 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 Table 6.1

	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 to Figure 6.4

Similarly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

	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 Table 6.1

- Sixteen cases are represented in Table 6.1, and no case appears twice.
- Consequently, the flowgraph appears to be complete and consistent.
- 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 Figure4:

1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.
2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.
3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
4. If none of the conditions is met, then do processes A1, A2, and A3.
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).

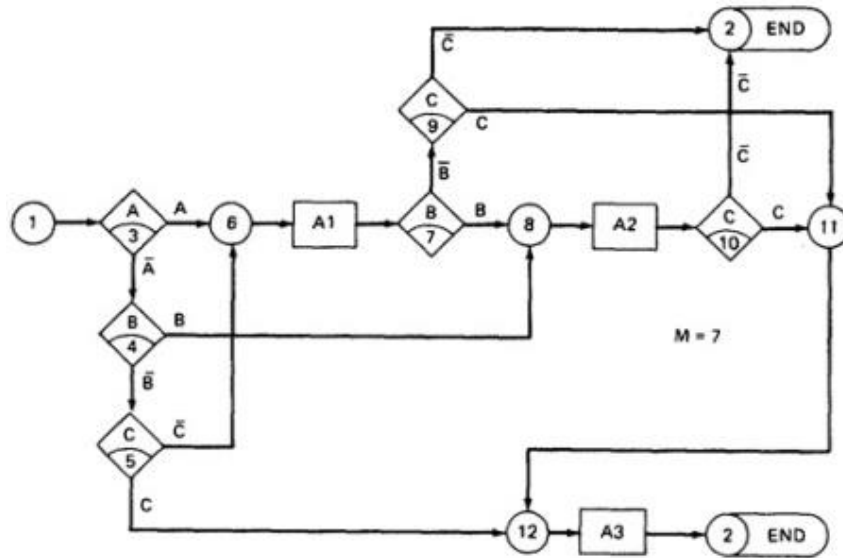


Figure 6.5 : A Troublesome Program

- The programmer tried to force all three processes to be executed for the cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.
- Table 6.3 shows the conversion of this flowgraph into a decision table after expansion.

RULES

	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
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

4.2 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 a specification.
- In logic-based testing we focus on the truth values of control flow predicates.
- A **predicate** is implemented as a process whose outcome is a truth-functional value.
- For our purpose, logic-based testing is restricted to binary predicates.
- 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} as weights).

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 $\bar{A}BC$.
3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

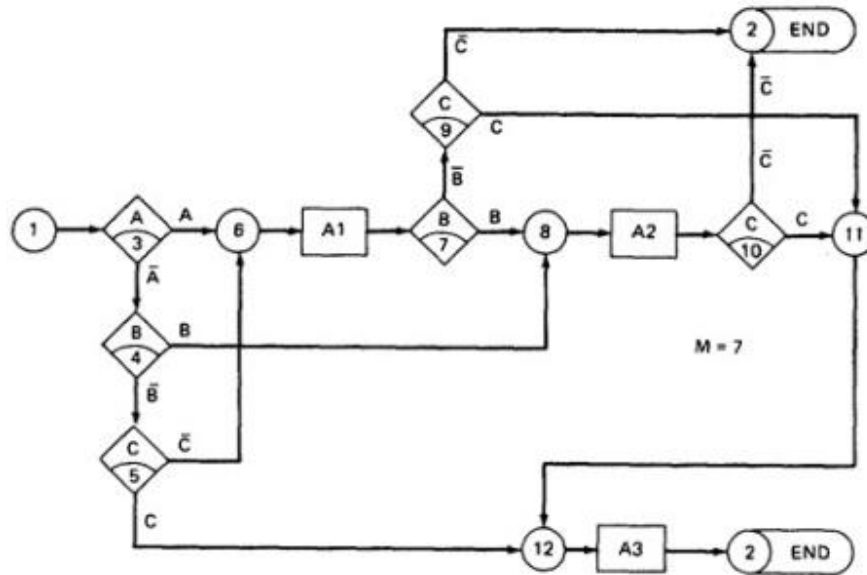


Figure 6.5 : A Troublesome Program

- 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 the point.

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

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

RULES OF BOOLEAN ALGEBRA:

- Boolean algebra has three operators: X (AND), + (OR) and $\overline{\quad}$ (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 ordinary algebra.
- +** : 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 boolean algebra:

1. $\frac{A + A}{\bar{A} + \bar{A}}$	$= \frac{A}{\bar{A}}$	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. $A + \bar{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6. $\frac{AA}{\bar{A}\bar{A}}$	$= \frac{A}{\bar{A}}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\bar{A}$	$= 0$	A statement can't be simultaneously true and false.
11. $\bar{\bar{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12. $\bar{0}$	$= 1$	
13. $\bar{1}$	$= 0$	
14. $\overline{A + B}$	$= \bar{A}\bar{B}$	Called "De Morgan's theorem or law."
15. \overline{AB}	$= \bar{A} + \bar{B}$	
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. $A + \bar{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

- In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.
- 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:

$$\begin{aligned}
N6 &= A + \overline{A}\overline{B}\overline{C} \\
&= A + \overline{B}\overline{C} && : \text{Use rule 19, with "B" = } \overline{B}\overline{C}. \\
N8 &= (N6)B + \overline{A}B \\
&= (A + \overline{B}\overline{C})B + \overline{A}B \\
&= AB + \overline{B}\overline{C}B + \overline{A}B \\
&= AB + \overline{B}\overline{C} + \overline{A}B \\
&= AB + 0C + \overline{A}B \\
&= AB + 0 + \overline{A}B \\
&= AB + \overline{A}B \\
&= (A + \overline{A})B \\
&= 1 \times B \\
&= B
\end{aligned}$$

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

Similarly,

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

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.

4.3 KV CHARTS:

INTRODUCTION:

- If you had to deal with expressions in four, five, or six variables, you could get stuck down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.
- **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
- Beyond six variables these diagrams get cumbersome and may not be effective.

SINGLE VARIABLE: The Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

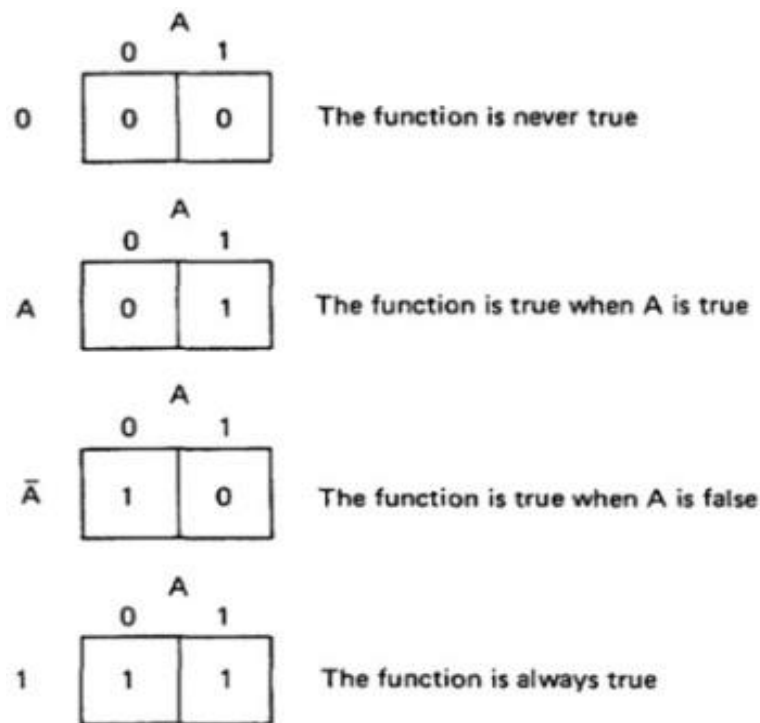


Figure 6.6 : KV Charts for Functions of a Single Variable.

The charts show all possible truth values that the variable A can have.

- A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.
- The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.

- We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.
- **TWO VARIABLES:**
 - Figure 6.7 shows eight of the sixteen possible functions of two variables.

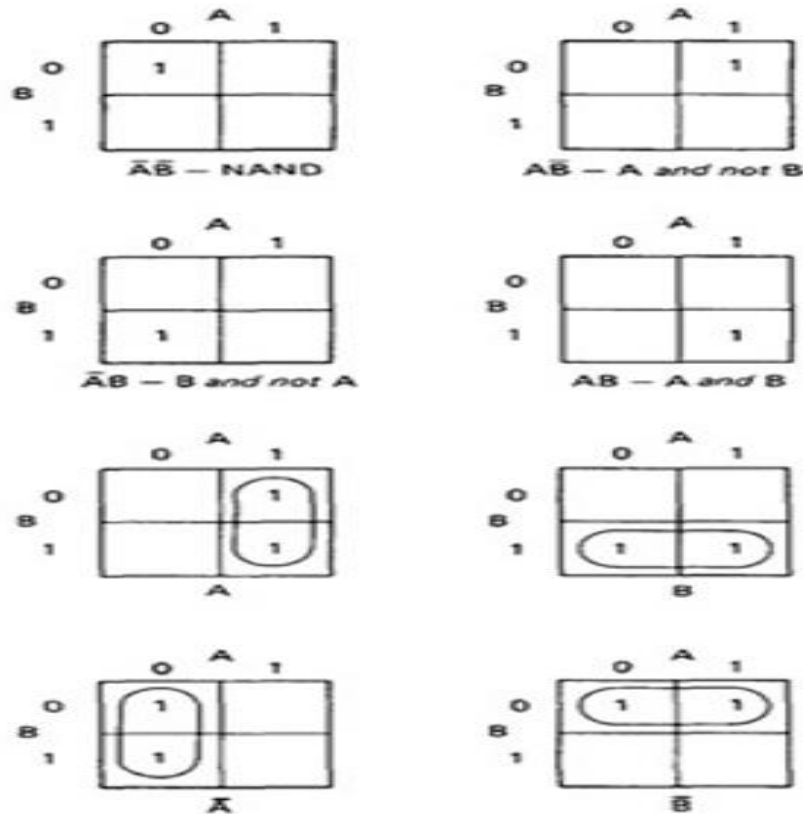


Figure 6.7 : KV Charts for Functions of Two Variables.

