Translation of Decision Tables

UDO W. POOCH

Assistant Professor, Industrial Engineering Department, Texas A & M University

Decomposition and conversion algorithms for translating decision tables are surveyed and contrasted under two broad categories: the mask rule technique and the network technique. Also, decision table structure is briefly covered, including checks for redundancy, contradiction, and completeness; decision table notation and terminology; and decision table types and applications. Extensive literature citations are provided.

Keywords and Phrases: decision tables, systems analysis, diagnostic aids, business applications.

CR categories: 3 50, 3 59, 4.19, 4 29, 4.49, 8.3.

I. INTRODUCTION

The use of decision tables by programmers, analysts, and other users of computer facilities is increasing because they provide a simple tabular representation of complex decision logic. Decision tables, although developed primarily as a vehicle for man-to-man communications, can ease the problems of programming and documentation in many applications where the feasibility of using the traditional flowchart, narrative description, or other communications media is questionable [10, 23, 30, 31, 64, 66, 70, 74, 75, 84, 88, 99, 101].

As higher level programming languages, such as COBOL, FORTRAN, ALGOL, and others, became widely accepted, the communication gap between the computer specialists and the users of computer facilities was expected to disappear. However, this has not been the case, so there continues to be a high degree of misunderstanding in systems analysis and design, and in implementing the chosen procedure into a workable computer program. This is especially true of management-to-man communication, specifically because management frequently does not understand this form of programming language communication. A language form, or structure, is therefore needed to bridge these man-to-man and man-to-machine communication gaps in these areas. Decision tables can contribute much to improve this communication link (Fergus [29]).

Decision tables provide an effective means of communication between those in and outside the data processing field by defining both the problems and their corresponding logical solutions. In addition, because decision tables succinctly display any conditions that must be satisfied before any prescribed actions will be performed, they are becoming very popular in computer programming and system design as devices for organizing logic, especially when attempting to handle very complex situations, and to be able to account for every possible combination of conditions [23, 32, 57, 62, 66, 89, 92, 104]. Furthermore, the extent and nature of the changes required to update or revise an application program is easily provided by the unique form of the problem statement in decision tables (Auerbach [3]).

Flowcharts, a graphic language form that has also been widely used for man-to-man communications, that was specifically developed for the purpose of representing operations related to computer activities, such

CONTENTS

as system analysis, system design, programming, documentation, etc., can also frequently be utilized for other noncomputer-related activities (Chapin [14]).

Comparison of Decision Tables and Flowcharts

The decision table is a convenient form for expressing any conditional alternatives, where a particular path to be followed is dictated by a combination of a number of conditions. Flowcharts in such cases can become very complex and difficult to follow, and involve testing for each condition more than once [105].

Decision tables overcome many of the disadvantages of flowcharts as a means of describing computer logic. As may be seen from Tables 1 and 2, decision tables are generally more suitable for direct communication with the computer, and are usually less confusing in the more complex situations, especially if we consider that a decision table contains every possible flowchart which can be drawn for any given problem. Decision tables afford precisely stated logic, more explicit relationships between variables, and simplification of programming (Klick [62]). Thus they provide a convenient way for the analyst or programmer to account for every possible combination of conditions.

It should be noted that the relative merits of decision tables must be weighed against the relative merits of well-structured flowcharts. In other words, with the developing "technology" of structured programming and methods for correctness proof methodologies, the utility of decision tables must be compared with that of a more modern version of programming via flowcharts, rather than with the less disciplined form that was in evidence, especially in nonscientific programming shops, until to very recently. Decisions in a flowchart must be tested in the order in which they appear; however in a decision table (except for the ELSE rule and any specific ordered decompositions, such as the left-to-right decomposition (Harrison [42]) the decision can be tested in any order, depending upon the particular algorithm used in translating the decision table. This enables programmers or

TABLE 1. ADVANTAGES/DISADVANTAGES OF FLOWCHARTS

Advantages	*Disadvantages*
• Easily produced. • Easily learned (few relatively simple rules and component parts). • Can be used unambiguously to describe the way computers handle data, as well as to represent operations performed by the computer • Can be produced by computer algorithms from source programs	• Heavily influenced by personal preference and jargon. • Difficult to follow if the problem conditions are complex. • Revision is difficult. • Limited in displaying all logical elements of the total problem • Difficult to ascertain if all logical elements are defined and analyzed, especially if the problem conditions are complex. • Flowcharts sharing detailed decision logic are unwieldy, resulting in "macrolizing" difficult sections.

TABLE 2 ADVANTAGES/DISADVANTAGES OF DECISION TABLES

Advantages	*Disadvantages*
• Clear enumeration of all operations performed • Clear identification of the sequence of operations • Easily learned. • Effective means of communication between people in and out of the data processing field; i.e., not limited to computer applications. • Concise and compact form of definition and description suitable for use in analysis, programming, and documentation. • Easy to construct, modify, and read. • Can be used to document applications involving complex interactions of variables. • When applied to computer systems, decision tables foster better use of subroutines, promote efficiency of computer runtime, and provide a complete data check for debugging. • Directly adapted (and possibly converted directly) to computer operations through symbolic logic and computer programs • Compared with narratives, decision tables are more concise and precise. • Easier visualization of relationships and alternatives.	• For complex situations, they may become extremely large. • Multiple tables may be needed in certain cases to document decision logic (Dixon [23], Fergus [28]) • Many people find the graphic display of flowcharts more meaningful than a tabular description of logic. • Desire for automatic translation ability causes too detailed requirements for man-to-man communication purposes (analogous to the use of programming languages and their restrictions).

analysts to consider the relative frequency with which transactions satisfy decision rules, and should lead to more efficient programs (Reinwald, et al. [93]). Therefore, although decision tables are not the answer to all documentation and programming problems, they do offer certain advantages that overcome some of the drawbacks of the flowchart technique [2, 55, 62, 73, 85, 87, 89, 92, 94, 98]. With the state-of-the-art advancing sufficiently to enable economic conversion of decision tables, their use will show a marked increase.

In Section II a broad spectrum of ideas, *including topics on the structure of decision* tables, and the varieties and formats of decision tables, are presented. Section III is devoted to the analysis of several different algorithms that can be used for converting decision tables into computer programs. A discussion of the advantages, disadvantages, and ambiguities of these algorithms is given. Finally, it should be pointed out that Shaw [96] and Denolf [20] present extensive, annotated bibliographies on decision tables, and that a recent issue of the SIGPLAN Notices (Shaw [97]) is dedicated completely to various aspects of decision tables.

II. DECISION TABLE STRUCTURE

A decision table provides a tabular representation of information and data. Information displayed in this manner is easily comprehended by eye, even if the table of information represents a complex logical problem. A decision table is a structure for describing a set of decision rules [4, 9, 13, 28, 46, 47, 49, 57, 68, 72, 103]. The basic structure of a decision table is universally accepted as that illustrated in Figure 1. Although other formats of decision tables exist, some of which are more convenient to certain input/output devices (Pollack, et al. [88]), they are all permutations of this basic format. Decision tables are easy to learn because of their simple structure; and efficiency in using them can be reached with little experience.

