<ch9 toc ch11>

Chapter 10
LOGIC-BASED TESTING

1. SYNOPSIS

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-Veitch charts**.

2. MOTIVATIONAL OVERVIEW

2.1. Programmers and Logic

"Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques. This chapter concerns logic in its simplest form, **boolean algebra**, and its application to program and specification test and design. 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 those techniques.

2.2. Hardware Logic Testing

Logic has been, for several decades, the primary tool of hardware logic designers. Today, hardware logic design and more important in the context of this book, hardware logic *test* design, are intensely automated. 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.*

---

*Hardware testing is ahead of software testing not because hardware testers are smarter than software testers but because hardware testing is easier. I say this from the perspective of a former hardware logic designer and tester who has worked in both camps and therefore has a basis for comparison.

Hardware testing methods will eventually filter into the software testers' toolkits, but there's another ongoing trend that provides an even stronger motivation. The distinction between hardware and software is blurring. Hardware designers talk about **silicon compilers**—compilers that start with a specification in a high-order language and automatically produce integrated circuit layouts. Similarly, the decision to implement a feature as hardware or software may one day be left to a silicon/software compiler. The hardware designers look more like programmers each day and we can expect them to meet at a middle ground where boolean algebra will be basic to their common language of discourse.

2.3. Specification Systems and Languages (BERZ85, CHIU85, HAYE85, KEMM85, VESS86)

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. The impact of specification systems and

languages was discussed in Chapter 2, Section 3.2.4.

The trouble with specifications is that they're hard to express. Logicians have struggled with this problem since Aristotle's time. Boolean algebra (also known as the **sentential calculus**) is the most basic of all logic systems. Higher-order logic systems are needed and used for formal specifications. That does not necessarily mean that future programmers will have to live with upside-down A's and backward E's because 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. So even if you have a relatively painless specification system, to understand why your specification was rejected or how it was transformed, you'll have to understand boolean algebra.

2.4. Knowledge-Based Systems

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 (WATE86). 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**. From the point of view of testing, there's nothing special about the inference engine—it's just another piece of software to be tested, and the methods discussed in this book apply. When we talk about testing knowledge-based systems, it's not the inference engine that concerns us, but testing the validity of the expert's knowledge and the correctness of the transcription (i.e., coding) of that knowledge into a rule base. Both of these testing areas are beyond the scope of this book; but understanding knowledge-based systems and their validation problems requires an understanding of formal logic (BELL87) to which the content of this chapter is basic.

2.5. Overview

We start with **decision tables** because: they 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. The next step is a review of boolean algebra (included to make this book self-contained). I included decision tables because the engineering/mathematically trained programmer may not be familiar with them, and boolean algebra because the business data processing programmer may have had only cursory exposure to it. That gets both kinds of readers to a common base, which is the use of decision tables and/or boolean algebra in test and software design.

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-Veitch** diagram is that conceptual tool. Few programmers, unless they're retreaded logic designers like me, or unless they had a fling with hardware design, learn about this method of doing boolean algebra. Without it, boolean algebra is tedious and error-prone and I don't wonder that people won't use it. With it, with practice, boolean algebra is no worse than arithmetic.

3. DECISION TABLES (HURL83, VESS86)

3.1. Definitions and Notation

Table 10.1 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. Table 10.1 can be translated as follows:

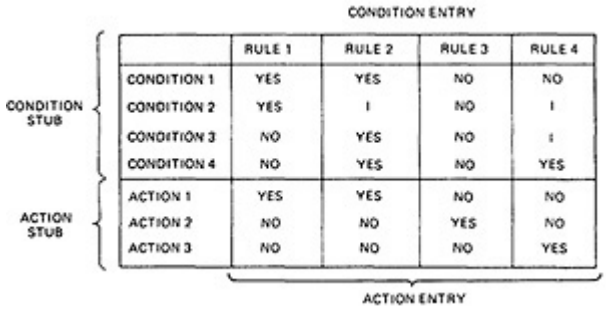
Table 10.1. An Example of a Decision Table.

> **1a.** 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: either a predicate in a specification or a control-flow predicate in a program. To say that a "condition is satisfied or met" is to say that the predicate is true. Similarly for "not met" and "false." Decision-table literature uses "condition" and "satisfied" or "met." In this book we prefer to use "predicate" and TRUE/FALSE.

Restating the conditions for action 1 to take place:

> **1b.** 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).



	RULE 5	RULE 6	RULE 7	RULE 8
CONDITION 1	1	NO	YES	YES

CONDITION 2	1	YES	1	NO
CONDITION 3	YES	1	NO	NO
CONDITION 4	NO	NO	YES	1
DEFAULT ACTION	YES	YES	YES	YES

Table 10.2. The Default Rules for Table 10.1.

It is not obvious from looking at this specification whether or not all sixteen possible combinations of the four predicates have been covered. In fact, they have not; a combination of YES, NO, NO, NO, for predicates 1 through 4 respectively, is not covered by these rules. In addition to the stated rules, therefore, we also need a **default rule** that specifies the default action to be taken when all other rules fail. The default rules for Table 10.1 is shown in Table 10.2. Decision tables can be specified in specialized languages that will automatically create the default rules.* Consequently, when programming in such languages, it is not necessary to make an explicit default specification; but when decision tables are used as a tool for test case design and specification analysis, the default rules and their associated predicates must be explicitly specified or determined. If the set of rules covers all the combinations of TRUE/FALSE (YES/NO) for the predicates, a default specification is not needed.

*Exactly the same situation occurs for languages with an IF P_1 THEN . . . ELSE IF P_2 . . . ELSE . . . construct. The last ELSE is the default case executed when none of the previous predicates is satisfied.

3.2. Decision-Table Processors

Decision tables can be automatically translated into code and, as such, are a higher-order language. The decision table's translator checks the source decision table for consistency and completeness and fills in any required default rules. The usual processing order in the resulting object code is, first, to examine rule 1. 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 as a source language have the virtue of clarity, direct correspondence to specifications, and maintainability. The principal deficiency is possible object-code inefficiency. There was a time when it was thought by some that decision tables would herald a new era in programming. Their use, it was claimed, would eliminate most bugs and poor programming practices and would reduce testing to trivia. Such claims are rarely made now, but despite such unrealistically high hopes, decision tables have become entrenched as a useful tool in the programmer's kit, especially in business data processing.

3.3. Decision Tables as a Basis for Test Case Design

If a specification is given as a decision table, it follows that decision tables should be used for test case design. Similarly, if a program's logic is to be implemented as decision tables, decision tables should also be used as a basis for test design. But if that's so, the consistency and completeness of the decision table is checked by the decision-table processor; therefore, it would seem that there would be no need to design those test cases. True, testing is not needed to expose contradictions and inconsistencies, but testing is still needed to determine whether the rules themselves are correct and to expose possible bugs in processing on which the rules' predicates depend.

Even if you can specify the program's logic as a decision table, it is not always possible or desirable to implement the program as a decision table because the program's logical behavior is only part of its behavior. The program interfaces with other programs, there are restrictions, or the decision-table language may not have needed features. Any of these reasons could be sufficient to reject a decision-table implementation. The use of a decision-table *model* to design tests is warranted when:

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.

These restrictions mean that the action selected is based on the combination of predicate truth values and nothing else. It might seem that these restrictions eliminate many potential applications but, despite external appearances, the rule evaluation order often doesn't matter. For example, if you use an automatic teller machine, the card must be valid, you must enter the right password, and there must be enough money in your account. It really doesn't matter in which order these predicates are evaluated. The specific order chosen may be sensible in that it might be more efficient, but the order is not inherent in the program's logic: the ordering is a by-product of the way we mapped parallel data flows into a sequential, Von Neumann language.

The above restrictions have further implications: (1) the rules are complete in the sense that every combination of predicate truth values, including the default cases, are inherent in the decision table, and (2) the rules are consistent if and only if every combination of predicate truth values results in only one action or set of actions. If the rules were inconsistent, that is if at least one combination of predicate truth values was implicit in two or more rules, then the action taken could depend on the order in which the rules were examined in contradiction to requirement 3 above. If the set of rules were incomplete, there could be a combination of inputs for which no action, normal or default, were specified and the routine's action would be unpredictable.