- Each box corresponds to the combination of values of the variables for the row and column of that box.
- A pair may be adjacent either horizontally or vertically but not diagonally.
- 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 two variables.

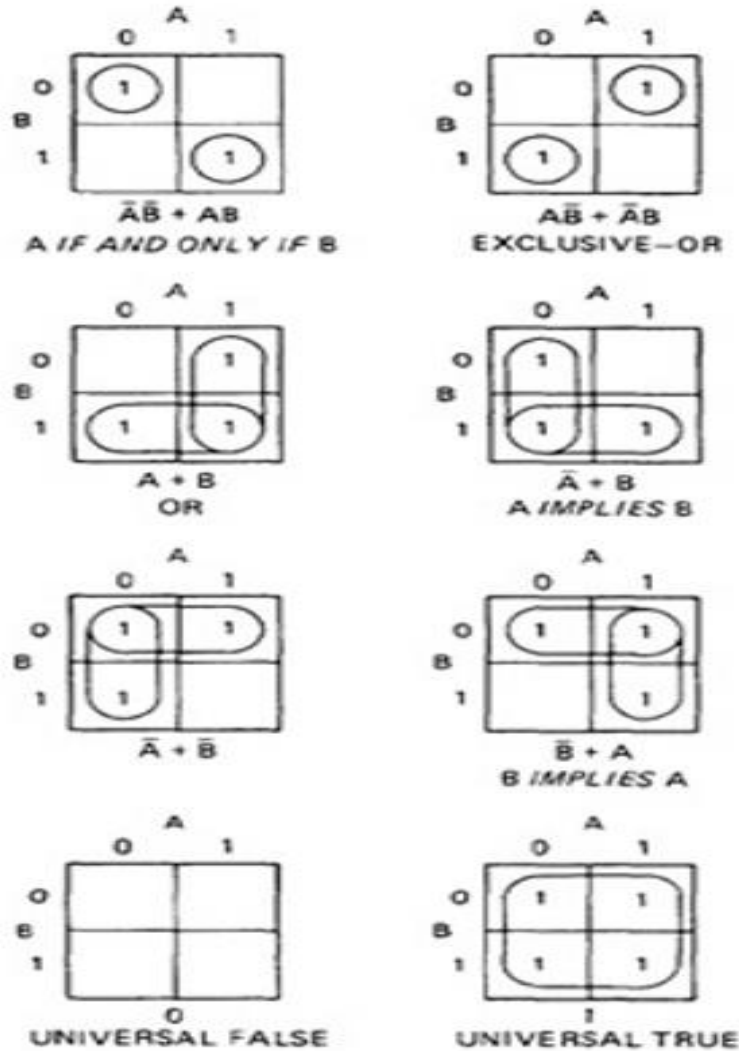
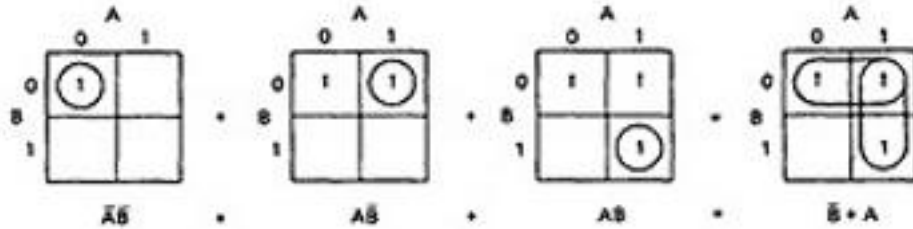
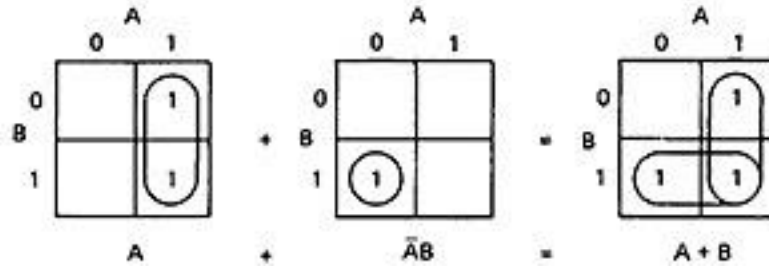


Figure 6.8 : More Functions of Two Variables.

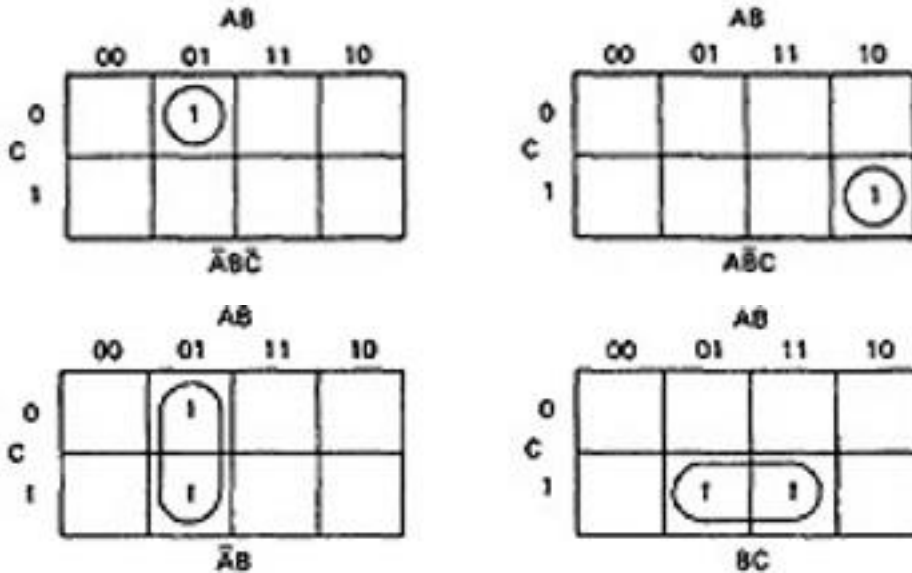
- The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.
- They are written using a plus sign.
- It is clear now why there are sixteen functions of two variables.
- Each box in the KV chart corresponds to a combination of the variables' values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).
- 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^{2^n} ways of doing this.
- Consequently for one variable there are $2^{2^1} = 4$ 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 the chart.

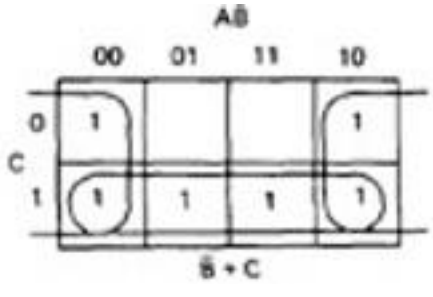
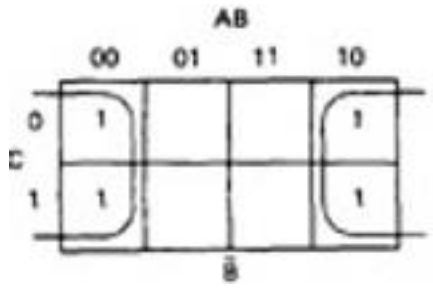
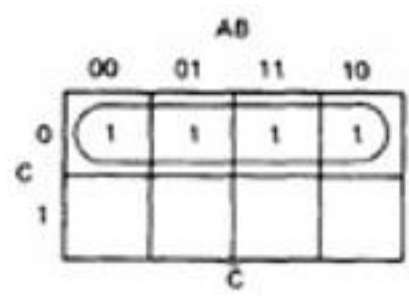
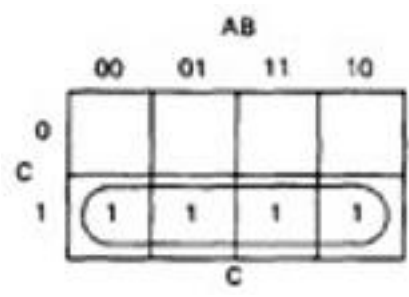
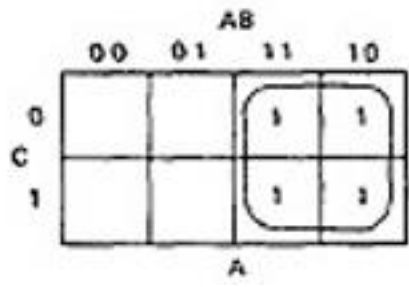
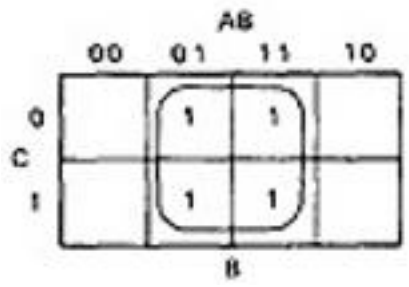
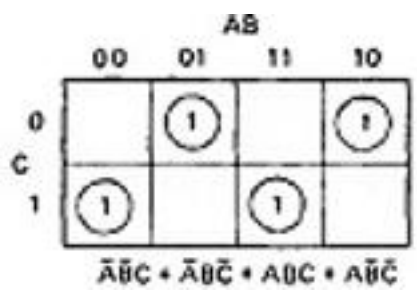
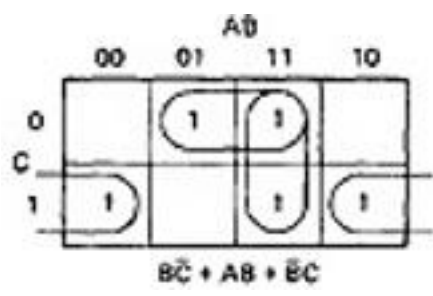
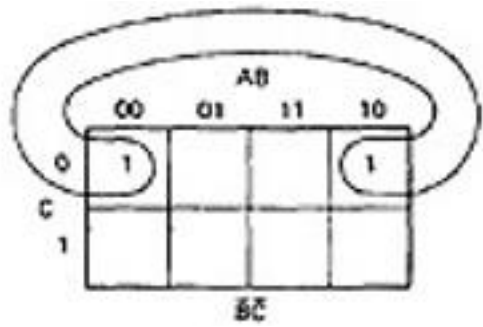
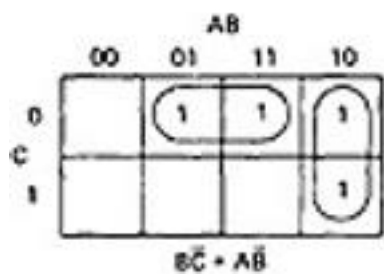