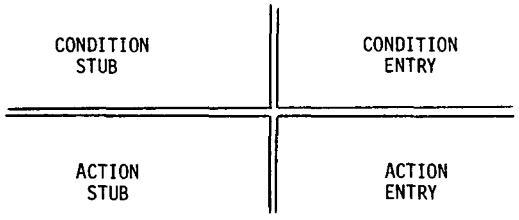
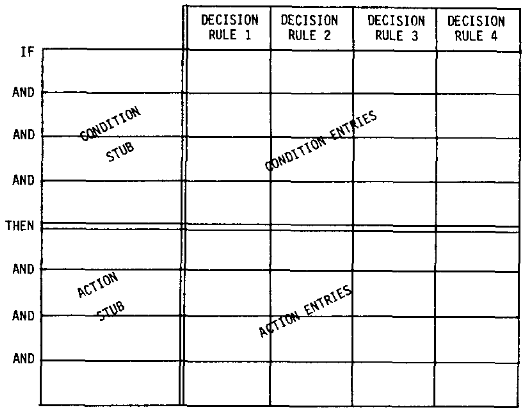


FIG. 1. Decision table structure.

TABLE 3. BASIC ELEMENTS OF A DECISION TABLE



A decision table can be divided into four quadrants [33, 38, 69, 81, 83, 106, 107]. The upper left quadrant, called the *condition stub,* should contain all those conditions being examined for a particular problem segment. The *condition entry* is the upper right quadrant. These two sections describe the set, or string, of conditions that is to be tested.

The lower left quadrant, called the *action stub,* contains a simple narrative format for all possible actions resulting from the conditions listed above the horizontal line *Action entries* are given in the lower right quadrant. Appropriate actions resulting from the various combinations of responses to the conditions will be indicated in the action entry. An example of decision rules and the IF-THEN function are illustrated in Table 3.

By considering Table 3, the meaning of the different sections can be illustrated. Each decision rule is a combination of responses to conditions in the condition entry quadrant. The decision rules are numbered for identification purposes in the rule header portion of the table. The topmost horizontal line represents IF, while the remaining horizontal lines represent AND, and the double horizontal line THEN. Note that the condition half of the table is separated from the action half by a double horizontal line and the stub sections are separated from the entry sections by a double vertical line. These lines improve the readability of a table, and can be preprinted on forms. Furthermore, many decision table processors permit ordering on the action entries, thereby making the explicit "AND" of questionable value.

In addition to the decision table elements already discussed, each table usually has a table header. The table header is used for identification purposes when the decision table is processed by the computer. The information that might be found in a table header includes an associated table number, a table name, the type of the table, the number of decision rules, the number of conditions, the number of actions, and various options available to maintain flexibility in formats.

If a condition in the condition stub is true, a Y is entered for that particular rule in the condition entry; if the condition is false, an N would be entered. In a situation where a particular condition is irrelevant, a don't-care would be indicated by use of a dash (–) or an I.

Two other entries, the * and $, are used to indicate mutual exclusion of one condition with another on a rule by rule basis (these symbols have been formulated by Pollack, et al. [88] and King [59]). Whenever the case arises within a single rule that the satisfaction of some "required" test (Y or N entry) makes some other required entry a foregone conclusion, then the special entries * (in place of N) or $ (in place of Y) can be used to indicate this fact. As an illustration of these inter-condition dependencies, consider the example given in Table 4 (Harrison [42]). Here, 'VALUE' must equal 1 for Rule R1 to be satisfied, at the same time, it may not equal to 3 nor greater than 2 This implies once 'VALUE' = 1 has been determined, the * will eliminate any further checks on the other two conditions; i.e. they can only be false. In other words, the * condition is required to be false for that rule, and some other condition for that same rule is adequate to satisfy the requirement. Rule R2, on the other hand, requires that 'VALUE' = 3, and therefore is certainly greater than 2, as indicated by the $. Thus, the $ indicates that a condition is required to be true, with some other condition available to insure satisfaction of that requirement. A more complete explanation of these entries can be found in Pollack, et al. [88], while an implication of these entries for completeness checking is given in Harrison [42].

TABLE 4. Example of * and $ Entries

STUBS	RULE	ENTRIES
'VALUE' = 1	Y	*
'VALUE' = 3	*	Y
'VALUE' > 2	*	$

Varieties and Formats of Decision Tables

Three types of decision tables are in current use today. The *limited entry* table is the most popular and most often used (King [59]). *Extended entry* and *mixed entry* tables are useful in some cases, but because they can always be transformed into limited entry tables, most of this analysis will be concerned with limited entry tables. Examples of the three different types of decision tables are presented in Tables 5A, 5B, and 5C.

In the limited entry table the only allowable entries in the entry quadrants are Y (true), N (false), * (implicit N), $ (implicit Y), X (execute action), or I (don't-care), and blank. All of the conditions and actions must be placed in the stub quadrants. Each rule of the decision table should be unique, so logically it does not matter which rule is tested first. Some of the techniques for selecting which rule to test first will be discussed in the next section. Only one rule should be satisfied by a single set of conditions, and if more than one rule can be satisfied the table is said to be *ambiguous* (King [58]).

An extended entry table has part of the condition in the stub quadrant and the remainder of the condition in the entry quadrant. The analogous format applies to the action part of the table. For example, in Table 5B if credit limit is *satisfactory* and pay experience is *favorable,* then approve the order. By considering Table 5B, it can be

Types of Decision Tables

TABLE 5A LIMITED ENTRY TABLE

	1	2	3	4
CREDIT LIMIT IS SATISFACTORY	Y	N	N	N
PAY EXPERIENCE IS FAVORABLE	-	Y	N	N
SPECIAL CLEARANCE IS OBTAINED	-	-	Y	N
PERFORM APPROVE ORDER	X	X	X	
GO TO REJECT ORDER				X

TABLE 5B EXTENDED ENTRY TABLE

	1	2	3
CREDIT LIMIT	SATISFACTORY	UNSATISFACTORY	UNSATISFACTORY
PAY EXPERIENCE	-	FAVORABLE	UNFAVORABLE
SPECIAL CLEARANCE	-	-	NOT OBTAINED
ORDER	APPROVE	APPROVE	REJECT

TABLE 5C. MIXED ENTRY TABLE

	1	2	3
CREDIT LIMIT IS	SATISFACTORY	UNSATISFACTORY	UNSATISFACTORY
PAY EXPERIENCE	-	Y	N
SPECIAL CLEARANCE	-	-	N
PERFORM APPROVE ORDER	X	X	
GO TO REJECT ORDER			X

seen that only one action line is required, whereas in the limited entry table two action lines were required. In general, it can be said that the limited entry table compresses a table vertically, while the extended entry table compresses it horizontally (IBM Corp. [48, 49]).

The mixed entry table is a combination of limited entry rows and extended entry rows (see Table 5C). The PERFORM and GO TO statements in Tables 5A and 5C were not just arbitrarily selected. The PERFORM, as used above, has the same connotation as the PERFORM verb as used in COBOL; i.e., execution is temporarily transferred into a *closed table* (or a *subroutine*), and control is subsequently returned to the next sequential action of the rule. The GO TO verb is used to exit to an open table or subroutine; that is, no provision is made for control of execution to return to the initiating table (CODASYL [18]). When constructing a table, the GO TO statement should be the last executable action within a decision rule.