3.4. Expansion of Immaterial Cases

Improperly specified **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,

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."

The above rules state that age is material for a male's raise, but immaterial for determining a female's raise. No one would suggest that females either under or over 30 are impossible. If there are n predicates there are 2^n cases to consider. You find the cases by expanding the immaterial cases. This is done by converting each I entry into a pair of entries, one with a YES and the other with a NO. Each I entry in a rule doubles the number of cases in the expansion of that rule. Rule 2 in Table 10.1 contains one I entry and therefore expands into two equivalent subrules. Rule 4 contains two I entries and therefore expands into four subrules. The expansion of rules 2 and 4 are shown in Table 10.3.

Table 10.3. Expansion of Immaterial Cases for Rules 2 and 4.

Rule 2 has been expanded by converting the I entry for condition 2 into a separate rule 2.1 for YES and 2.2 for NO. Similarly, condition 2 was expanded in rule 4 to yield intermediate rules 4.1/4.2 and 4.3/4.4, which were then expanded via condition 3 to yield the four subrules shown.

The key to test case design based on decision tables is to expand the immaterial entries and to generate tests that correspond to all the subrules that result. If some conditions are three-way, an immaterial entry expands into three subrules. Similarly, an immaterial *n*-way case statement expands into *n* subrules.

If no default rules are given, then all cases not covered by explicit rules are perforce default rules (or are intended to be). If default rules are given, then you must test the specification for consistency. The specification is complete if and only if *n* (binary) conditions expand into exactly 2^n unique subrules. It is consistent if and only if all rules expand into subrules whose condition combinations do not match those of any other rules. Table 10.4 is an example of an inconsistent specification in which the expansion of two rules yields a contradiction.

	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 10.4. The Expansion of an Inconsistent Specification.

Rules I and 2 are contradictory because the expansion of rule 1 via condition 2 leads to the same set of predicate truth values as the expansion of rule 2 via condition 3. Therefore action 1 or action 2 is taken depending on which rule is evaluated first.

3.5. Test Case Design (GOOD75, LEWA76, TAIA87)

Test case design by decision tables begins with examining the specification's consistency and completeness. This is done by expanding all immaterial cases and checking the expanded tables. Also, make the default case explicit and treat it as just another set of rules for the default action. Efficient methods are given in later sections. Once the specifications have been verified, the objective of the test cases is to show that the implementation provides the correct action for all combinations of predicate values.

1. If there are k rules over n binary predicates, there are at least k cases to consider and at most 2^n cases. You can design test cases based on the unexpanded rules, with one case per rule, or based on the expanded rules with 2^n tests. Find input values to force each case.

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. Similarly, an input packet could consist of several dozen or possibly a variable number of input blocks, each of which was self-identifying and could therefore be tolerated in any order. Finally, complicated forms can have conditional subforms, which can result in the same data being input in different orders, depending on the path taken through the form and on the values of previous data. All such examples can result in variable orders of predicate evaluation and/or variable orders of rule evaluation. but if the implementation allows the order to be changed by input values, augment the test cases by using different predicate evaluation orders. Try all pairs of interchanges for a representative set of values. For example, if the normal order is predicate A followed by B, try a test in which B is followed by A. For N predicates, there will be $N(N - 1)/2$ interchanges for each combination of predicate truth values.

---

*Predicate and rule evaluation order could be variable in a system in which operators use explicit identifiers for every input field in a command. For example, the input specification might be of the form

A = 1, B = 3, C = 17, D = ABCDE. . ., and so on.

A tolerant implementation would allow these fields in any order. Similarly, an input packet could consist of several dozen or possibly a variable number of input blocks, each of which was self-identifying and could therefore be tolerated in any order. Finally, complicated forms can have conditional subforms, which can result in the same data being input in different orders, depending on the path taken through the form and on the values of previous data. All such examples can result in variable orders of predicate evaluation and/or variable orders of rule evaluation.

---

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. One set of predicate values per rule should suffice.

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. Instrument the paths from the rule processors to the actions so that you can show that the correct action was invoked for each rule.

3.6. Decision Tables and Structure

Decision tables can also be used to examine a program's structure (GOOD75). Figure 10.1 shows a program segment that consists of a

decision tree. These decisions, in various combinations, can lead to actions 1, 2, or 3. Does this flowgraph correspond to a complete and consistent set of conditions?

The corresponding decision table is shown in Table 10.5. You can almost read it from the flowgraph. 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. Expanding the immaterial cases for Table 10.5 leads to Table 10.6.

Sixteen cases are represented in Table 10.5, 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. I found my bug that way.

	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

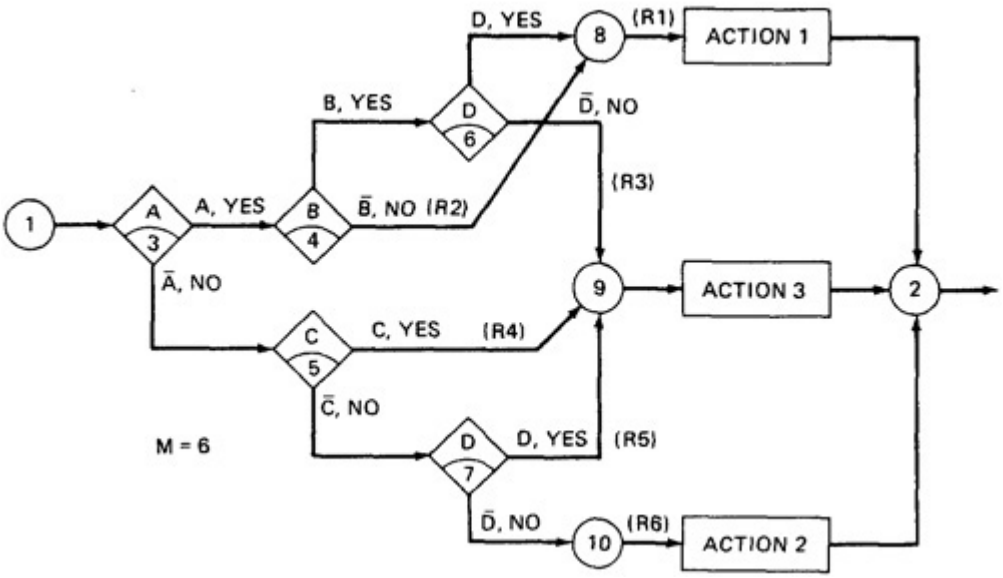


Figure 10.1. A Sample Program.

	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	1	1	1

CONDITION C	1	1	1	YES	NO	NO
CONDITION D	YES	1	NO	1	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 10.5. The Decision Table Corresponding to Figure 10.1.

	R1	**RULE 2**	**R3**	**RULE 4**	**R5**	**R6**
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

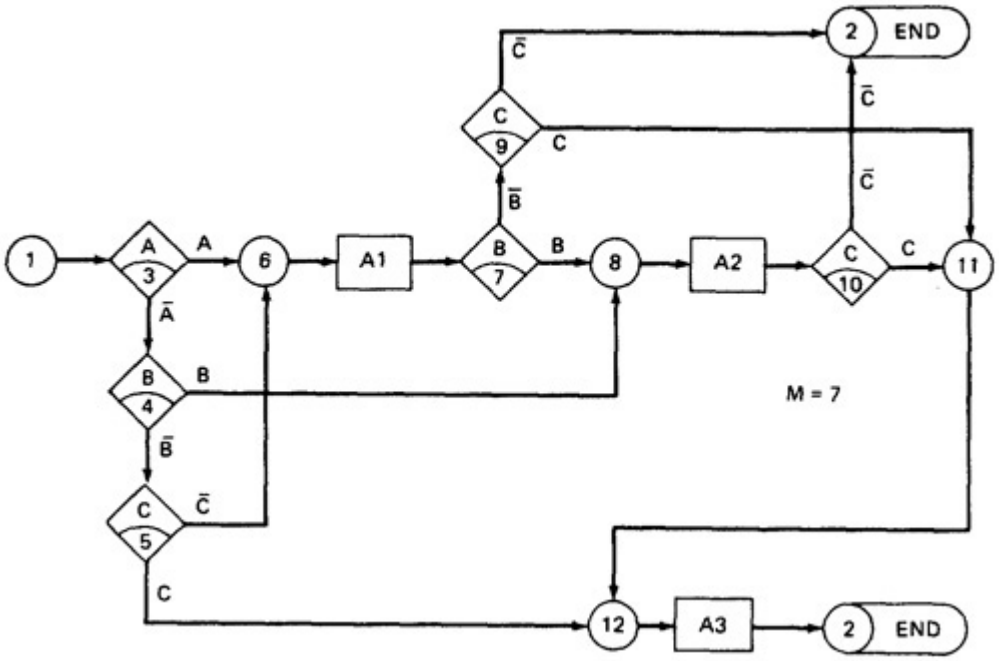
Table 10.6. The Expansion of Table 10.5.