THREE VARIABLES:

- KV charts for three variables are shown below.
- 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 or 1.
- A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.

A few examples will illustrate the principles:





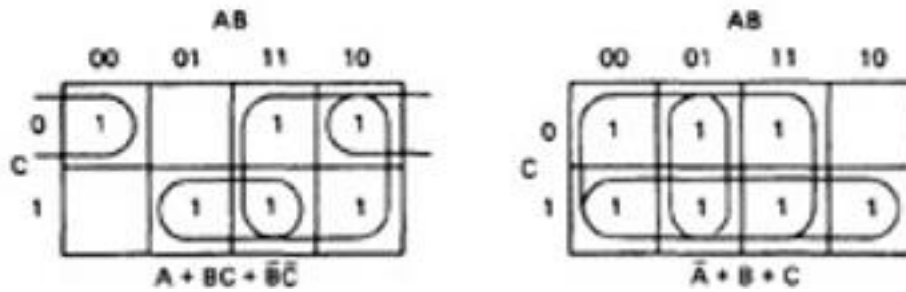
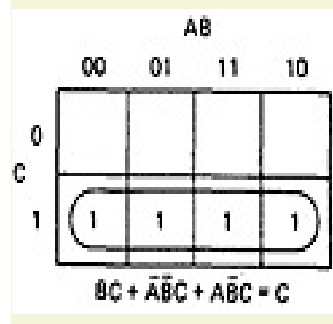
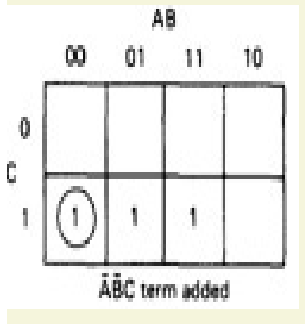
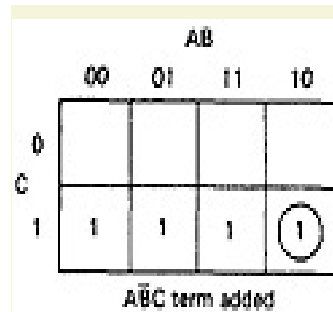
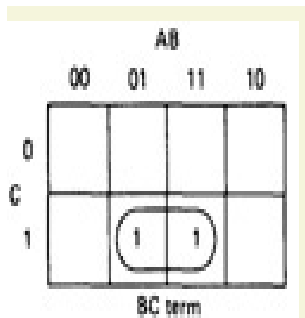


Figure 6.8 : KV Charts for Functions of Three Variables.

As an example, consider

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



KV CHARTS Four Variables and More

The same principles hold for four and more variables. A four variable chart and several possible adjacencies are shown below. Adjacencies can now consists of 1,2,4,8 and 16 boxes

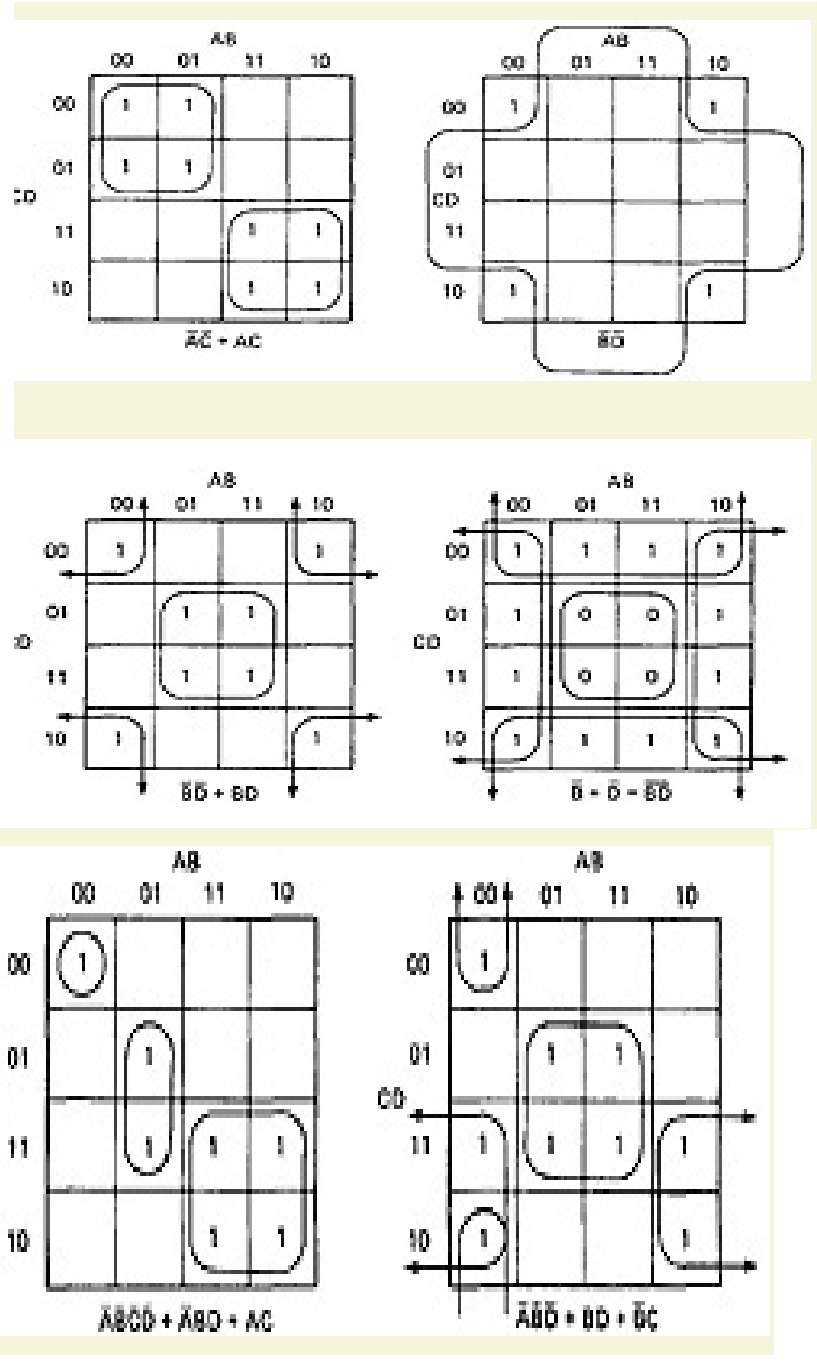
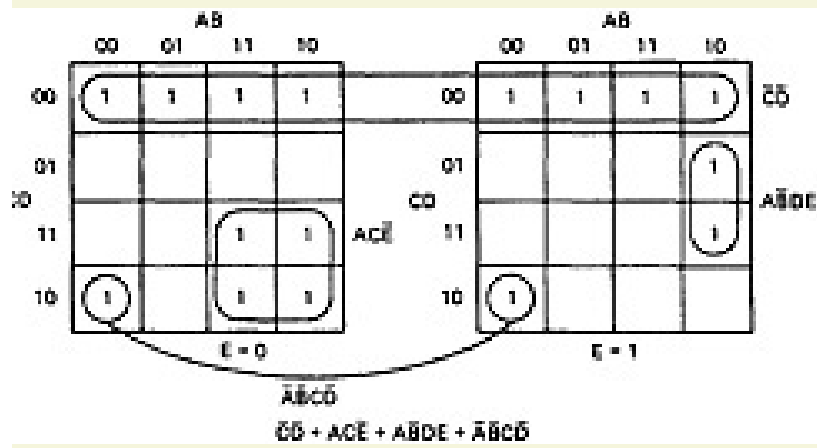
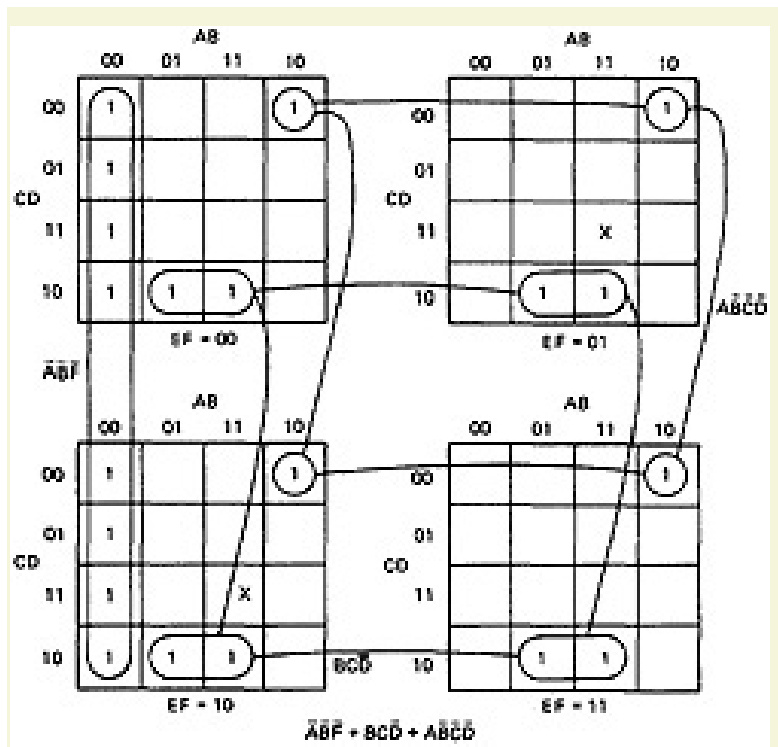