Decision Table Notation

The basic structure presented in the previous section is easy to learn and understand, yet a logical step-by-step analysis is required in the preparation of a complete, accurate decision table. One of the benefits of this tabular method of communication is its adaptability to systematic and analytical techniques for checking completeness, contradictions, and redundancies [8, 11, 25, 26, 43, 44, 52, 53, 88]. Before considering some of the analytical techniques, it is necessary to define some of the notation and terminology in common use.

One of the specific types of Boolean algebra functions is used as the basis for most decision tables. This function, the *AND function,* is considered to be the ordered set of Y, N, I, or blanks that appear in the condition entry boxes for a particular decision rule. The application of the OR function can also be made in decision tables, however this analysis will be limited to the AND function (Hirschhorn [45]) Considering Table 5a, the following AND functions are found:

the AND function of Rule 1 = YII
the AND function of Rule 2 = NYI
the AND function of Rule 3 = NNY
the AND function of Rule 4 = NNN

To determine whether or not a decision rule is satisfied, evaluate the AND function for that decision rule, and check that it equals the required transaction. For example, the AND function of Rule 3 (NNY) in Table 5a would be the selected decision rule if the transaction was to approve the order, provided special clearance was obtained, even though credit limit and pay experience was unsatisfactory.

Definitions

Two AND functions are considered to be *dependent* if a transaction exists that satisfies both AND functions. If, on the other hand, a transaction satisfies one, and only one, of the AND functions, that AND function is *independent*.

A *pure AND function* is one that contains no I (don't-cares) (Pollack, et al. [88]). The following is a pure AND function: N·N·Y (of Rule 3 in Table 56), where "·" is defined as the Boolean operator AND.

A decision rule is *simple* if it contains a pure AND function. For example, Rule R3 in Table 5C is simple since it contains the pure AND function N·N·Y. If an AND function contains one or more I's, it is considered to be a *mixed AND function.* For example the AND function of Rule 2 in Table 5C is N·Y·I; hence, Rule 2 is a complex decision rule. If all the decision rules of a decision table are simple, the table is defined as a *full table;* a *partial table* is a decision table that has some mixed decision rules.

Redundancy, Contradiction, and Completeness

Before discussing the problems of redundancy, contradiction, and completeness, it is necessary to outline two of the basic requirements for decision tables:

(1) Every decision rule must specify at least one action (weak condition).

(2) Each transaction must be able to satisfy one, and only one, set of conditions in a decision table. Although there are exceptions to this requirement, for the type of "conventional" tables (Pollack, et al. [88]) under consideration here, this requirement holds (strong condition).

In practice it is often convenient and intuitive to define *all* rules (i.e., no ELSE) implying some no action rules; however, in theory all of these no-action rules should go to the ELSE, therefore the need for Requirement (1). For example, consider the situation where the conditions in a decision rule are: if the customer requests a first-class ticket and a first-class seat is available. Without an action, such as "issue a first-class ticket," the above conditions are nonsensical. The second requirement, which is one of the underlying axioms for decision table theory, must be true for other decision table rules to be valid. Compliance with Requirement (2) will also help to insure completeness of decision tables and reduce contradictions and redundancies among decision rules.

Contradictions and redundancies are checked by examining the decision rules to be certain that between each pair of decision rules there exists at least one condition row with a Y, N pair for the two rules. If this Y, N pair does not exist, similar action entries indicate *redundancy,* and different action entries indicate *contradiction* (Pollack [80]). Examples of contradiction and redundancy are illustrated in Table 6.

Rules R1 and R2 of Table 6 are acceptable rules because neither is redundant nor contradictory. However, R2 contradicts R3 and R4 because they all have the same decision rule, yet different action entries. Redundancy exists between R3 and R4 because both have the same decision rule and the same action.

A quick visual check, comparing two decision rules at a time, can easily identify if any redundancy or contradiction exists. If two or more rules do not have at least one Y, N pair in any of the rows, and the actions specified are not identical, then a contradiction of logic exists. An easy way to make this check for redundancy and contradiction is to compare the AND functions of different decision rules. In the following examples the mixed AND functions are

TABLE 6 Example of Redundancy and Contradiction

	R1	R2	R3	R4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

broken down into pure AND functions to correct the redundancy problem in Table 7.

TABLE 7. CREDIT APPROVAL

	R1	R2	R3
CREDIT OK	Y	I	N
PAY EXPERIENCE FAVORABLE	I	Y	N
PERFORM APPROVE ORDER	X	X	
GO TO REJECT ORDER			X

Rule R1 of Table 7 breaks down as follows:

$$\underset{(R1)}{\left[\begin{array}{c} Y \\ I \\ \hline\hline X \end{array}\right]} \longrightarrow \left[\begin{array}{c} Y \\ Y \\ \hline\hline X \end{array}\right] \;\&\; \left[\begin{array}{c} Y \\ N \\ \hline\hline X \end{array}\right]$$

Rule R2 of Table 7 breaks down as follows:

$$\underset{(R2)}{\left[\begin{array}{c} I \\ Y \\ \hline\hline X \end{array}\right]} \longrightarrow \left[\begin{array}{c} Y \\ Y \\ \hline\hline X \end{array}\right] \;\&\; \left[\begin{array}{c} N \\ Y \\ \hline\hline X \end{array}\right]$$

The common AND function of Rules 1 and 2 is:

$$\left[\begin{array}{c} Y \\ Y \\ \hline\hline X \end{array}\right]$$

This AND function can be eliminated. The redundancy-free table is given in Table 8.

TABLE 8. REVISED CREDIT APPROVAL

	R1	R3
CREDIT OK	Y	N
PAY EXPERIENCE FAVORABLE	Y	N
PERFORM APPROVE ORDER	X	
GO TO REJECT ORDER		X

It has been shown how redundancy and contradiction can be checked in decision rules that have both pure and mixed AND functions. In Table 9 a summary of all the rules for contradiction and redundancy are presented for a pair of decision rules R1 and R2. The AND functions of R1 and R2 are represented by AF1 and AF2, respectively, and the actions are represented by A1 and A2, respectively (Pollack, et al. [88]).

Another problem that always arises is whether or not the decision table is complete, and if it is complete, is there any redundancy, or contradiction in the table. The first step in checking a decision table for completeness is to analyze the table to see if the table contains simple decision rules, complex decision rules (don't-cares), and if any ELSE decision rule is present.

Pollack, Hicks, and Harrison [88] have developed and proved that there exist exactly 2^n independent pure AND functions in a decision table, where n is the number of conditions found in the decision table. For example, in Table 10A, three ($n = 3$) conditions appear in the decision table, therefore $2^3 = 8$ possible simple decision rules must exist. All of the decision rules in Table 10A are simple decision rules because there are no "don't care" entries. Furthermore, the decision rules are independent because in any two decision rules, one of the functions contains a Y and the other, an N. Hence, it can now be stated that Table 10a is a complete decision table (Pollack, et al. [88]).

If a transaction in Table 10A is "a guest asking a bartender to mix him a certain type (base) of drink," several rules could be combined. The rules in Table 10B illustrate how Table 10A could be rewritten to contain both simple and complex decision rules. One way to test Table 10B for completeness would be to expand it into Table 10A; however, the following preferred method has been developed (Pollack, et al. [88]):

(a) Check that each decision rule contains at least one action.