Consider the following specification whose putative flowgraph is shown in Figure 10.2:

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).

Table 10.7 shows the conversion of this flowgraph into a decision table after expansion. There are eight cases, and all paths lead to the evaluation of all three predicates even though some predicates on some paths may be evaluated more than once. We can use the predicate notation introduced in Chapter 3 and name the rules by their corresponding combination of predicate truth values to make things clearer.

Figure 10.2. A Troublesome Program.



As clever as this design may seem, perhaps because it is clever, it has a bug. 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 and A3. This would have been easily found by the techniques of the next section, but for the moment, if the processes had been instrumented and if all eight cases were used to test, the bug would have been revealed. Note that testing based on structure alone would reveal nothing because the design was at fault—it did something, but what it did didn't match the specification.

Table 10.7. Decision Table for Figure 10.2.

4. PATH EXPRESSIONS AGAIN

4.1. General

4.1.1. The Model

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. As with all test techniques, we start with a model: we focus on one program characteristic and ignore the others. In logic-based testing we focus on the truth values of control flow predicates.

4.1.2. Predicates and Relational Operators

A **predicate** is implemented as a process whose outcome is a truth-functional value. Don't restrict your notion of predicate to

arithmetic relations such as >, >=, =, <, <, ≠. Predicates are based on **relational operators**, of which the arithmetic relational operators are merely the most common. Here's a sample of some other relational operators: ... is a member of ..., ... is a subset of ..., ... is a substring of ..., ... is a subgraph of ..., ... dominates ..., ... is dominated by ..., ... is the greatest lower bound of ..., ... hides ..., ... is in the shadow of ..., ... is above ..., ... is below .... The point about thinking of predicates as processes that yield truth values is that it usually pays to look at predicates top-down—typically from the point of view of predicates as specified in requirements rather than from the point of view of predicates as implemented. Almost all programming languages have arithmetic relational operators and few others. Therefore, in most languages you construct the predicates you need by using the more primitive arithmetic relations. For example, you need a set membership predicate for numbers (e.g., . . . is a member of . . .) but it's implemented as an equality predicate in a loop that scans the whole set.

4.1.3. Case Statements and Multivalued Logics

Predicates need not be restricted to binary truth values (TRUE/FALSE). There are multiway predicates, of which the FORTRAN three-way IF is the most notorious and the case statement the most useful. There are multivalued logics such as Post-algebras that can be used to analyze such predicate structures, but their use is technically difficult and semantically tricky. Three-valued logics are used routinely at the hardware interfaces between chips to reduce the number signal lines needed. For our purpose, logic-based testing is restricted to binary predicates. If you have case statements, you have to analyze things one case at a time if you're to use these methods. If you have many case statements, there will be a lot of bookkeeping and you're probably pushing the applicability of logic-based testing.

4.1.4. What Goes Wrong with Predicates

Several things can go wrong with predicates, especially if the predicate has to be interpreted

in order to express it as a predicate over input values.

1. The wrong relational operator is used: e.g., > instead of <=.
2. The predicate expression of a compound predicate is incorrect: e.g., A + B instead of AB.
3. The wrong operands are used: e.g., A > X instead of A > Z.
4. The processing leading to the predicate (along the predicate's interpretation path) is faulty.

Logic-based testing is useful against the first two bug types, whereas data flow testing is more useful against the latter two.

4.1.5. Overview

We start by generating path expressions by path tracing as in Chapter 8, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\overline{A}$) as weights. Once we have predicate expressions that cover all paths, we can examine the logical sum of those expressions for consistency and ambiguity. We then consider a hierarchy of logic-based testing strategies and associated coverage concepts starting with simple branch testing and going on to explore strategies up to complete path testing.

4.2. Boolean Algebra

4.2.1. Notation

Let's take a structural viewpoint for the moment and review the steps taken to get the predicate expression of a path.

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 and the NO or FALSE branch with the same letter overscored.
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 10.2, 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 $AB\overline{C}$.

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

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.

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

The expression for node 6 can be rewritten as:

"Either it is true that decision A is satisfied (YES) *OR* it is true that decision A is not satisfied (NO) *AND* decision B is not satisfied *AND* decision C is not satisfied, OR both."

The "OR" in boolean algebra is always an **inclusive OR**, which means "A or B or both." The **exclusive OR**, which means "A or B, but *not* both" is $A\overline{B} + \overline{A}B$. Each letter in a boolean expression represents the truth or falsity of a statement such as:

"It is snowing outside."

"Decision A is satisfied."

"A mouse is green when it is spinning."

There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "always false." "Truth" and "falsity" should not be taken in the ordinary sense but in a more technical sense—such as meaning that a specific bit is set. Actually, it doesn't matter whether it is or is not snowing outside. If a program's decision is evaluated as corresponding to "it is snowing," then we say that the statement is true or that the value of the variable that represents it is 1.

4.2.2. The Rules of Boolean Algebra

Boolean algebra has three operators:

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	NO	YES	YES	NO	YES	YES	NO

- × meaning *AND*. Also called multiplication. A statement such as AB 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."
- $\bar{A}$ meaning *NOT*. Also **negation** or **complementation**. This is read as either "not A" or "A bar." The entire expression under the bar is negated. For example, $\bar{A}$ is a statement that is true only when statement A is false. The statement $\overline{(A + B)}$ is translated as, "It is not true that either A is true or B is not true or both."

We usually dispense with the clumsy phraseology of "it is true" or "it is false" and say "equals 1" or "equals 0" respectively. With these preambles, we can set down the laws of boolean algebra:

1. $A + A$ $\bar{A} + \bar{A}$	$= A$ $= \bar{A}$	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. AA $\bar{A}\bar{A}$	$= A$ $= \bar{A}$	
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. $A + \bar{A}B$	$= A + \bar{B}$	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**. The product of several literals is called a **product term** (e.g., ABC, DE). Usually, product terms are simplified by removing duplicate appearances of the same literal barred or unbarred. For example, AAB and $A\bar{B}C\bar{B}$ are replaced by AB and $A\bar{B}C$ respectively. Also, any product term that has both a barred and unbarred appearance of the same literal (e.g., $\bar{A}BAC$) is removed because it equals zero by rule 10. 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. Boolean expressions can be simplified by successive applications of rules 19 and 20. We'll discuss much easier ways to do this in Section 5 below. The result of such simplifications 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.

4.2.3. Examples

The path expressions of Section 4.2 can now be simplified by applying the rules. The steps are shown in detail to illustrate the use of the rules. Usually, it's done with far less work. It pays to practice.

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

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:

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

4.2.4. Paths and Domains

Consider a loop-free entry/exit path and assume for the moment that all predicates are simple. Each predicate on the path is denoted by a capital letter (either overscored or not). The result is a term that consists of the product of several literals. For example, $\overline{A}BC$. If a literal appears twice in a product term then not only can one appearance be removed but it also means that the decision is redundant. If a literal appears both barred and unbarred in a product term, then by rule 10 the term is equal to zero, which is to

say that the path is unachievable.

A product term on an entry/exit path specifies a domain because each of the underlying predicate expressions specifies a domain boundary over the input space. Now let's allow the predicates to be compound and again trace a path from entry to exit. Because the predicates are compound, the boolean expression corresponding to the path will be (after simplification) a sum of product terms such as ABC + DEF + GH. Because this expression was derived from one path, the expression also specifies a domain. However, the domain now need not be simply connected. For example, each of the product terms ABC, DEF, and GH could correspond to three separate, disconnected subdomains. If any one of the product terms is included in another, as in ABC + AB, then it means that the domain corresponding to ABC is wholly contained within the domain corresponding to AB and it is always possible to eliminate the included subdomain (ABC in this case) by boolean algebra simplification. Moreover, if the product of any two terms is not zero, then the two domains overlap even though one may not be contained in the other.