Figure: KV charts for 4 variables

- A five-variable KV chart is shown below



* For more hard sixth variable chart is there.



Even More Testing Strategies?

1. Use one prime implicant per domain. You'll obviously miss parts of the domain not covered by the prime implicant you chose—e.g., for disconnected domains.
2. Use the prime implicant with the fewest variables and work down to the prime implicant with the greatest number of variables
3. Overcome the obvious weaknesses of the above strategy (not all subdomains are covered) by using one test per prime implicant.

4. Every term in the product form for n variables has at most n literals but, because of simplifications made possible by adjacencies, terms may have fewer than n literals, say k . Any term with k literals can be expanded into two terms with $k + 1$ literals

4.4 SPECIFICATIONS

4.4.1. General

The procedure for specification validation is straightforward:

1. Rewrite the specification using consistent terminology.
2. Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.
3. Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it might seem.
4. Convert the rewritten specification into an equivalent set of boolean expressions
5. Identify the default action and cases, if any are specified.
6. Enter the boolean expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps, except for cases that result in multiple actions.
7. Enter the default cases and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the corresponding boxes of the KV chart back into English and get a clarification, explanation, or revision.
10. If the default cases were not specified explicitly, translate the default cases back into English and get a confirmation.

4.4.2. Finding and Translating the Logic

The specifications into sentences of the following form:

“IF predicate THEN action.”

If—based on, based upon, because, but, if, if and when, only if, only when, provided that, when, when or if, whenever.

Then—applies to, assign, consequently, do, implies that, infers that, initiate, means that, shall, should, then, will, would.

And—all, and, as well as, both, but, in conjunction with, coincidental with, consisting of, comprising, either . . . or, furthermore, in addition to, including, jointly, moreover, mutually, plus, together with, total, with.

OR—and, and if . . . then, and/or, alternatively, any of, anyone of, as well as, but, case, contrast, depending upon, each, either, either . . . or, except if, conversely, failing that, furthermore, in addition to, nor, not only . . . but, although, other than, otherwise, or, or else, on the other hand, plus.

NOT—but, but not, by contrast, besides, contrary, conversely, contrast, except if, excluding, excepting, fail, failing, less, neither, never, no, not, other than.

EXCLUSIVE OR—but, by contrast, conversely, nor, on the other hand, other than, or.

IMMATERIAL—independent of, irregardless, irrespective, irrelevant, regardless, but not if, whether or not.

4.4.3. Ambiguities and Contradictions

* Specification is

$$\begin{aligned}
 A1 &= \overline{BCD} + \overline{ABC}D \\
 A2 &= \overline{ACD} + \overline{AC}D + \overline{ABC} + \overline{ABC} \\
 A3 &= BD + \overline{BCD} \\
 ELSE &= \overline{BC} + \overline{ABC}D
 \end{aligned}$$

* Here is the KV chart for this specification (I've used the numerals 1, 2, 3, and 4 to represent the actions and the default case):

		AB			
		00	01	11	10
CD	00	4	1	1,2	2
	01		3	2,3	1,2
	11	4	3	3	4
	10	4	3	3	4

There is an ambiguity probably related to the default case : $\overline{A}\overline{B}\overline{C}\overline{D}$ is missing

When the specifier asks about his ambiguity the end users answers are contradictions.

It is necessary to resolve those ambiguities by presenting the complete specification in form of tables to the end user

4.4 4. Don't-Care and Impossible Terms

There are only three things in this universe that certain are impossible:

1. Solving a provably unsolvable problem, such as creating a universal program verifier.
2. Knowing both the exact position and the exact momentum of a fundamental particle.
3. Knowing what happened before the “big bang” that started the universe.

There are two kinds of so-called impossible conditions:

(1) the condition cannot be created or is seemingly contradictory or improbable;

(2) the condition results from our insistence on forcing a complex, continuous world into a binary, logical mold.

- Most program illogical conditions are of the latter kind. There are twelve cases for something, say, and we represent those cases by 4 bits
- Our conversion from the continuous world to binary world
- Taking advantage of impossible conditions is a dangerous practice and should be avoided but if you are insisted on doing that sort of the thing we must to do it right so the steps are as follows

1. Identify all “impossible” and “illogical” cases and confirm them.

2. Document the fact that you intend to take advantage of them.

3. Fill out the KV chart with the possible cases and then fill in the impossible cases. Use the combined symbol ϕ , which is to be interpreted as a 0 or 1, depending on which value provides the greatest simplification of the logic. These terms are called don't-care terms, because the case is presumed impossible, and we don't care which value (0 or 1) is used.

* Here is an example:

		AB			
		00	01	11	10
CD	00	0	1		
	01	1	0	0	
	11	ϕ	1	1	1
	10	1	1	1	1

* By not taking advantage of the impossible conditions, we get the resulting boolean expression

$$\overline{C}\overline{D} + CB + CA + \overline{A}\overline{B}\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D}$$

* By taking advantage of the impossible conditions, we get:

$$C + \overline{A}$$

* The corresponding flowgraphs are shown in below diagram

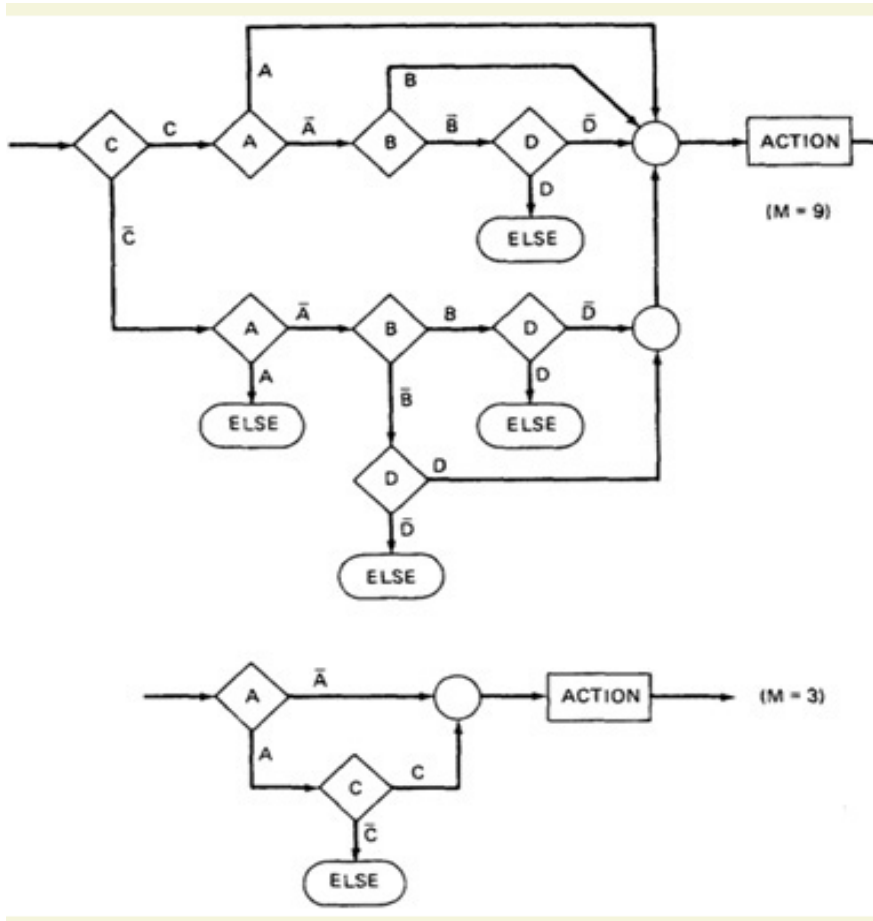


Fig: Reducing Complexity by Simplifying the Logic.

4.5 STATE GRAPHS

- The **state graph** and its associated **state table** are useful models for describing software behavior.
- The **finite-state machine** is a functional testing tool and testable design programming tool

4.5.1 Motivational Overview

- The **finite-state machine** is as fundamental to software engineering as boolean algebra.
- State testing strategies are based on the use of finite-state machine models for software structure, software behavior, or specifications of software behavior.
- Finite-state machines can also be implemented as table-driven software, in which case they are a powerful design option. Independent testers are likeliest to use a finite-state machine model as a guide to the design of functional tests—especially system tests.