(b) Check each pair of decision rules to see if they are independent. Do this by checking the AND functions of the decision rules to see that there is a Y in at least one position of the

TABLE 9. CONTRADICTION AND REDUNDANCY

	R1	R2	R3	R4	R5	R6	R7
AF1 AND AF2 ARE DEPENDENT	N	Y	Y	Y	Y	Y	Y
AF1 = AF2	I	Y	N	N	N	N	I
A1 = A2	I	Y	Y	Y	Y	Y	N
AF1 IS PURE	I	I	Y	Y	N	N	I
AF2 IS PURE	I	I	Y	N	Y	N	I
AF1 AND AF2 ARE VALID	X	X		X	X	X	
AF1 AND AF2 ARE NOT VALID							X
REDUNDANT RULE IS AF2		X			X		
REDUNDANT RULE IS AF1				X			
AF1 AND AF2 CONTAIN REDUNDANT RULES		X		X	X	X	
AF1 AND AF2 DO NOT CONTAIN REDUNDANT RULES	X						
AF1 AND AF2 ARE CONTRADICTORY							X
AF1 AND AF2 ARE NOT CONTRADICTORY	X	X		X	X	X	
THIS RULE IS IMPOSSIBLE			X				

A = ACTION.
AF = AND FUNCTION.

function and the other function contains a N.

(c) Show that the 4 rules in Table 10B can be expanded into 8 decision rules.

To accomplish (c), recall that each decision rule containing an AND function with an I in r positions is equivalent to 2^r simple decision rules (Pollack, et al. [88]). For example, in Table 10B.

R1 has $2^1 = 2$
R2 has $2^0 = 1$
R3 has $2^1 = 2$
R4 has $2^5 = 1$
R5 has $2^0 = 1$
R6 has $2^0 = 1$

TOTAL 8 SIMPLE DECISION RULES

Table 10B satisfies all of the three tests for completeness, therefore it is a complete decision table.

One way to insure completeness in any decision table is to incorporate the ELSE rule into the decision table. The ELSE rule, by definition, includes all rules not specifically given in the tables. Usually the ELSE rule is merely a convenient catchall rule placed at the extreme right of the table with a special symbol in the rule header that identifies it as such (McDaniel [69]). For decision tables that are converted into computer programs, ELSE rules reduce the amount of needed coding, thereby making the program more efficient. However, the ELSE rule should not be used in this way;

TABLE 10A EXPANDED BARTENDER

	1	2	3	4	5	6	7	8
BOURBON BASE	Y	Y	Y	N	Y	N	N	N
GIN BASE	Y	Y	N	Y	N	Y	N	N
VODKA BASE	Y	N	Y	Y	N	N	Y	N
OFFER VODKA OR GIN AND VERMOUTH (MARTINI)	X	X	X	X		X	X	
OFFER VODKA AND ORANGE JUICE (SCREWDRIVER)	X		X	X			X	
OFFER VODKA AND TOMATO JUICE (BLOODY MARY)	X		X	X			X	
OFFER BOURBON AND VERMOUTH (MANHATTAN)	X	X	X		X			
OFFER BOURBON AND WATER (OLD FASHION)	X	X	X		X			
OFFER GUEST A SOFT DRINK								X

TABLE 10B. BARTENDER

BARTENDER

	1	2	3	4	5	6
BOURBON BASE	Y	Y	N	Y	N	N
GIN BASE	I	Y	I	N	Y	N
VODKA BASE	Y	N	Y	N	N	N
OFFER VODKA OR GIN AND VERMOUTH (MARTINI)	X	X	X		X	
OFFER VODKA AND ORANGE JUICE (SCREWDRIVER)	X		X			
OFFER VODKA AND TOMATO JUICE (BLOODY MARY)	X		X			
OFFER BOURBON AND VERMOUTH (MANHATTAN)	X	X		X		
OFFER BOURBON AND WATER (OLD FASHION)	X	X		X		
OFFER GUEST A SOFT DRINK						X

it should be used only for those transactions that analysts say cannot possibly happen, and not as a catchall for the convenience of the practitioners. Even with the ELSE rule present, a decision table must still allow ever case to occur. In other words, a table must be checked for exactly how many simple rules are in the table, and that the total number of ELSE rules does not exceed 2^n. If the total does exceed 2^n, then a contra-

TABLE 11 THE ELSE RULE

	1	2	3	4	5	ELSE
A = 1	Y	N	I	I	N	
B = 2	I	Y	I	Y	N	
C = 3	I	Y	N	N	N	
D = 4	Y	I	N	Y	Y	
E = 5	N	I	I	Y	Y	
CALCULATE SUM	X	X				
CALCULATE DIFFERENCE			X			
CALCULATE PRODUCT				X	X	

NUMBER OF POSSI-, BLE SIMPLE RULES		$= 2^n = 2^5 = 32$
NUMBER OF SIMPLE RULES REPRE- SENTED		$= 2^r$ FOR EACH RULE
	R1	$= 2^2 = 4$
	R2	$= 2^2 = 4$
	R3	$= 2^3 = 8$
	R4	$= 2^1 = 2$
	R5	$= 2^0 = 1$
		19
NUMBER OF RULES REPRESENTED BY ELSE RULE		$= 32 - 19 = 13$

Conversion of Extended Entry to Limited Entry

diction or redundancy exists. If the total is less than 2^n, it may be readily determined how many simple rules are represented by the ELSE rule (Pollack, et al. [88]). An illustration of the number of decision rules that may be represented by an ELSE rule is provided by Table 11. Since there are five conditions, there are $2^5 = 32$ possible simple decision rules. The number of simple rules actually present in the table is $(2^2 + 2^2 + 2^3 + 2^1 + 2^0) = 19$. The difference, 13 $(32 - 19)$, is the number of rules represented by the ELSE rule.

In summary, checking for completeness in decision tables has been discussed with the following three possible constraints:

1) Simple decision rules only.
2) A decision table with both simple and complex decision rules.
3) Decision tables with an ELSE rule.

It should be kept in mind that even though a decision table is a full table, a check for redundancies and contradictions is still a requirement. Once a table has been found to contain no redundancies or contradictions, a check should be made to determine if it is complete.

Uses and Applications of Decision Tables

Decision tables are useful in many areas of application. This section will analyze some broad-based uses of decision tables, including one specific application.

In Simulation Models

The previously emphasized advantages of decision tables in handling complex logic makes them a definite aid in formulating the logical flow of simulation models The queueing structures involved in a model are governed by decision rules which can be easily described by decision table usage. In a model employing decision tables, the table structure is mainly used to determine whether a subprogram is to be executed at a particular time in the simulation. Decision tables also provide a diagnostic aid for the programmer, as well as improving the general communication between the programmer and his problem (Ludwig [66]).

In an Organization

Decision tables can be used at different levels and for different functions in an organization. Policies of top management may often be expressed tabularly. Tables may be applied to areas such as engineering, mathematics, personnel, and accounting. Furthermore, decision tables can be combined with a decision documentation plan such as that of Fergus [29]. Features of this plan are:

1) Use of tables throughout an organization to document all decision making that deserves documentation.
2) Tables and their rules are cross-referenced.
3) All data elements used in an organization are cataloged and coded.
4) All these tables, data element codes, and cross-referencing information are maintained under computer control.