An alternative design could have eliminated the compound predicates by providing a separate path for each product term. For example, we can implement ABC + DEF + GH as one path using a compound predicate or as three separate paths (ABC, DEF, and GH) that specify three separate domains that happen to call the same processing subroutine to calculate the outcomes.

Let's say that we've rewritten our program, design, or specification such that there is one and only one product term for each domain: call these D_1, D_2, . . . D_i, . . . D_m. Consider any two of these product terms, D_i and D_j. For every i not equal to j, D_iD_j must equal zero. If the product doesn't equal zero, then there's an overlap of the domains, which is to say a contradictory domain specification. Furthermore, the sum of all the D_i must equal 1 or else there is an ambiguity. Actually, the same will hold even if the D_i are not simple product terms but arbitrary boolean expressions that could result from compound predicates—i.e., domains that are not simply connected.

4.2.5. Test Case Design

It is, in principle, necessary to design a test case for each possible TRUE/ FALSE combination of the predicates. In general, as in the example, the predicates are correlated, so not all paths are achievable. If the predicates were all uncorrelated, then each of the 2^n combinations would correspond to a different path, a different domain, and their sum would correspond to a minimum-covering set.

Although it is possible to have ambiguities and contradictions in a specification (given, say, as a list of conditions), it is *not* possible for a program to have contradictions or ambiguities if:

1. The routine has a single entry and a single exit.
2. No combination of predicate values leads to nonterminating loops.
3. There are no pieces of dangling code that lead nowhere.

Under these circumstances, the boolean algebra expression corresponding to the set of all entry/exit paths must equal I exactly. If it doesn't, either you've made a mistake in evaluating it or there are pieces of unreachable code or nonterminating code for some predicate-value combination.

Let's consider a hierarchy of test cases for an arbitrary loop-free routine that has compound predicates. The routine has one entry and exit and has no dead-end code. Because the predicates may be compound, the boolean algebra expression of a domain will generally be a sum of products after simplification. We can build a hierarchy of test strategies by considering how we test for each domain and the whole

routine.

The most complete test case set you can use is one where the paths correspond to all 2^n combinations of predicate values, which we know is equivalent to all paths. There can be no more cases than this (from the point of view of logic-based testing), but testing can usually be achieved with fewer test cases. For example, branch coverage of Figure 10.2 can be achieved by using all the cases except ABC and $A\bar{B}\bar{C}$: and there are several other combinations that do not include all 2^n cases but still provide branch coverage. To find a set of covering paths, write out all 2^n combinations. Each combination specifies a path. Add combinations until branch coverage has been achieved.

Typically, we're interested in the boolean algebra expression corresponding to paths from one node to another. For example, from the entry to the point at which processing specific to a domain is done. We had three other points of interest in this routine, corresponding to the three processes. Those nodes were N6, N8, and N12. The simplified product-form boolean algebra expression for those nodes specifies the test cases needed for each. Any one prime implicant in the boolean expression covering all the paths from the entry to the node is sufficient to reach the node. The expansion of the expression for that node specifies all possible ways of reaching that node, although all terms may not be needed to provide coverage to that point. For example:

$$\begin{aligned} N6 &= A + \bar{B}\bar{C} \\ &= A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + ABC + \bar{A}\bar{B}\bar{C} \end{aligned}$$

Any term starting with A will get us to node 6, and it doesn't matter what happens subsequently. The only other way to get to node 6 is via $\bar{A}\bar{B}\bar{C}$. This is exactly what the simplified version of the expression for N6 said. A gets you there and the other predicates are immaterial. Consequently, the $A\bar{B}\bar{C}$ obtained from the expansion of $\bar{B}\bar{C}$ is also immaterial, and only the $\bar{A}\bar{B}\bar{C}$ term remains. N8 is reached by B, no matter what values the other predicates may have. In particular, $\bar{A}\bar{B}$ and AB provide all the ways to get to node 8. Node 12 can be reached by C, but all four terms in C's expansion are needed to get there in all possible ways.

Note that it does not matter whether the predicates at a decision are simple or compound. Say that a predicate at a node is X + YZ. We can replace it by a new variable, say U = X + YZ. The analysis is the same as before, and when we're through, we can replace every instance of U with the equivalent expression in X, Y, and Z. Why bother when we can work directly with the compound predicate? So although we started with structure-based path predicate expressions and notions of branch coverage, we end up with notions of predicate coverage. The set of paths used to reach any point of interest in the flowgraph (such as the exit) can be characterized by an increasingly more thorough set of test cases:

1. *Simplest*—Use any prime implicant in the expression to the point of interest as a basis for a path. The only values that must be taken are those that appear in the prime implicant. All predicates not appearing in the prime implicant chosen can be set arbitrarily. If we chose the $\bar{B}\bar{C}$ prime implicant for node 6, we could still get to node 6, whether we chose A or $\bar{A}$. If we picked the A prime implicant, it doesn't matter what we do about B or C. Note that this still leaves us exposed to problems of compound predicates discussed in Chapter 3, Section 3.4.
2. *Prime Implicant Cover*—Pick input values so that there is at least one path for each prime implicant at the node of interest.
3. *All Terms*—Test all expanded terms for that node—for example, five terms for node 6, four for node 8, and four for node 12. That is, at least one path for each term.
4. *Path Dependence*—Because in general, the truth value of a predicate is obtained by

interpreting the predicate, its value may depend on the path taken to get there. Do every term by every path to that term.

The exit is treated as any other point of interest except that it should have a value of 1 for its boolean expression, which when expanded yields all 2^n combinations of predicate values. A set of covering paths could be achieved with fewer than 2^n test cases. K. C. Tai (TAIK87, TAIK89) has investigated strategies that can, with some restrictions, test for both relational operator errors and boolean expression errors using of the order of n tests for n predicates rather than 2^n. The pragmatic utility of these strategies and other logic-based test strategies is currently unknown because we do not know how frequently the kinds of bugs these strategies expose occur in actual software.

4.3. Boolean Equations

Loops complicate things because we may have to solve a boolean equation to determine what predicate-value combinations lead to where. Furthermore, the boolean expression for the end point does not necessarily equal 1. Consider the flowgraph of Figure 10.3.

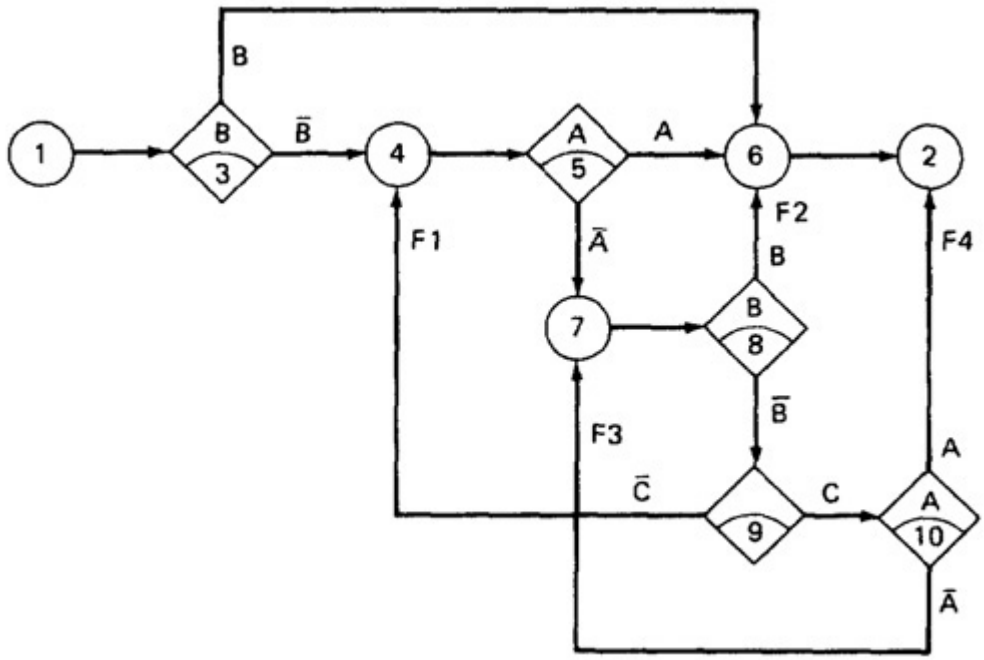


Figure 10.3. A Flowgraph with Loops.