4.5.2 STATE GRAPHS

- The word “**state**” is used in much the same way it’s used in ordinary English, as in “state of the union,” or “state of health.”
- The Oxford English Dictionary defines “state” as: “A combination of circumstances or attributes belonging for the time being to a person or thing.”
- States are denoted by nodes

A below program that detects the character sequence “ZCZC” can be in the following states:

- **1.** Neither ZCZC nor any part of it has been detected.
- **2.** Z has been detected.
- **3.** ZC has been detected.
- **4.** ZCZ has been detected.
- **5.** ZCZC has been detected

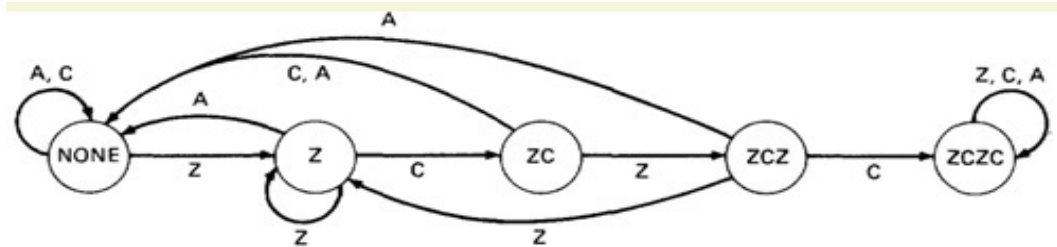


Figure: One-Time ZCZC sequence-Detector state graph

- A moving automobile whose engine is running can have the following states with respect to its transmission:
 1. Reverse gear
 2. Neutral gear
 3. First gear
 4. Second gear
 5. Third gear
 6. Fourth gear
- A person's checkbook can have the what states with respect to the bank balance:
 1. Equal
 2. Less than
 3. Greater than
- A word processing program menu can be in the what states with respect to file manipulation
 1. Create document
 2. Save document
 3. Rename document
 4. Copy document
 5. Delete document

4.5.3 Inputs and Transitions

- Result of those inputs, the state changes, or is said to have made a **transition**.
- Transitions are denoted by links that join the states
- The input that causes the transitions are marked on the link i.e, the inputs are link weights. There is an out link for every input

- If several inputs in a state cause a transition to the same subsequent state then we can abbreviate as "input1,input2,input3....."
- 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 between states.
- The ZCZC detection example can have the following kinds of inputs:
 1. Z
 2. C
 3. Any character other than Z or C, which we'll denote by A

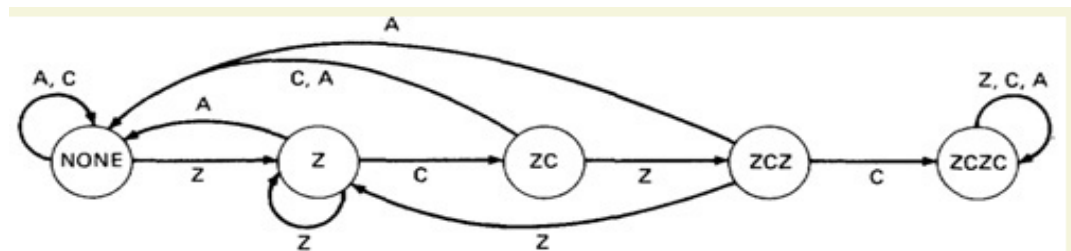


Figure 11.1 : One-Time ZCZC sequence-Detector state graph

The state graph of Figure 11.1 is interpreted as follows:

1. If the system is in the "NONE" state, any input other than a Z will keep it in that state.
2. If a Z is received, the system transitions to the "Z" state.
3. If the system is in the "Z" state and a Z is received, it will remain in the "Z" state. If a C is received, it will go to the "ZC" state; if any other character is received, it will go back to the "NONE" state because the sequence has been broken.
4. A Z received in the "ZC" state progresses to the "ZCZ" state, but any other character breaks the sequence and causes a return to the "NONE" state.
5. A C received in the "ZCZ" state completes the sequence and the system enters the "ZCZC" state. A Z breaks the sequence and causes a transition back to the "Z" state; any other character causes a return to the "NONE" state.
6. The system stays in the "ZCZC" state no matter what is received.

4.5.4 Outputs

- An Output can be associated with any link. Outputs are denoted by letters or words and are separated from inputs by a slash as follows: "input/output."

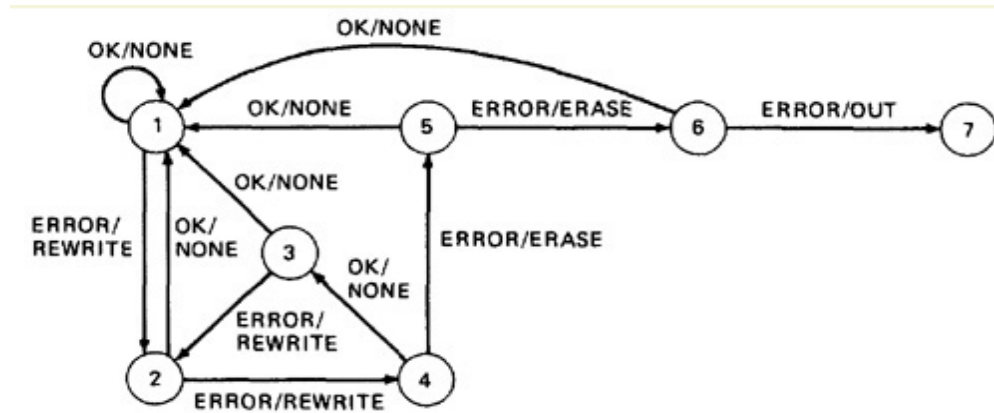


Figure 11.2 Tape control recovery routine state graph

- If no errors are detected then Input=OK and no special action is taken then Output=NONE
- If write error is detected then input=ERROR backspace the tape one block and rewrite the block output=REWRITE.
- If rewrite is successful then Input=OK
- If there have been two successive rewrites and a third error occurs backspace ten centimeters and erase forward then output=ERASE
- The inputs and actions have been simplified. There are only two kinds of inputs (OK, ERROR) and four kinds of outputs (REWRITE, ERASE, NONE, OUT-OF-SERVICE) from the figure 11.2

4.5.5 State Tables

- It is more convenient to represent the state graph as a table
 - The state table or the state transition table specifies the states, the inputs, the transitions, the outputs
 - The following conventions are used to represent the state table
1. Each row of the table corresponds to a state.
 2. Each column corresponds to an input condition.
 3. The box at the intersection of a row and column specifies the next state (the transition) and the output, if any

INPUT

STATE	OKAY	ERROR
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7

Table: state table for figure 11.2

We did not specify what happens in state 7 because there is no inputs and outputs to that state.

4.5.6 Time Versus Sequence

Time	Sequence
1. State graphs don't represent time	1. State graphs represent sequence.
2. A transition might take microseconds or centuries; a system could be in one state for milliseconds and another for eons, or the other way around;	2. The state graph would be the same because it has no notion of time

4.5.7 Software Implementation

4.5.7.1 Implementation and Operation

- The state graph represents the total behavior consisting of the transport, the software, the executive, the status returns, interrupts and so on.
- There is no simple correspondence between lines of code and states.
- The state table forms the basis for a widely used implementation shown in the PDL program below.

There are four tables involved:

- **1.** A table or process that encodes the input values into a compact list (INPUT_CODE_TABLE).
- **2.** A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).

- **3.** A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).
- **4.** A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport (DEVICE_TABLE).

The routine operates as follows, where # means concatenation:

```

BEGIN
PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
ACCEPT INPUT_VALUE
INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
POINTER := INPUT_CODE#PRESENT STATE
NEW_STATE := TRANSITION_TABLE(POINTER)
OUTPUT_CODE := OUTPUT_TABLE(POINTER)
CALL OUTPUT_HANDLER(OUTPUT_CODE)
DEVICE_TABLE(DEVICE_NAME) := NEW_STATE
END

```

- 1.** The present state is fetched from memory.
- 2.** The present input value is fetched. If it is already numerical, it can be used directly; otherwise, it may have to be encoded into a numerical value, say by use of a case statement, a table, or some other process.
- 3.** The present state and the input code are combined (e.g., concatenated) to yield a pointer (row and column) of the transition table and its logical image (the output table).
- 4.** The output table, either directly or via a case statement, contains a pointer to the routine to be executed (the output) for that state-input combination. The routine is invoked (possibly a trivial routine if no output is required).
- 5.** The same pointer is used to fetch the new state value, which is then stored.

4.5.7.2. Input Encoding and Input Alphabet

- The alternative to input encoding is a huge state graph and table because there must be one outlink in every state for every possible different input.
- Input encoding compresses the cases and therefore the state graph.