Following this plan permits:

1) Display of documented decision making.

2) Instant tracing of the effect of decisions throughout the documented structure.
3) Rapid reflection and implementation of decision rule changes at all related lower levels.
4) Improvement in study and design of large integrated systems while providing a total view of the organization's data flows and requirements.
5) Easier application of advanced techniques for systems simulation and information flow studies.

Suggested steps for evolving a decision table usage plan in an organization can be outlined as follows (Fergus [29]):

1) Acquire a brief, broad picture of what decision table usage is all about.
2) Have at least one individual in the organization become an expert with decision tables.
3) Anticipate problems to be incurred through usage.
4) Have a reference document.
5) Encourage the use of decision tables.
6) Explain tables thoroughly to systems users.
7) Follow up on the program:
   a) Identify and correct problems.
   b) Look for areas that are not using tables, and find out why they are not being used.
   c) Keep up with developments in the organization that might suggest changes in the use of tables

In Systematics

Systematics [40, 41, 56] is a set of techniques for designing and describing information systems which permit the user to concentrate on the design and description of a system without having to consider problems concerned with system implementation.

The basic statement in systematics is called an *element*. The element is actually a special case of a decision table. Three features of this decision table-like element are:

1) It is confined to providing rules for obtaining one derivation only. For example, pension contribution and holiday entitlement may both be influenced by length of service; the general decision table structure would consider all the relevant states, but the element considers only those which affect one derivation, pensions *or* holidays, in this case.
2) The "primary conditions" determining when an element is performed, are introduced.
3) Substitute a "use" list for "go to" information in the action entry of a general decision table form. (This use list contains the names of other elements that use the derivative.)

This variant decision table form is considered to be more manageable because the entries are limited to the combinations of conditions that yield the derivation of only one item. The new table is output-oriented in that the designer can work back through the output and determine exactly which rules have been used for a particular derivation. The new table form also avoids any processing sequence, and therefore permits directing attention to any one element, ignoring the rest of the system (Grindley [40]).

In Automatic Test Equipment Systems

The use of a programming language, based on decision table techniques, permits the test engineer to write test statements easily, and permits programming a test specification with minimal knowledge of programming techniques and of the specific test equipment system involved.

The envisaged program involves the process of translating test requirements into a test program. A testing system must automatically perform any sequence of tests on a unit being tested, and must choose a new sequence of tests in accordance with previous results.

A modified decision table structure is used with the test conditions placed in the condition statements quadrant of the table, with the resultant test actions being placed in the action statements quadrant and the necessary testing parameters filling in the rules portion. The advantages gained by the decision table structure are, once again, to enable the test engineer to divorce himself from both knowledge of programming techniques and the test equipment itself

III. DECOMPOSITION AND CONVERSION ALGORITHMS

Evolution of Decision Table-to-Computer Program Translators

Systems analysts and individuals involved in program development were confronted with situations in which existing methods of problem descriptions, such as flowcharting and narratives, were inadequate. As a result, decision tables were invented (McDaniel [67], Pollack [84]). Truth tables like that in Figure 2 and logic tables, such as Federal Income Tax forms, had existed prior to the introduction of decision tables; but they did not have a standard format, nor were they automatically convertible into computer programs (Quine [90, 91]). The logic used in those truth tables provided a ready springboard for pioneers in the development of decision tables. The truth table in Figure 2 indicates the truth values that X and Y can assume, as well as the truth values of the logical statements "X OR Y" and "X AND Y". An up-to-date discussion of the application of decision tables to tax forms is given by Ainslie and Kenney [1].

In 1957 a task group at General Electric, one of the earliest users of decision-structured tables, developed such tables, and a computerized method of solving them. The processor for solving these tables initially operated on an IBM-702, and was subsequently implemented on the IBM-305, 650, and 704. An improved processor and language called TABSOL (Tabular Systems Oriented Language) was implemented on the GE-225 in early 1961 [35, 50, 51, 63, 88]. With the TABSOL processor, an analyst could bypass the programmer, and input the prepared decision table directly to the compiler for processing. This processor, which is still in use, utilizes the horizontal rule format, in which the decision rules are read horizontally. For example, the first decision rule of Figure 3 reads: If Item-A EQ 3 and Item-B NE 4 and Item C EQ 20 then GO TO TABLE-2.

X	Y	A ∨ B	A ∧ B
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F

FIG. 2. Truth table

ITEM-A	ITEM-B	ITEM-C	THEN GO TO
EQ 3	NE 4	EQ 20	TABLE-2
NE 3	EQ 10	EQ 40	TABLE-3
EQ 5	NE 4	EQ 60	TABLE-4
NE 5	EQ 10	EQ 80	TABLE-5

FIG 3 Horizontal rule format (TABSOL)

About the same time, a special committee for the CODASYL (*CO*nference on *DA*ta *SY*stems *L*anguages) Group, studying decision tables, developed a decision table language known as DETAB-X (Decision Tables, Experimental) [7, 16, 17, 19, 69, 77, 78, 79, 80, 82, 86]. In 1965, a follow-up by a working group of the Los Angeles ACM SIGPLAN (Special Interest Group for Programming Languages) resulted in an improved processor called DETAB-65. This processor, integrated into a COBOL compiler, was a significant data processing development [82, 95, 104].

Other decision table innovators in the early 1960s were Hunt Foods and the Sutherland Company. About this time, work by IBM in conjunction with the Rand Corporation, resulted in a processor called FORTAB, using the scientific programming language FORTRAN (Fergus [27]). Prior to FORTAB, most of the processors had used COBOL. The Insurance Company of North America, produced a decision table processor called LOBOC which was used on the IBM-7080 [21, 22, 36, 37].

Several government agencies also participated in this early development. CENTAB was developed through a joint effort of the United States Bureau of the Census and Sperry Rand. A processor subsequent to CENTAB was TAB-7C which, like FORTAB, used FORTRAN as its base programming language.

One of the more unique processors developed during the 1960s called PET (Preprocessor of Encoded Tables) was a product of Bell of Canada. PET, using a PL/I decision table language program, produced PL/I source statements (Fergus [27]). Some of the more recently developed processors include DETRAN, DETOC, DETAP, DETAB-70 (McDaniel [68]), TABTRAN, SMP, DECISUS (Pollack, et al. [88]), and LOGTAB (King [54]), with an application in systematics [40, 41, 56].

Many favorable comments have come from the users of decision tables. The supervisor of preparation of computer programs for the 1964 Census of Agriculture stated that the very extensive and comprehensive consistency checks and the resulting desirable adjustments in data could not have been computerized without the use of decision tables (McDaniel [69]).

An extremely complex file maintenance problem arose at the USAF ARLS (Automatic Resupply Logistic System) at Norton AFB, Calif Almost seven man/years had been spent trying to define the problem using narrative descriptions and flowcharts, but to little avail. Then a crash program using decision tables was implemented. Four analysts spent one week establishing the decision table format. Three weeks later the problem was completed (Fisher [32]).