Assign a name to any node whose value you don't know yet and write the expressions down, one at a time, working backward to something you do know, simplifying as you go. It's usually convenient to give names to links. The names represent the boolean expression corresponding to that link. I've named the links F1, F2, F3, and F4.

$$
\begin{aligned}
N4 &= \overline{B} + F1 \\
F1 &= \overline{B}\,\overline{C}N7 \\
N4 &= \overline{B} + \overline{B}\,\overline{C}N7 \\
&= \overline{B} \\
N6 &= B + AN4 \\
&= B + \overline{A}B \\
&= A + B \\
N7 &= \overline{A}N4 + F3 \\
&= \overline{A}\,\overline{B} + F3 \\
F3 &= N7\overline{B}C\overline{A} \\
N7 &= \overline{A}\,\overline{B} + \overline{A}\,\overline{B}CN7 \\
&= \overline{A}\,\overline{B} \\
N2 &= N6 + F4 \\
&= A + B + F4 \\
F4 &= A\overline{B}CN7 \\
N2 &= A + B + A\overline{B}CN7 \\
&= A + B
\end{aligned}
$$

You might argue that this is a silly flowgraph, but it illustrates some interesting points. The fact that the expression for the end point does not reduce to 1 means that there are predicate-value combinations for which the routine will loop indefinitely. Because the flowgraph's exit expression is A + B, the condition under which it does not exit is the negation of this or $\overline{(A + B)}$, which by De Morgan's theorem (rule 14) equals $\overline{A}\,\overline{B}$ This term when expanded yields $\overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C}$, which identifies the two ways of looping, via nodes 7,8,9, 10 and 4,5,7,8,9, respectively. It is conceivable that this unstructured horror could have been deliberately constructed (other than as a tutorial example, that is), but it's not likely. If the predicate values are independent of the processing, this routine must loop indefinitely for $\overline{A}\,\overline{B}$. A test consisting of all eight predicate-value combinations would have revealed the loops. Alternatively, the fact that the exit expression did not equal 1 implied that there had to be a loop. Feeding the logic back into itself this way, usually in the interest of saving some code or some work, leads to simultaneous boolean equations, which are rarely as easy to solve as the given example; it may also lead to dead paths and infinite loops.

5. KV CHARTS

5.1. The Problem

It's okay to slug through boolean algebra expressions to determine which cases are interesting and which combination of predicate values should be used to reach which node; it's okay, but not necessary. 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're testing. The **Karnaugh-Veitch chart** (this is known by every combination of "Karnaugh" and/or "Veitch" with any one of "map," "chart," or "diagram") reduces boolean algebraic manipulations to graphical trivia (KARN53, VEIT52). Beyond six variables these diagrams get cumbersome, and other techniques such as the Quine-McCluskey (MCCL56, QUIN55) method (which are beyond the scope of this book) should be used.

5.2. Simple Forms

Figure 10.4 shows all the boolean functions of a single variable and their equivalent representation as a KV chart. The charts show all possible truth values that the variable A can have. The heading above each box in the chart denotes this fact. 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.

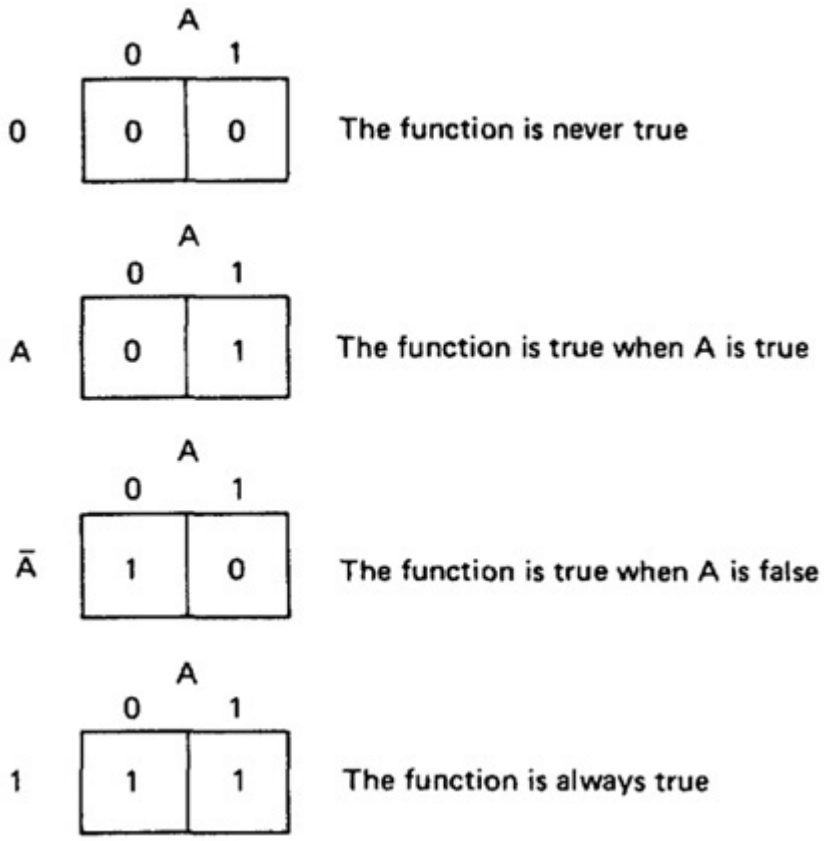


Figure 10.4. KV Charts for Functions of a Single Variable.

Figure 10.5 shows eight of the sixteen possible functions of two variables. Each box corresponds to the combination of values of the variables for the row and column of that box. The single entry for $\bar{A}\bar{B}$ in the first chart is interpreted that way because both the A and B variables' value for the box is 0. Similarly, $A\bar{B}$ corresponds to A = 1 and B = 0, $\bar{A}B$ to A = 0 and B = 1, and AB to A = 1 and B = 1. The next four functions all have two nonzero entries, and each entry forms an adjacent pair. 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. Similarly, in the sixth chart it is the A variable that changes in the B = 1 row, and consequently the chart is equivalent to B. Similarly for $\bar{A}$ and $\bar{B}$.

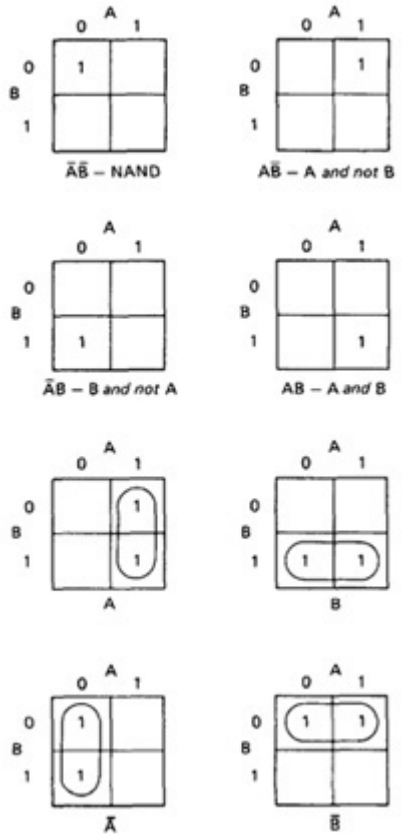


Figure 10.5. Functions of Two Variables.

Figure 10.6 shows the remaining eight functions of two variables. The interpretation of these charts is a combination of the interpretations of the previous charts in Figure 10.5. 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. Because the first is $\bar{A}\bar{B}$ and the second AB, the entire chart is equivalent to $\bar{A}\bar{B}$ + AB. Similarly, the second chart is equivalent to $\bar{A}B + A\bar{B}$. The next four charts have three 1's in them, and each can be grouped into adjacent groups of two (remember, adjacency is either horizontal or vertical).* Each adjacent group is a prime implicant and is therefore written down connected to the others by a "+". The first example consists of two adjacent groups of two boxes, corresponding to $\bar{A}$ (vertical group) and B (horizontal group), to yield $\bar{A}$ + B. The last case has all boxes filled with 1's and consequently, whatever the value of the variables might be, the function equals 1. The four entries in this case form an adjacent grouping of four boxes. 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).