- Another advantage of input encoding is that we can run the machine from a mixture of otherwise incompatible input events, such as characters, device response codes, thermostat settings, or gearshift lever positions.
- The set of different encoded input values is called the **input alphabet**.
- For example in the ZCZC detector if ASCII Characters are included and we are interested to the inputs only Z,C,Other
- The input encoding could be implemented and it contains Other=0, Z=1,C=2.
- Alternatively we can implement as

If Input="Z" then go to code 1

Else if Input="C" then go to code 2

Else code=0

End if

4.5.7.3. Output Encoding and Output Alphabet

- There are only a finite number of such distinct actions, which we can encode into a convenient **output alphabet**
- The word output as used in the context of finite state machine as character from the output alphabet

4.5.7.4. State Codes and State-Symbol Products

- If there are n state and k different inputs both numbered and the state code and input code are S and I then their value is $Sk+I$ or $In+s$.
- Finite state machines are used in time-critical applications because they have response time
- A faster implementation it to use a binary number for input codes and to form the pointer by alternating the state and input code.
- The term **state-symbol product** is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table without holes.
- "State codes" in the context of finite-state machines, we mean the (possibly) hypothetical integer used to denote the state and not the actual form of the state code that could result from an encoding process.

4.5.7.5. Application Comments for Designers

- State-table implementation is advantageous when either the control function is likely to change in the future or when the system has many similar, but slightly different, control functions.
- This technique can provide fast response time—one pass through the above program can be done in ten to fifteen machine instruction execution times.
- It is not an effective technique for very small (four states or less) or big (256 states or more) state graphs.

4.5.7.6. Application Comments for Testers

- Independent testers are not usually concerned with either implementation details or the economics of this approach but with how a state-table or state-graph representation of the behavior of a program or system can help us to design effective tests.
- There is an interesting correlation, though: when a finite-state machine *model* is appropriate, so is a finite-state machine *implementation*.

4.6 . STATE GRAPHS - GOOD AND BAD STATE GRAPHS

4.6.1 General

Principles of judging a graph as a good and state graph are

- **1.** The total number of states is equal to the product of the possibilities of factors that make up the state.
- **2.** For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
- **3.** For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.*
- **4.** For every state there is a sequence of inputs that will drive the system back to the same state.**

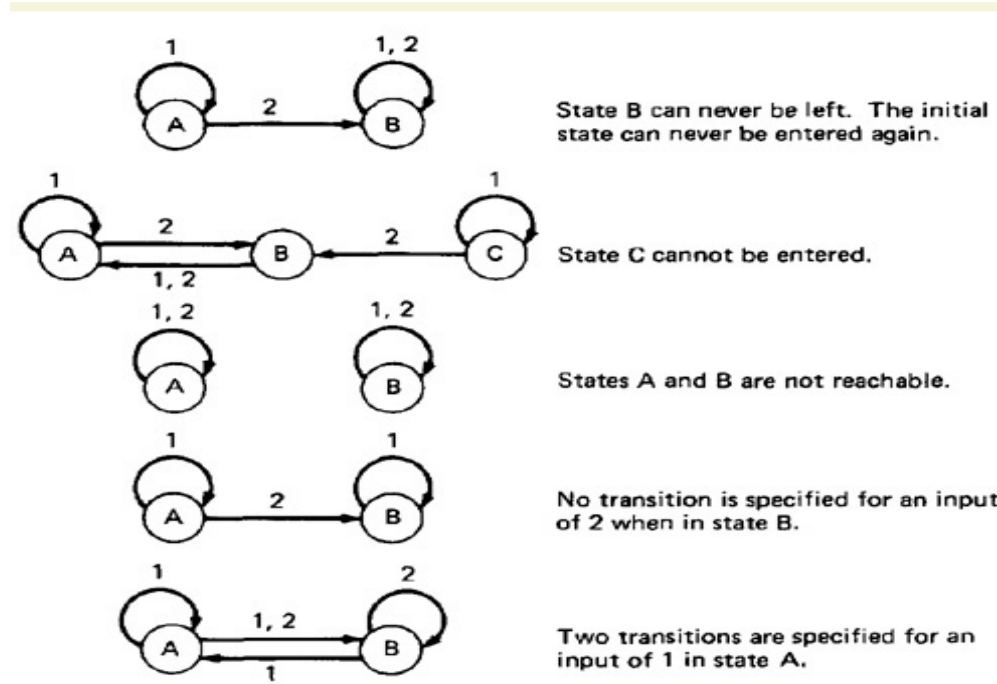


Figure 11.3 shows examples of improper state graphs.

There are two aspects of state graphs

1. The states with their transitions and the inputs that cause them
2. The outputs associated with the transitions
 - In state testing we may ignore outputs because if the states and transitions are the primary interest
 - Two state graphs with identical states, inputs and transitions would have different outputs yet from a state-testing point they could be identical.

4.6.2. State Bugs

4.6.2.1 Number of States

- The number of states in a state graph is the number of states we choose to recognize or model.
- In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base.
- Find the number of states as follows:
 1. Identify all the component factors of the state.
 2. Identify all the allowable values for each factor.

3. The number of states is the product of the number of allowable values of all the factors.

Components	Allowable values	
GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
	TOTAL	= 144 states

Table: various components of an Auto mobile and its allowable values and states

In above table it shows various components of an Auto mobile and its allowable values and states

- It is necessary to know the number of states before designing the test cases. There is no point in designing tests to check the systems behavior in various states if there is no agreement of how many states there are
- And if there is no agreement of how many states there are, there must be disagreement on what the system does in which states and transitions and the outputs.

4.6.2.2. Impossible States

- The states we deal with inside computers are not the stats of the real world but rather a numerical representation of real world states, the “impossible” states can occur
- A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical condition handler when they appear to have occurred.

Components	Allowable values	
GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
	TOTAL	= 144 states

Table: various components of an Auto mobile and its allowable values and states

- Actually, broken engine can not run so number of factors or states for engine condition and engine operation should be 3 rather than 4 so number of states equals to 108.
- Another impossible state is that broken transmission cannot move long so the number of states should be reduced
- State bugs raises due to the discrepancy between the programmers state count and the testers state count and this discrepancy is after due to a difference opinion concerning “Impossible states”.

4.6.2.3 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.

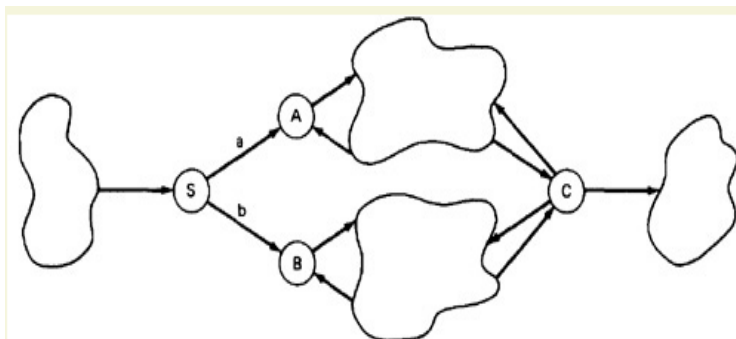


Figure 11.4. Equivalent States.

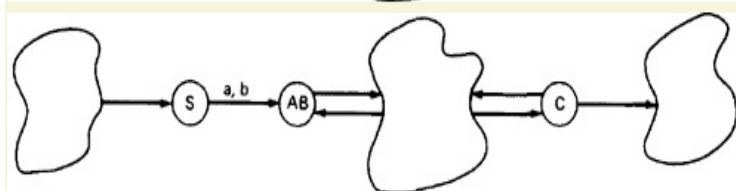


Figure 11.5. Equivalent States of Figure 11.4 Merged.

If starting from state A, Every possible sequence of inputs produces exactly the same sequence of outputs that would occur when starting from the state B. So A and B are equivalent states and they can be merged as shown in figure 11.5

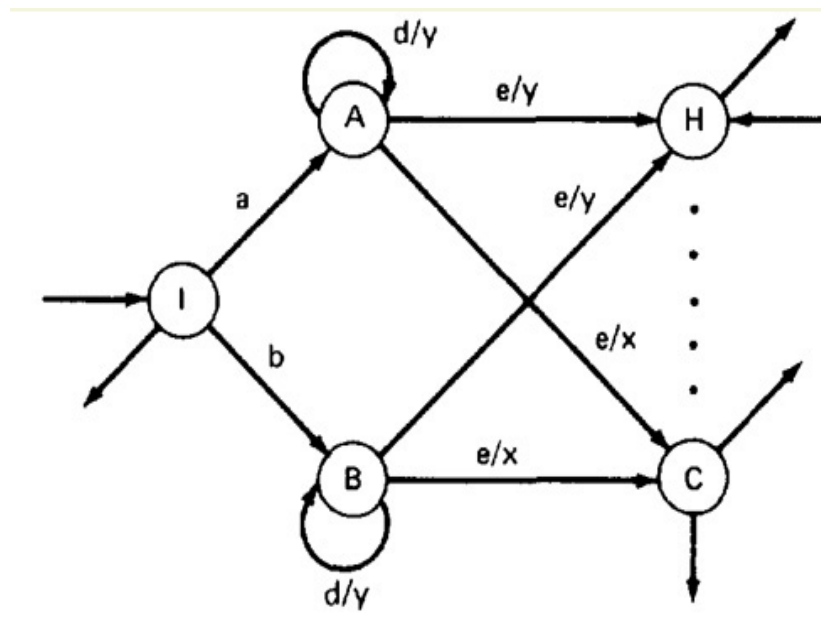


Figure 11.6 Equivalent states

Equivalent states can be recognized by the following procedures:

1. The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ. The two states are differentiated only by the input that distinguishes between them.
 - This situation is shown in Figure 11.6. Except for the a, b inputs, which distinguish between states A and B, the system's behavior in the two states is identical for every input sequence; they can be merged.
2. 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 be merged (see Figure 11.7).