Techniques Used in Translating Decision Tables

The purpose of this section is to discuss the translation of decision tables into computer programs For this purpose, an orderly procedure, or algorithm, is needed to translate the tabular structure of the decision table into an efficient sequence of machine-executable instructions. The term *decomposition* is used to describe any of the techniques by which decision tables are converted into conventional decision trees or programming language (Pollack, et al. [88]). Two main classes of decision table-to-computer program conversion algorithms are available: tree, or network, structure methods [71, 85, 92, 94, 98] and mask methods [55, 62, 89] The algorithm selection should be based on efficiency criteria established by the user [2, 12, 24, 73, 76, 85, 87, 89, 92, 94, 98, 108].

Efficiency of a decomposition algorithm usually falls into one of two classifications; namely minimization of storage necessary for the object program (Pollack [85], Sprague [100]) and minimization of processing time (King [55], Montalbano [71]). It is impossible, with current techniques, to design an algorithm that would be globally optimal in all situations, so it is necessary to analyze the constraints in each situation before determining what kind of an algorithm to consider. Once an algorithm has been selected, it can either be used in a hand coding technique or built into a preprocessor compiler for automatic translation of tables (Pollack, et al [88])

Most of the decomposition algorithms deal with limited entry decision tables rather than the extended entry or the mixed entry decision tables [85, 92, 94, 98] This does not create a problem because extended and mixed entry decision tables can be easily changed into limited entry decision tables. An extended entry table is illustrated in Table 12a and an equivalent limited entry version in Table 12b

There are two major ways of translating a limited entry table into a computer program. The first technique, called *scanning* (or, at more sophisticated levels, the *rule mask technique* [5, 28, 55, 105]), involves

TABLE 12A. ORIGINAL EXTENDED ENTRY

	1	2	3	4	ELSE
X =	6	2	4	4	
Y =	1	3	1	3	
COMPUTE Z =	X - Y	X + Y	X - Y	X + Y	

TABLE 12B. NEW LIMITED ENTRY

	1	2	3	4	ELSE
X = 2	*	Y	*	*	
X = 4	*	*	Y	Y	
X = 6	Y	*	*	*	
Y = 1	Y	*	Y	*	
Y = 3	*	Y	*	Y	
COMPUTE Z = X + Y		X		X	
COMPUTE Z = X - Y	X		X		

testing each transaction against all pertinent conditions in a single rule and scanning across the rules until one is found in which all conditions are satisfied. The solution is said to be in that rule, and consequently the actions associated with this rule are executed. To minimize run-time for the resultant program the rules are often ordered on the frequency in which they are expected to be selected (Fergus [28], Pollack, et al [88]). An example of the scanning technique is illustrated in Figure 4, using Table 13 as a sample decision table.

Scanning Technique

TABLE 13. SAMPLE SCANNING TABLE

	1	2	3	4	ELSE
C1	Y	N	Y	N	
C2	N	-	Y	N	
C3	Y	Y	-	N	
A1	X	X	X	X	

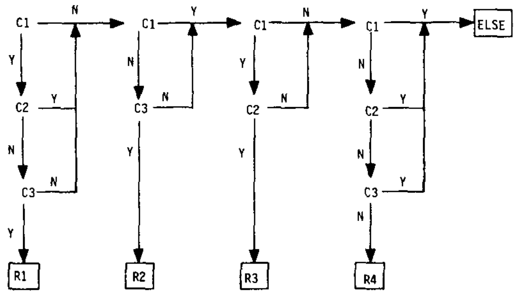


FIG 4 Testing sequence of sample table.

The transaction vector is compared with one condition at a time in each rule. For example, test the table at the first pertinent condition and if the first condition is satisfied, test the transaction entry against the first pertinent condition of the second rule. The complete scanning technique of Table 13 appears in Figure 4. (Note that Table 12 contains mutually exclusive conditions such that, as soon as "Y = 1" has been determined, no further checks on the "Y" values need be made. The * notation eliminates, for example, the "AND" function C1·C2·C3·C4·C5.)

The second technique for translating limited entry decision tables into computer programs is called *condition testing,* or the *network technique* (Fergus [28], Montalbano [71]). This method tests one condition at a time and requires the rules in the decision table to be unambiguous; i.e., one transaction cannot satisfy more than one rule. The network technique takes advantage of this requirement and seeks to isolate the unique rule satisfied by each transaction entry. It is primarily condition-oriented (Pollack, et al. [88]). A typical decomposition tree for a decision table is given in Figure 5 (Fife [31]).

It can be observed from Figure 5 that one row of the original decision has been selected as the starting point The particular row that is selected for a starting point can be based on several different criteria that are discussed later. The condition in the selected row becomes the first comparison in the tree structure. The original decision table is then decomposed into two subtables

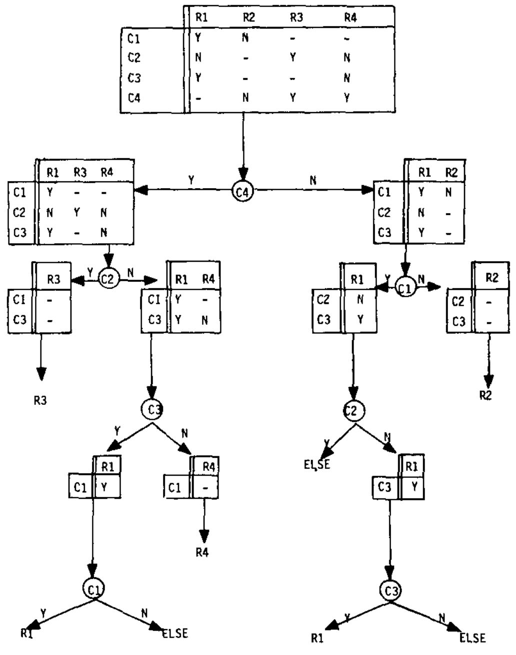


FIG 5. Decomposition tree of a decision tree.

(containing one less row), one subtable and one rule, or two rules; each of these is associated with each branch of the comparison A row is then selected from each of these subtables and its condition is tested. This process is continued until each rule of the original decision table or an ELSE rule appears as one of the branches of a condition (Pollack [85]).

In dealing with these specifics of the two previously mentioned decomposition techniques, the action part of the decision table is omitted because the algorithms are designed to isolate a unique rule which, in turn, defines an action set. (Only limited entry tables are discussed because the extended entry decision tables can be converted to limited entry tables.)

Evolution of Translating Algorithms

Most of the decision table techniques discussed in the literature, as was seen in the previous section, can be divided into two broad categories. Techniques that optimize core storage, and those that optimize execution time; some techniques attempt to optimize both categories.

Montalbano [71], the first to devise techniques for obtaining computer programs to optimize storage requirements and execution time, developed two methods called the *Quick Rule Method* and the *Delayed Rule Method*. The objective of the quick-rule method is to perform, as soon as possible, those tests which will isolate a rule as quickly as possible This method is efficient with respect to storage requirements. The objective of the delayed-rule method is to delay the tests isolating rules as long as possible. This method is efficient with respect to average execution time. Montalbano's work was used as a basis for the techniques developed by Pollack, [85].

The objective of Pollack's first algorithm is to convert a decision table to a computer program using the minimum number of storage locations. In his second algorithm, the objective is to convert a decision table to a computer program in which comparisons can be executed in minimum time (Pollack [85]). The algorithms automatically handle the ELSE rule and isolate any redundant or contradictory decision rules during the conversion process.