*Overlapping and multiple use is allowed because A + AB = A.

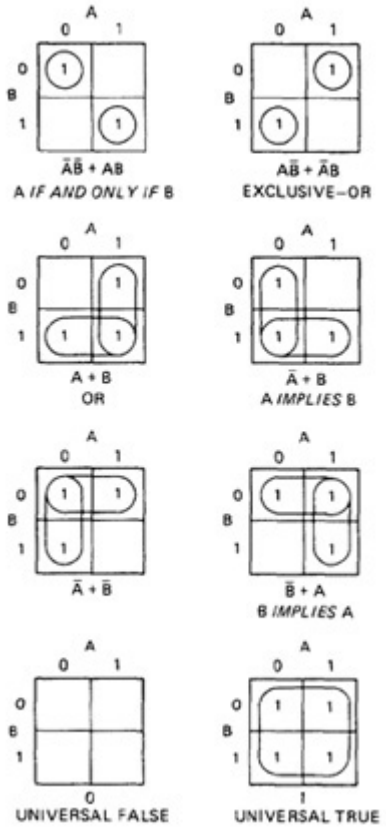
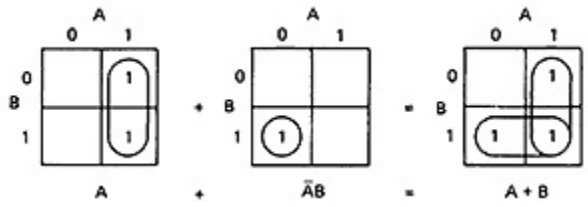
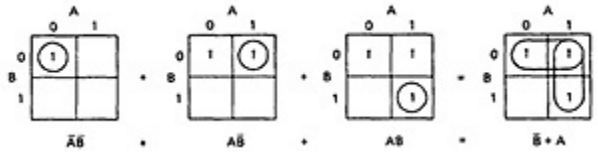


Figure 10.6. More Functions of Two Variables.

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. The third example of Figure 10.6 explains rule 19 on page 336. In fact, it's trivially obvious:

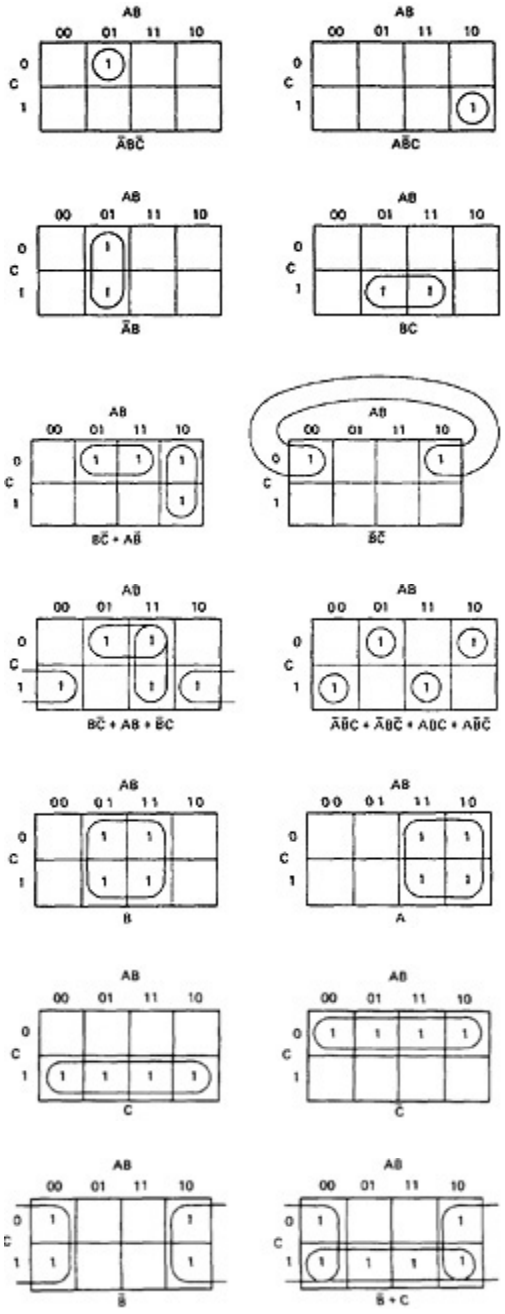


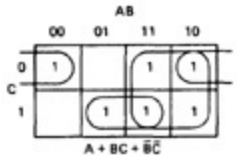
I've used the fact that KV charts are boolean functions. Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart. The procedure for simplifying expressions using KV charts is to fill in each term one at a time, and then to look for adjacencies and to rewrite the expression in terms of the largest groupings you can find that cover all the 1's in the chart. Say the expression is $\bar{A}\bar{B} + A\bar{B} + AB$, then

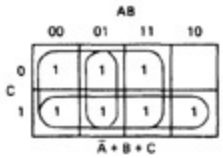


5.3. Three Variables

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. Note that I've labeled the column headings in an unusual way "00, 01, 11, 10" rather than with the expected "00, 01, 10, 11." Recall that the variable whose value did not change is the one we ended with. This labeling preserves the adjacency properties of the chart. However, note that adjacencies can go around corners, because 00 is adjacent to 10. The meaning of "adjacency" can now be specified more precisely: two boxes are **adjacent** if they change in only one bit, and two groupings are adjacent if they change in only one bit. A three-variable chart can have groupings of 1, 2, 4, and 8 boxes. A few examples will illustrate the principles:

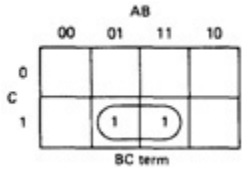


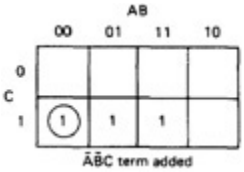


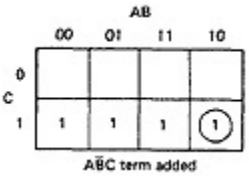


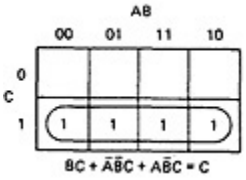
You'll notice that there are several ways to circle the boxes into maximum-sized covering groups. All such covering sets are equivalent, no matter how different they might appear to be. As an example, consider

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



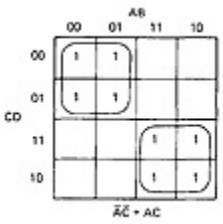


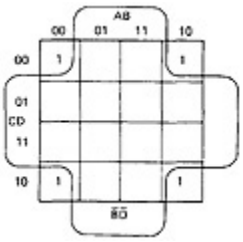


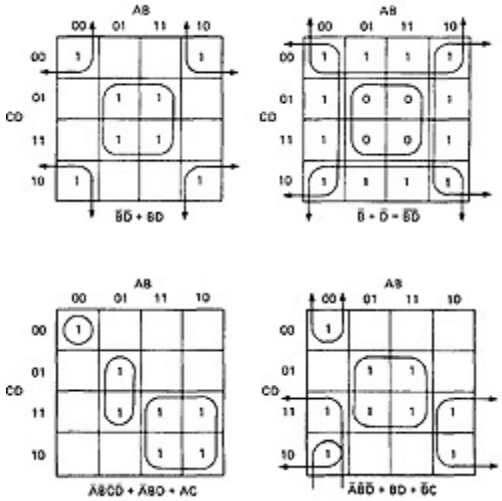


5.4. Four Variables and More

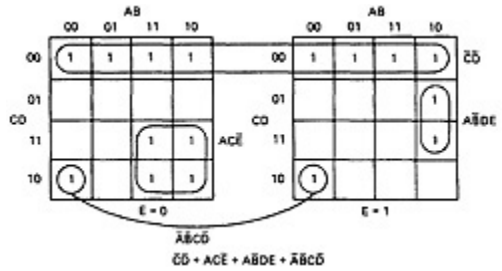
The same principles hold for four and more variables. A four-variable chart and several possible adjacencies are shown below. Adjacencies can now consist of 1, 2, 4, 8, and 16 boxes, and the terms resulting will have 4, 3, 2, 1, and 0 literals in them respectively:



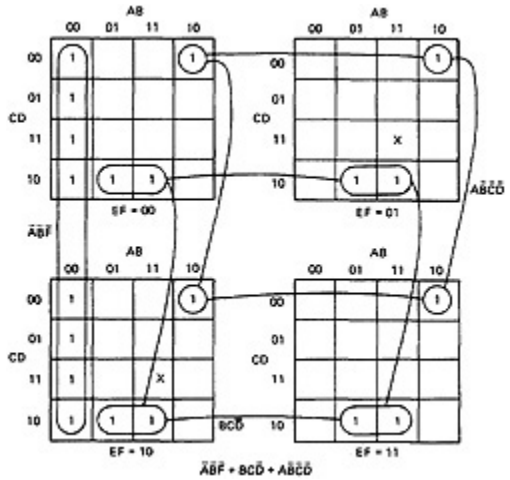




As with three-variable charts, the way you can group adjacent entries to cover all the Is in the chart is not unique, but all such ways are equivalent, even though the resulting boolean expressions may not look the same.