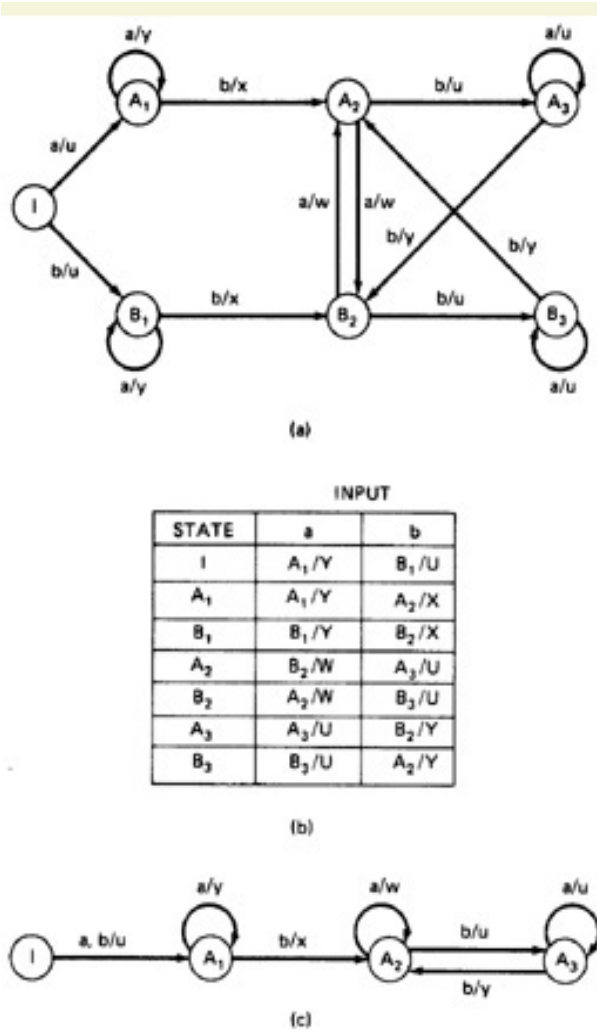


Figure 11.7. Merged Equivalent States.

In the figure 11.7

The states are equal i.e, $A_1=B_1$, $A_2=B_2$, $A_3=B_3$ means

A_1, B_1 are equivalent states

A_2, B_2 are equivalent states

A_3, B_3 are equivalent states

- With respect to input, output only the next state changes so the equivalent states can be merged
- The process can be automated because we are using state graphs as a design tool rather than a program design tool.
- The state graph of two versions(the tester's and the designer's) is usually enough to expose the similarities and the possibility of merging equivalent states.

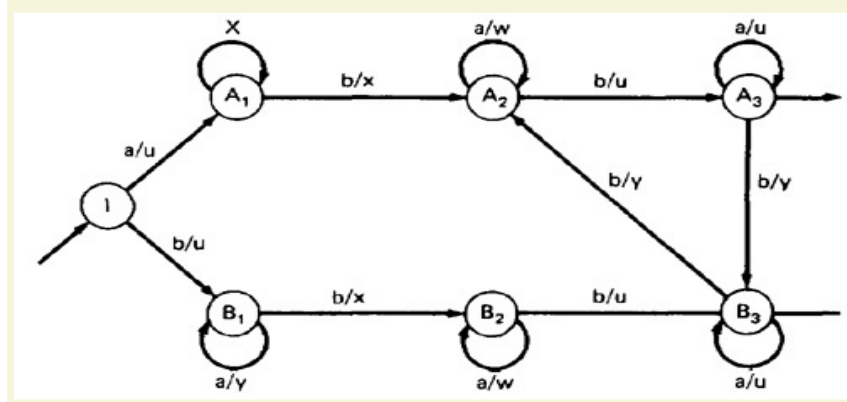


Figure 11.8 Unmergable procedure

- Bugs are often the result of the unjustifiable merger of seemingly equivalent states. Two states or two sets appear to be equivalent because the programmer has failed to carry through a proof of equivalence for every input sequence.
- From the figure 11.8 The input sequence abbbb produces the output sequence uxuyy while the input sequence bbbbb produces the output sequence uxuyu the two set of states are not equivalent.

4.6.3 Transition Bugs

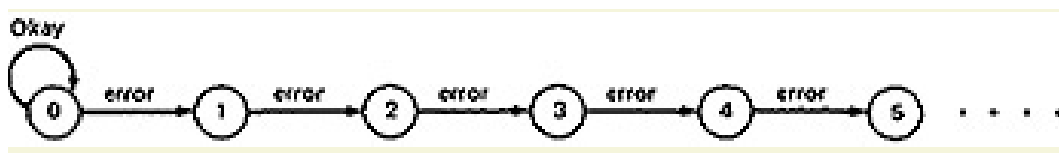
4.6.3.1. Unspecified and Contradictory Transitions

- Every input –state combination must have specified transition
- *Exactly one transition must be specified for every combination of input and state.*
- A program can't have contradictions or ambiguities. Ambiguities are impossible because the program will do *something* (right or wrong) for every input.
- One of the most common source of contradictions/ambiguities are specifications

4.6.3.2 An Example

- Specifications are one of the most common source of ambiguities and contradictions.
- The first statement of the specification:

Rule 1: The program will maintain an error counter, which will be incremented whenever there's an error.



- There are only two input values “okay” and “error”. A state table will be easier to work and its very easy to spot ambiguities and contradictions.. The state table for the above graph is shown below

Input

STATE	OKAY	ERROR
0	0/none	1/
1		2/
2		3/
3		4/
4		5/
5		6/
6		7/
7		8/

Rule 2: If there is an error, rewrite the block

Input

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE
4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block

Input

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE, ERASE, REWRITE
3		4/REWRITE, ERASE, REWRITE
4		5/REWRITE, ERASE, REWRITE
5		6/REWRITE, ERASE, REWRITE
6		7/REWRITE, ERASE, REWRITE
7		8/REWRITE, ERASE, REWRITE

- Rule 3, if followed blindly, causes an unnecessary rewrite. It's a minor bug, so let it go for now, but it pays to check such things. There might be an arcane security reason for rewriting, erasing, and then rewriting again.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service

Input

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3		4/ER, RW
4		5/ER, RW
5		6/OUT
6		
7		

Rule 4 terminates our interest in this state graph so we can dispose of states beyond 6. The details of state 6 will not be covered by this specification; presumably there is a way to get back to state 0. Also, we can credit the specifier with enough intelligence not to have expected a useless rewrite and erase prior to going out of service.

Rule 5: If the erasure is successful, return to the normal state and clear the counter

Input

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/RW,ER, RW
3	0/NONE	4/RW,ER, RW
4	0/NONE	5/RW,ER, RW
5	0/NONE	6/OUT
6		

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state and try another rewrite

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

Input

STATE	OKAY	ERROR
0	0/NONE	1/RW
1	0/NONE	2/RW
2	1/NONE	3/RW,ER, RW
3	0/NONE 2/NONE	4/RW,ER, RW
4	0/NONE 3/NONE	5/RW,ER, RW
5	0/NONE 4/NONE	6/OUT
6		

For example say that we are at state 1. The error counter here is 2. If we have a successful (okay) rewrite, the entry for okay at state 2 will have error counter decremented by 1 which is (2-1)/NONE

4.6.3.3 unreachable state

- An **unreachable state** is like unreachable code—a state that no input sequence can reach.
- An unreachable state is not impossible, just as unreachable code is not impossible.

There are two possibilities

1. There is a bug that is some transitions are missing
2. The transitions are there but you don't know about it

Typically such hidden transitions are caused by software opening at a higher priority level or by interrupt processing.

4.6.3.4 Dead States

- A dead state, (or set of dead states) is a state that once entered cannot be left.
- A set of states may appear to be dead because the program has two modes of operation.
- In the first mode it goes through initialization process that consists of several states
- Once initialized it goes strongly connected states of working states, which with in the context of the routine which can not be exited
- The initialization states are unreachable to the working states, and the working states are to be dead to the initialization states
- The only way to get back from dead state is either system crash or restart.

4.6.3.4. Output Errors

- The states, the transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect.
- The likeliest reason for an incorrect output is an incorrect call to the routine that executes the output. This is localized and it is a minor bug

4.6.3.5. Encoding Bugs

- *The behavior of a finite-state machine is invariant under all encodings.* That is, say that the states are numbered 1 to n.

4.7 STATE TESTING

4.7.1. Impact of Bugs

- A bug can manifest itself as one or more of the following symptoms
 1. Wrong number of states.
 2. Wrong transition for a given state-input combination.
 3. Wrong output for a given transition.
 4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
 5. States or sets of states that are split to create inequivalent duplicates.
 6. States or sets of states that have become dead.
 7. States or sets of states that have become unreachable.

4.7.2. Principles

- A path in a state graph, of course is a succession of transitions caused by sequence of inputs
- The notion of coverage is identical to that used for flow graphs i.e., each transition must be exercised
- The starting point of state testing is:
 - 1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
 - 2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.
- A set of tests, then, consists of three sets of sequences:
 - 1. Input sequences.
 - 2. Corresponding transitions or next-state names.
 - 3. Output sequences.

4.7.3. Limitations and Extensions

1. Simply identifying the factors that contribute to the state, calculating the total number of states, and comparing this number to the designer's notion catches some bugs.
2. Insisting on a justification for all supposedly dead, unreachable, and impossible states and transitions catches a few more bugs.
3. Insisting on an explicit specification of the transition and output for every combination of input and state catches many more bugs.
4. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.
5. In executing state tests, it is essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

4.7.4. What to Model

1. 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 is important.