A different process that used a rule mask technique was developed by Kirk [62]. This technique resulted in the optimization of storage requirements, but was inefficient in average execution time because it required the sensing of all conditions by way of a mask which is used to screen out nonpertinent conditions according to the input data prior to scanning the decision rules Further work in this area was done by Press [89] whose method offered better run time optimization than Kirk's technique. Another technique that expanded Kirk's work was developed by King [55]. One of the assumptions in this technique is that advance information on evaluation times and frequency of occurrence of rules is available. King's method offers a marked savings in computer run time in comparison with Kirk's, but it uses more core storage space because of the increased complexity of the branching structure.

Some techniques for programming decision tables in higher level languages were explored by Bjork [6] and Veinott [104]. Specifically, they used FORTRAN, COBOL, and ALGOL in their translation of decision tables to programs. One of the most rigorous works on translating decision tables into an optimal branching sequence has been done by Reinwald and Soland [92, 94]. They have developed two algorithms that minimize run time and core storage plus optimizing the resulting test sequence Furthermore, they claim that the two algorithms can be combined to yield a testing sequence that minimizes the total cost of both core usage and run time (Pollack, et al. [88]). Even though these algorithms are quite efficient, they are not widely used because of their complexity. Besides they require prior information concerning the frequency with which the decision rules are satisfied

Further work on Pollack's algorithm has been done by Shwayder [98]. He proposed two alternatives to Pollack's algorithm which he advises will result in lower execution time. His first alternative uses the communications concept of entropy (i.e., a measure of the variability of a set of messages) and Shannon's noiseless coding theorem. This algorithm is most effective when the estimated frequency of the ELSE rule is very low. Shannon's noiseless coding theorem can be used to find the average code-word length, which is necessary in order to *minimize* average code-word length. Shwayder's second modification completely tests the ELSE rule, but results in greater run time. These alternatives do not necessarily lead to globally optimal solutions because they suboptimize one subdecision table at a time.

A technique for parsing large decision tables into smaller ones is offered by Chapin [12, 14] who developed a technique whereby a decision table can shrink to one twenty-fourth of its original size by use of parsing methods. Another parsing technique, proposed by H Strunz [102], permits parsing utilizing only the syntactical characteristic of the decision problem. It requires a description of the problem in decision grid chart format, and allows the development of decision tables within defined limits by avoiding, or at least minimizing, repetition of conditions and actions in the resulting tables. Some of the factors affecting decision table parsing are indicated in Figure 6.

Hierarchies for different levels of decision tables can be established by using the interrelationships of Figure 6. An example of vertical parsing would be separate tables dealing with data at various levels such as

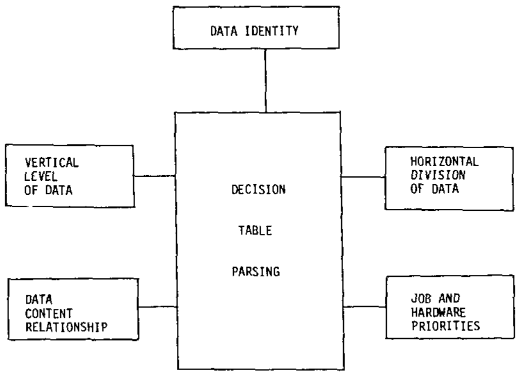

DATA IDENTITY
VERTICAL LEVEL OF DATA
DECISION TABLE PARSING
HORIZONTAL DIVISION OF DATA
DATA CONTENT RELATIONSHIP
JOB AND HARDWARE PRIORITIES

FIG. 6 Parsing of decision tables

file, volume, record, field, character, and bit level. Parsing of data to recognize horizontal structure would utilize separate tables for head, body, and tail of the data sets. Job and hardware priorities would depend on the type of environment in which the decision table is processed.

Data content can be tested or sorted, and then grouped into separate decision tables according to content. The parsing factors shown in Figure 6 can be given different priorities, depending upon the type of processing environment.

Another method of parsing the tables is the use of proper, or more effective, linkage between tables. It is possible, in an action entry in one decision table to direct entry into another table. If the new table is entered without qualifications, then it must be processed from the beginning. If the directive statement is actually a return to a particular rule in a table from which an exit was originally made, then it can be said that the task has been broken into parts; that is, instead of the complete processing of each table, only parts of each table may be necessary to process and satisfy input data (Chapin [12]).

Scanning and Rule Mask Techniques (Masking Techniques)

The straight scanning technique, which has already been discussed, is inefficient with respect to the utilization of core storage and run time. This technique has no remembering capability in its testing sequence, so the same condition may be interrogated many times. One way to improve scanning is by using the rule mask technique (Barnard [5], Kirk [62]).

Rule Mask Algorithm

Many of the authors refer directly to Kirk's article [62] and his algorithm, therefore a fairly detailed outline of this algorithm is given:

1) Prepare a binary image of the *condition matrix* of the table by placing a "1" in each position in which the original table has a "Y" and a "0" in all

other positions. Table 14 is the original Credit Approval decision table, and Table 15 shows the *table matrix* for Table 14.

TABLE 14. Credit Approval

	R1	R2	R3	R4
CREDIT LIMIT OK	Y	N	N	N
PAY EXPERIENCE FAVORABLE	-	Y	N	N
SPECIAL CLEARANCE OBTAINED	-	-	Y	N
DO APPROVE ORDER	X	X	X	
DON'T APPROVE ORDER				X

TABLE 15 Table Matrix for Credit Approval

	R1	R2	R3	R4
C1	1	0	0	0
C2	0	1	0	0
C3	0	0	1	0

2) A masking matrix is needed to screen out nonpertinent conditions from the transaction or data vector prior to scanning the table matrix. A *masking matrix* is made by placing a "1" wherever the original decision table shows a pertinent condition ("Y" or "N"); everything else is set to zero. Table 16 shows the masking matrix for Table 14.

TABLE 16 Masking Matrix for Credit Approval

	R1	R2	R3	R4
C1	1	1	1	1
C2	0	1	1	1
C3	0	0	1	1

3) Prepare a binary transaction vector by placing a "1" in each true condition position and a "0" in all other positions. A simple transaction entry and its vector is shown in Figure 7.

CONDITION 1 - FALSE	0
CONDITION 2 - TRUE	1
CONDITION 3 - TRUE	1

Fig. 7. Transaction vector.

4) The actual scanning operation is made rule by rule. The first rule of the masking matrix is logically multiplied by the transaction vector to eliminate the rule's nonpertinent conditions from the transaction vector. The result is then compared with the first rule vector of the table matrix. If the two are equal, the rule is satisfied. If not, the scan proceeds to the next rule Table 17 illustrates the scanning operation, and it indicates that Rule 2 satisfies the transaction entry.

TABLE 17. Scanning Operation

RULE	DATA VECTOR	"AND"	MASKING VECTOR		RESULT	TABLE VECTOR	EQUAL?
	0	X	1	=	0	1	
1	1	X	0	=	0	0	NO
	1	X	0	=	0	0	
	0	X	1	=	0	0	
2	1	X	1	=	1	1	YES
	1	X	0	=	0	0	

In terms of total storage requirements this approach appears to be very efficient. Each test need appear only once in the program; and additional storage required to generate and interpret the mask may not be much greater than that used for transfer instructions to achieve the branching inherent in the conditional testing techniques.