This is a five-variable chart with some of the adjacencies shown: things start to get cumbersome. For the hardy, there is a six-variable chart.



The two points labeled “X” are not adjacent because the subcharts on which they appear are diagonal to one another. If you really have to do a lot of work with six variables, you can build a three-dimensional tic-tac-toe game (4 × 4 × 4) out of transparent plastic. I once used eight transparent chessboards to do nine-variable problems—it was impressive but it didn’t work very well.

5.5. Even More Testing Strategies?

The KV map leads to yet another family of testing strategies to play with whose pragmatic utility is unknown. You developed a boolean algebra expression for every interesting action (e.g., domain). Consider any one of these and how you might go from the weakest to the strongest set of logic-based test cases:

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. Within this strategy there's a hierarchy also.
2. Use the prime implicant with the fewest variables (i.e., largest adjacency set or, equivalently, fewest domain boundaries) and work down to the prime implicant with the greatest number of variables (smallest adjacency set or most domain boundaries). For the five variable map above, the cases are: $\overline{C}D$, $AC\overline{E}$, $A\overline{B}D\overline{E}$ and $\overline{A}\overline{B}C\overline{D}$.
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. For example, $A = AB + A\overline{B}$.. This is equivalent to considering maximum-size adjacency groups, and working down to smaller groups. Ultimately all terms contain n literals, which for the routine as a whole is equivalent to testing 2^n cases.

6. SPECIFICATIONS

6.1. General

There's no point in getting into design and coding until you're sure that the specification is logically consistent and complete. This section shows you how much of specification analysis can be done using the KV chart, pencil, and paper. You'll need to use such methods until your automated specification system and language comes on stream—which could be a long wait. Alternatively, if you have a specification system, this section gives you an insight into some of the processing done by it. 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.

6.2. Finding and Translating the Logic

This is the most difficult part of the job, because it takes intelligence to disentangle intentions that are hidden by ambiguities inherent in English and by poor English usage. We cast the specifications into sentences of the following form:

"IF predicate THEN action."

The predicates are written using the AND, OR, and NOT boolean connectives. Therefore, the problem should be one of finding the keywords: IF, THEN, AND, OR, and NOT. Unfortunately we have to deal with the real world of specifications and specification writers, where clarity ranges from elusive, through esoteric, into incomprehensible. Here is a sample of phrases that have been or can be used (and abused) for the words we need:

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.

The above is *not* a list of recommended synonyms for specification writers because I've included many examples of bad usage. Several entries appear in more than one list—a source of danger. There are other dangerous phrases, such as "respectively," "similarly," "conversely," "and so forth," and "etc." More than one project's been sunk by an "etc." The main point, maybe the only point, of translating the specification into unambiguous English that uses IF, THEN, AND, OR, and NOT, is that this form is less likely to be misinterpreted.

Start rewriting the specification by getting rid of ambiguous terms, words, and phrases and expressing it all as a long list of IF . . . THEN statements. Then identify the actions and give them names such as A1, A2, A3, etc. Break the actions down into small units at first. All actions at this point should be mutually exclusive in the sense that no one action is part of another. If some actions always occur in conjunction with other actions and vice versa, then lump them into one action and give it a new name. Now substitute the action names in the sentences. Identify the "OR" components of all sentences and rewrite them so that each "OR" is on a separate line. You now have a specification of the form

IF A AND B AND C, THEN A1,
IF C AND D AND F, THEN A1,
IF A AND B AND D, THEN A2,
. . .

Now identify all the NOTs, which can be knotty because some sentences may have the form $\overline{(A + B + C)}$ or $\overline{A}\overline{B}\overline{C}$. Put phrases in parentheses if that helps to clarify things. The only English now remaining are the A's, B's and C's, which should resemble predicates of the form, "A is true" or "NOT A . . .". Identify all the predicates in both negated and unnegated form and group them. Select a single representative for each, preferably the clearest one, or rewrite the predicates if that helps. Give each predicate a letter. You now have a set of boolean expressions that can be retranslated back into English preparatory to confirmation. An alternative is a table. List the predicates on one side and the actions on the other—a decision table is a handy format—and use that instead of English sentences. It's helpful to expand the immaterial cases and show the 2^n combinations of predicate values explicitly. Immaterial cases are always confusing.

This process should be done as early as possible because the translation of the specification into boolean algebra may require discussion among the specifiers, especially if contradictions and ambiguities emerge. If the specification has been given as a decision table or in another equally unambiguous tabular form, then most of the above work has been avoided and so has much of the potential confusion and the bugs that inevitably result therefrom.

6.3. Ambiguities and Contradictions

Here is a specification:

$$\begin{aligned} A1 &= B\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D \\ A2 &= A\overline{C}\,\overline{D} + A\overline{C}D + A\overline{B}\,\overline{C} + AB\overline{C} \\ A3 &= BD + B\overline{C}\,\overline{D} \\ ELSE &= B\overline{C} + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \end{aligned}$$

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):

CD \ AB	00	01	11	10
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: $\overline{A}\,\overline{B}\,\overline{C}D$ is missing. The specification layout seems to suggest that this term also belongs to the default action. I would ask the question twice:

1. Is $\overline{A}\,\overline{B}\,\overline{C}D$ also to be considered a default action?
2. May the default action be rephrased as $\overline{B}\,\overline{C} + \overline{A}\,\overline{B}$?

You might get contradictory answers, in which case, you may have to rephrase your question or, better yet, lay out all the combinations in a table and ask for a resolution of the ambiguities. There are several boxes that call for more than one action. If the specification did not explicitly call out both actions in a sentence, such as, "IF $AB\overline{C}\,\overline{D}$ then *both* action 1 and action 2 shall be taken," I would treat each box that contained more than one action as a potential conflict. Similarly, if the specification did say, ". . . both A1 and A2 . . ." but did not mention A3, as in the $AB\overline{C}D$ entry, I would also question that entry.

If no explicit default action is specified, then fill out the KV chart with explicit entries for the explicitly specified actions, negate the entire chart, and present the equivalent expression as a statement of the default. In the above example, had no default action been given, all the blank spaces would have been replaced with 1's and the $\overline{B}\overline{C} + \overline{A}\,\overline{B}$ expression would have resulted.

Be suspicious of almost complete groups of adjacencies. For example, if a term contains seven adjacent boxes and lacks only one to make a full eight adjacency, question the missing box. I would question 3 out of 4, 6 or 7 out of 8, 13 through 15 out of 16, and so on, especially if the missing boxes are not themselves adjacent.

It's also useful to present the new version of the specification as a table that shows all cases explicitly and also as a compact version in which you've taken advantage of the possibility of simplifying the expression by using a KV chart. You present the table and say that, "This table can also be expressed as. . . ." There may be disagreement. The specifier might insist that the table does not correspond to the specification and that the table also does not correspond to your compact statement, which you know was derived from the table by using boolean algebra. Don't be smug if that happens. Just as often as the seeming contradiction will be due to not understanding the equivalence, it will be due to a predicate that has not been explicitly stated.