2. Most protocols between systems, between humans and machines, between components of a system (CHOI84, CHUN78, SARI88).

3. Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.

4. Transaction flows where the transactions are such that they can stay in the system indefinitely—for example, online users, tasks in a multitasking system.

5. High-level control functions within an operating system. Transitions between user states, supervisor's states, and so on. Security handling of records, permission for read/write/modify privileges, priority interrupts and transitions between interrupt states and levels, recovery issues and the safety state of records and/or processes with respect to recording recovery data

6. The behavior of the system with respect to resource management and what it will do when various levels of resource utilization are reached. Any control function that involves responses to thresholds where the system's action depends not just on the threshold value, but also on the direction in which the threshold is crossed. This is a normal approach to control functions

7. A set of menus and ways that one can go from one to the other. The currently active menus are the states, the input alphabet is the choices one can make, and the transitions are invocations of the next menu in a menu tree. Many menu-driven software packages suffer from dead states—menus from which the only way out is to reboot.

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

4.7.5. Getting the Data

- State testing, more than most functional test strategies, tends to have a labor-intensive data-gathering phase and tends to need many more meetings to resolve issues.

4.7.6. Tools

- There are tools to do the same for hardware logic designs. That's the good news.
- The bad news is that these systems and languages are proprietary, of the home-brew variety, internal, and/or not applicable to the general use of software implementations of finite-state machines

4.8 TESTABILITY TIPS

4.8.1. A Balm for Programmers

- Testability is to test the software

- The key to testability design is easy: build explicit finite-state machines

4.8.2. How Big, How Small?

- There are about eighty possible good and bad three-state machines, 2700 four-state machines, 275,000 five-state machines, and close to 100 million six-state machines, most of which are bad.
- You must do finite state machine model and identify how you are implementing every part of that model for anything with four or five states

4.8.3 Switches, Flags, and Unachievable Paths

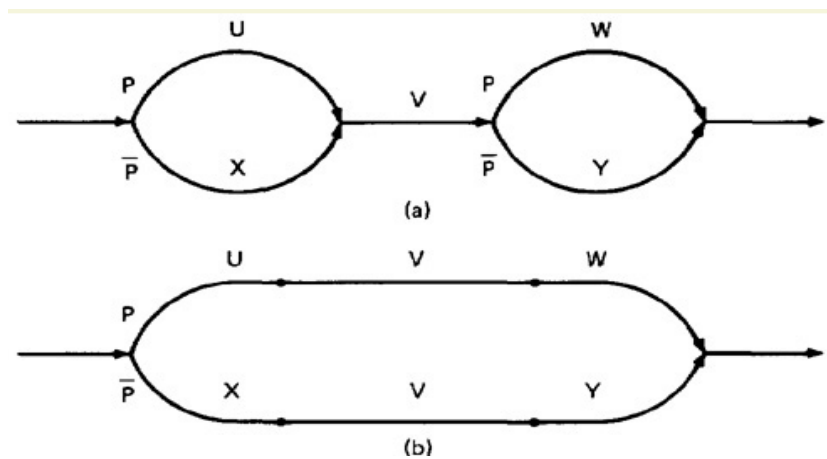


Figure 11.9 program with one switch variable

- The above figure 11.9a shows one switch or flag. In figure 11.9b we have rewritten the routine to eliminate the flag as soon as the flag value is calculated we branch the cost is the cost of converting segment V into a subroutine and calling it twice.
- We went from four paths two of which are achievable to two paths, both of which are achievable and both of which are needed to achieve branch coverage.

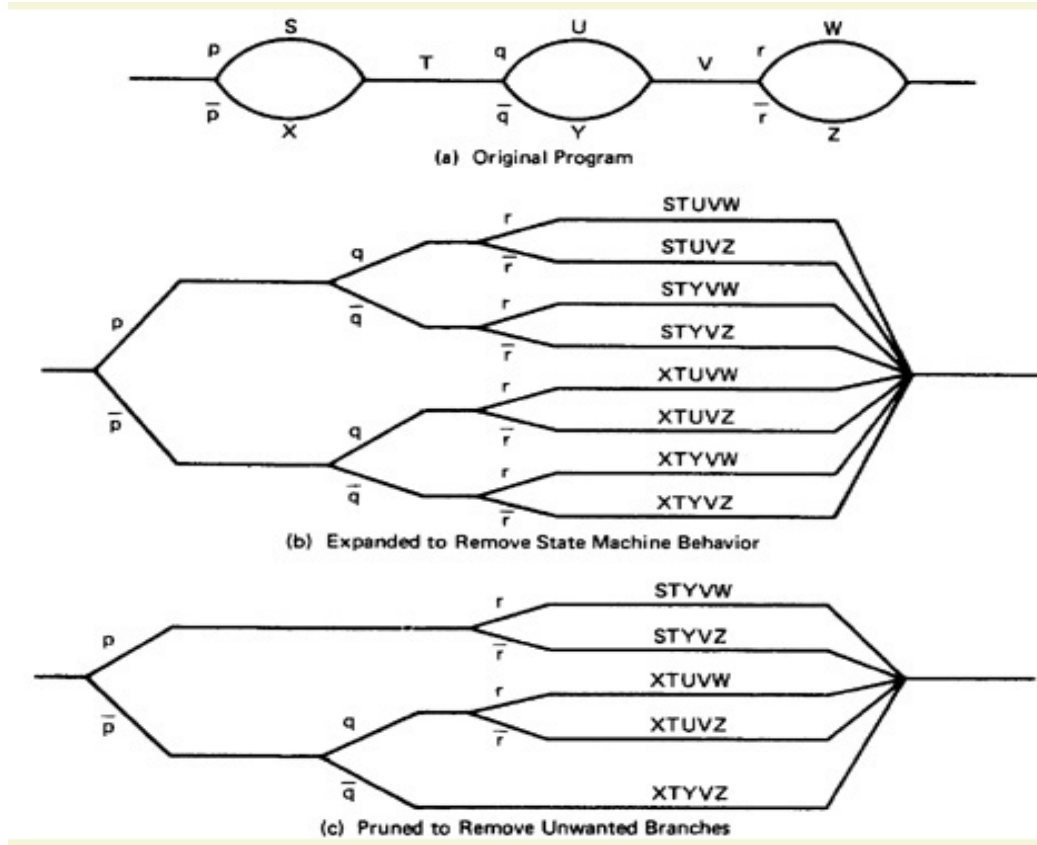


Figure 11.10: Three-Switch-Variable Program.

- In figure 11.10 the figure is complicated there are three switches . We can put the decision up front and branch directly, and again we use subroutines to make each path explicitly and do with the switches.
- The advantages of this implementation is that if any of the combinations are not needed we clip out that part
- In figure 11.10c the paths are achievable and all paths are needed for branch cover

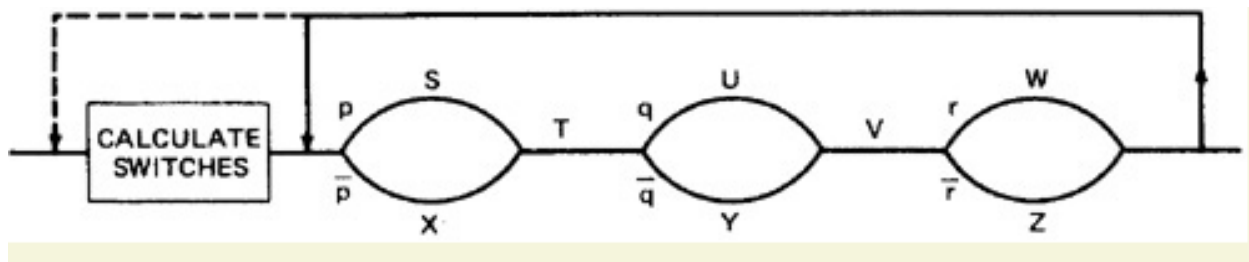


Figure 11.11 Switches in a Loop.

- The above figure is similar to the previous but we have put the switched parts in a loop. We now have a very difficult situation. We don't know which paths are

achievable and which are not required it depends on the switch settings we must do o attempt branch coverage in every possible state.

4.8.4. Essential and Inessential Finite-State Behavior

- The simplest essential finite-state machine is a flip-flop. There is no logic that can implement it without some kind of feedback. we cannot describe this behavior by a decision table or decision tree unless you provide feedback into the table or call it recursively.

4.8.5. Design Guidelines

1. Start by designing the abstract machine. Verify that it is what you want to do. Do an explicit analysis, in the form of a state graph or table, for anything with three states or more.
2. Start with an explicit design—that is, input encoding, output encoding, state code assignment, transition table, output table, state storage, and how you intend to form the state-symbol product. Do this at the PDL level. But be sure to document that explicit design.
3. Before you start taking shortcuts, see if it really matters. Neither the time nor the memory for the explicit implementation usually matters
4. Take shortcuts by making things implicit only as you must to make significant reductions in time or space and only if you can show that such savings matter in the context of the whole system.
5. After all, doubling the speed of your implementation may mean nothing if all you've done is shaved 100 microseconds from a 500-millisecond process. The order in which you should make things implicit are: output encoding, input encoding, state code, state-symbol product, output table, transition table, state storage. That's the order from least to most dangerous.
6. Consider a hierarchical design if you have more than a few dozen states.
7. Build, buy, or implement tools and languages that implement finite-state machines as software if you're doing more than a dozen states routinely.
8. Build in the means to initialize to any arbitrary state. Build in the transition verification instrumentation (the coverage analyzer). These are much easier to do with an explicit machine