6.4. Don't-Care and Impossible Terms

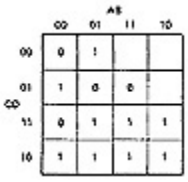
There are only three things in this universe that I'm certain are impossible:

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.

Everything else is improbable, but not impossible. So-called "impossible" cases can be used to advantage to simplify logic and, consequently, to simplify the programs that implement that logic. There are two kinds of so-called impossible conditions: (1) the condition cannot be created or is seemingly contradictory or improbable; and (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 the binary world has "created" four impossible cases. The external world can also contain "impossible" and "mutually exclusive" cases, such as female fighter pilots, consistent specifications, honest mechanics and politicians, insincere used-car salesmen and funeral directors, and blind editors. The seemingly impossible cases of semantic origin can appear to occur within a program because of malfunctions or alpha particles. The supposed impossibilities of the real world can come into being because the world changes. Consequently, you can take advantage of an "impossible" case only when you are sure that there is data validation or protection in a preceding module or when appropriate illogical condition checks are made elsewhere in the program. Taking advantage of "impossible" and "illogical" cases is a dangerous practice and should be avoided, but if you insist on doing that sort of thing, you might as well do it right:

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:



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

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

By taking advantage of the impossible conditions, we get:

$$C + \overline{A}$$

The corresponding flowgraphs are shown in Figure 10.7. The B and D decisions have disappeared for the second case. This a two-edged sword.

By reducing the logic's complexity we reduced instructions, data references for B and D, and the routine's complexity, and thereby reduced the probability of bugs. However, the routine now depends on nature's good graces, how thoroughly preceding routines have done data validation, and how thoroughly data validation will be done after this design has been modified in maintenance. It is not obvious whether long-term quality has been improved or degraded.

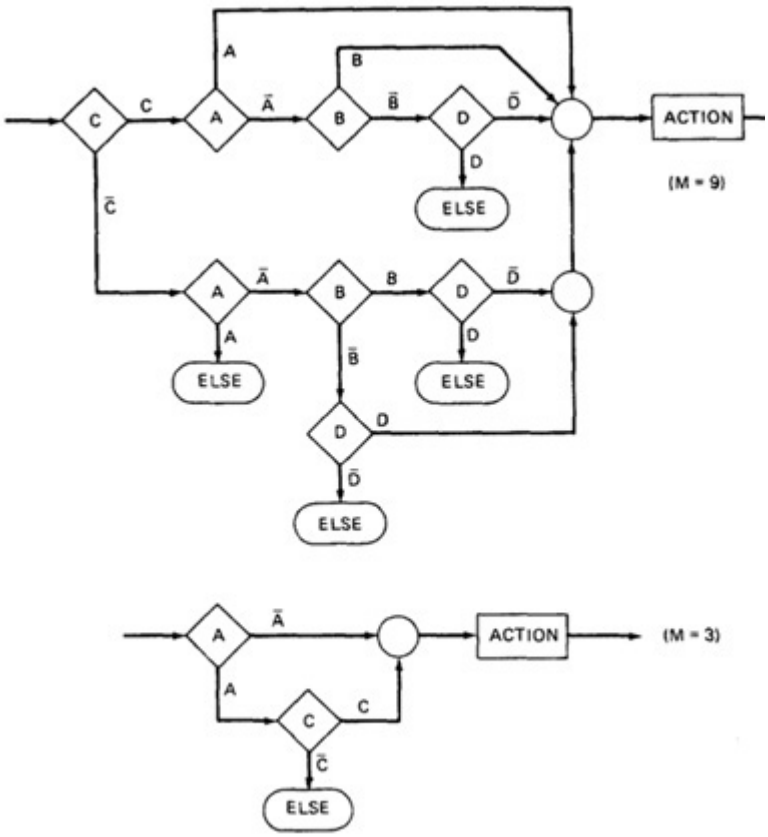


Figure 10.7. Reducing Complexity by Simplifying the Logic.

7. TESTABILITY TIPS

Logic-intensive software designed by the seat of the pants is almost never right. We learned this lesson decades ago in the simpler hardware logic design arena. It is in our interest as software engineers to use the simplest possible predicate expressions in our design. The objective is not to simplify the code in order to save a few bytes of memory but to reduce the opportunities for bugs. Hardware logic designers learned that there were many advantages to designing their logic in a **canonical form**—that is, a form that followed certain rules. The testability considerations of this chapter apply to loop-free software, or to the portion of the software that is loop-free; for example, a logic-intensive program segment within a loop can be examined by these means. You can start either from specifications or, if you're doing a redesign, from code. I'll speak to the latter case because it's more general. Think in terms of redesign if you have sensitization difficulties.

1. Identify your predicates (simple or compound).
2. If starting from code, get a branch covering set of path predicates.
3. Interpret the predicates so that they are expressed in terms of the input vector for the chosen path.
4. Simplify the path predicate expression for each selected path. If any expression is logically zero, the path is unachievable. Pick another path or paths to achieve branch coverage.
5. If any path predicate expression equals logical I then all other paths must be unachievable—find and fix the design bug.
6. The logical sum of the path predicate expressions must equal 1 or else there is an unsuspected loop, dangling code, or branch coverage is an inadequate test criterion.

The canonical processor has three successive stages:

1. Predicate calculator.
2. Logic analyzer.
3. Domain processor.

The **predicate calculator** transforms (e.g., processes) the input vector to get the values of the variables that are actually used in the predicates. Every predicate is evaluated exactly once, so that it's truth value is known. The **logic analyzer** forms the predicate expression appropriate to the case and directs the control flow to the appropriate domain processor. Because each predicate defines a domain boundary and each predicate expression defines a domain, there is a one-to-one correspondence between the various outcomes of the logic analyzer and the domains. The **domain processor** does the processing appropriate to each domain, for example, with a separate hunk of code for each domain. Only one control-flow statement (a case statement) is needed—one case, one predicate expression, one domain. The canonical form, if it is achieved, has the following obvious advantages:

1. Branch coverage and all-paths coverage are identical.
2. All paths are achievable and easy to sensitize.
3. Separation simplifies maintenance.

The above canonical form is an ideal that you cannot expect to achieve. Achieving it could mean redundant software, excessive nesting depth (if you encapsulate the redundancies in subroutines), or slow execution on some paths; conversely, however, the canonical form can be faster and tighter. You may be able to achieve it locally, or globally but not both; but you don't know unless you try. And why try? Because it works. The proof comes from hardware design, where we learned, three decades ago, that seat-of-the-pants logic was buggy, slow, dangerous, and hard to build, test, and maintain.

8. SUMMARY

1. Use decision tables as a convenient way to organize statements in a specification—possibly as an intermediate step toward a more compact and more revealing equivalent boolean algebra expression.
2. Label the links following binary decisions with a weight that corresponds to the predicate's logical value, and evaluate the boolean expressions to the nodes of interest.
3. Simplify the resulting expressions or solve equations and then simplify if you cannot directly express the boolean function for the node in terms of the path predicate values.
4. The boolean expression for the exit node should equal 1. If it does not, or if attempting to solve for it leads to a loop of equations, then there are conditions under which the routine will loop indefinitely. The negation of the exit expression specifies all the combinations of predicate values that will lead to the loop or loops.
5. Any node of interest can be reached by a test case derived from the expansion of any prime implicant in the boolean expression for that node.
6. The set of all paths from the entry to a node can be obtained by expanding all the prime implicants of the boolean expression that corresponds to that node. A branch-covering set of paths, however, may not require all the terms of the expansion.
7. You don't do boolean algebra by algebra. You use KV charts for up to six variables. Keep quadrille-ruled paper pads handy.
8. For logic-intensive routines, examine specification completeness and consistency by using boolean algebra via KV charts. Use the canonical form as a model of clean logic.
9. Be careful in translating English into boolean algebra. Retranslate and discuss the retranslation of the algebra with the specifier. Be tricky and use alternate, logically equivalent forms to see whether they (specifiers) are consistent and whether they really want what they say they want.
10. Question all missing entries, question overlapped entries if there was no explicit statement of multiple actions, question all almost-complete groups.
11. Don't take advantage of don't-care cases or impossible cases unless you're willing to pay the maintenance penalties; but if you must, get the maximum payoff by making the resulting logic as simple as you can and document all instances in which you take advantage of don't-care cases.s

<ch9 toc ch11>