With respect to average processing time, however, this approach is not very efficient, since all conditions must be tested regardless of the nature of the input, and additional time must be spent generating and interpreting the mask (Reinwald [92, 94]). A method for adding some improvements to the rule mask technique is the interrupt rule mask method.

Interrupt Rule Mask Algorithm

One of the drawbacks to the rule mask technique, as presented above, is that it might produce object programs of longer run time than necessary (King [55]) A modification of the rule mask technique, discussed below, takes into account both rule frequencies and relative times for evaluating conditions. The interrupted rule mask procedure, due to King [55], does not evaluate the rules of a decision table in a sequential manner like the rule mask technique. Before discussing the strategies for interrupting the testing sequence, a few terms need to be defined; therefore, let

T	=	expected execution time for a program;
t_i	=	evaluation time for each condition;
f_j	=	frequency of occurrence for each rule;
S	=	time for carrying out the testing of transaction vector for a single rule;
Σf_j	=	total frequency.

Some of the above conditions must be determined or estimated for the decision table under consideration. The total run time (see Table 18) can be determined for the simple rule mask technique by using the following formula:

$$T = (t_1 + t_2 + t_3 + S)f_1 + (t_1 + t_2 + t_3 + 2S)f_4 + (t_1 + t_2 + t_3 + 3S)f_3 + (t_1 + t_2 + t_3 + 4S)f_2$$

Testing according to frequency of occurrence, and substituting the values of Table 18 into the formula gives:

R1 $(2 + 7 + 4 + 1)$	$* 35$	$= 490$
R4 $(2 + 7 + 4 + 1 + 1)$	$* 30$	$= 450$
R3 $(2 + 7 + 4 + 1 + 1 + 1)$	$* 20$	$= 320$
R2 $(2 + 7 + 4 + 1 + 1 + 1 + 1)$	$* 15$	$= 255$
Total Run Time		1515

TABLE 18 Calculations for Interrupt Rule Mask Technique

	R1	R2	R3	R4	t_i	Σf_j	$\Sigma f_j / t_i$
$f_j / \Sigma t_i$	2 7	2 5	2 2	2.3			
Σt_i	13	6	9	13			
f_j	35	15	20	30			
C1	Y	N	Y	N	2	100	50
C2	N	_	Y	N	7	85	12 1
C3	Y	Y	-	N	4	80	20

Note· Assume $t = 1$ for multiplying the data vector by the masking vector and compare the result with the transaction vector

There are several strategies that can be used to decrease the run time. These strategies usually yield different results, and the one that produces a testing sequence with the lowest total run time should be selected for use in generating the translated code. The strategies do not guarantee optimal testing sequences in all cases, but they do show an improvement in minimizing object program run time (King [55]).

STRATEGY A tests the conditions in descending order of magnitude of relevance frequency (Σf_j). This is based on the supposition that it may be best to evaluate first those conditions most likely to be pertinent. With this strategy, the relative times of evaluation of the conditions are ignored. The testing sequence for Table 18 using this strategy would be C1-C2-R3-C3-R1-R4-R2:

R3 $(2 + 7 + 1)$	$* 20$	$= 200$
R1 $(2 + 7 + 1 + 4 + 1)$	$* 35$	$= 525$
R4 $(2 + 7 + 1 + 4 + 1 + 1)$	$* 30$	$= 480$
R2 $(2 + 7 + 1 + 4 + 1 + 1 + 1)$	$* 15$	$= 255$
Total Run Time		1460

STRATEGY B tests the conditions in descending order of $\Sigma f_j/t_i$. This is based on the supposition that it may be best to evaluate first those conditions with the shortest evaluation times even though they may be less likely to be pertinent. This results in a testing sequence of C1-C3-R2-C2-R1-R4-R3:

R2 $(2 + 4 + 1)$	$* 15$	$= 105$
R1 $(2 + 4 + 1 + 7 + 1)$	$* 35$	$= 525$
R4 $(2 + 4 + 1 + 7 + 1 + 1)$	$* 30$	$= 480$
R3 $(2 + 4 + 1 + 7 + 1 + 1 + 1)$	$* 20$	$= 340$
Total Run Time		1450

STRATEGY C tests the rules in descending order of frequency, evaluating conditions only when they become necessary for testing the rule The testing sequence of Table 18 using this strategy would be C1-C2-C3-R1-R4-R3-R2:

R1 $(2 + 7 + 4 + 1)$	$* 35$	$= 490$
R4 $(2 + 7 + 4 + 1 + 1)$	$* 30$	$= 450$
R3 $(2 + 7 + 4 + 1 + 1 + 1)$	$* 20$	$= 320$
R2 $(2 + 7 + 4 + 1 + 1 + 1 + 1)$	$* 15$	$= 255$
Total Run Time		1515

STRATEGY D tests the rules in descending order $f_j/\Sigma t_i$. This is based on the supposition that it may be best to test first those rules with relatively shorter condition evaluation times, even though they may have lower frequencies. This results in a testing sequence of C1-C2-C3-R1-R4-R3-R2 for Table 18 and a total run time of 1515.

For Table 18, the best testing sequence is derived by using STRATEGY B, which yields a total test time of 1450. This strategy would then be used to translate the decision table into a machine executable sequence of instructions.

The interrupt rule mask technique will use more storage than the simple rule mask technique because of greater program complication. The algorithm relies on user-supplied condition testing times and rule frequency of occurrence. These disadvantages can be outweighed by the marked savings in run time, so users that have large tables should consider using this method rather than the simple rule mask technique.

Conditional Testing and Network Techniques (Tree Structure Techniques)

The basis for the more sophisticated network or tree techniques are two algorithms, due to Press [89], for evaluating nonambiguous limited entry and extended entry decision tables

In the quick-rule method the objective is to make, as soon as possible, those tests which will isolate a rule. This technique reduces the amount of storage required because it minimizes the number of branching instructions.

In Table 19 the condition portion of a decision table is shown. On the right of the table is the row count matrix which indicates the rumber of occurrences of each value in the condition entries of each row. For example, in the first row there are three "1's" and one "0". It can be seen that the smallest nonzero number in the row count matrix is in row one so the conditional interrogations associated with this row would be made. This would isolate rule 3 as indicated in the flow diagram in Table 19. A new subtable and row count matrix are constructed to test the remaining rules. Row two and row three both have the same smallest occurrence value, so they are interrogated, and this isolates the remaining rules. The flow diagram in Table 19 depicts the final result of the quick-rule method as applied to the table.

TABLE 19 Quick-Rule Decision Table

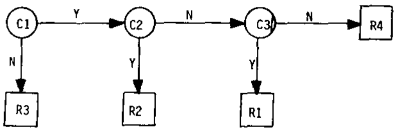
	R1	R2	R3	R4
C1	1	1	0	1
C2	0	1	1	0
C3	1	0	1	0

ROW COUNT	
1	0
3	1
2	2
2	2

SUBTABLE 1

	R1	R2	R4
C1	1	1	1
C2	0	1	0
C3	1	0	0

ROW COUNT	
1	0
3	0
1	2
1	2



The objective of the delayed-rule method is to delay the tests which isolate rules as long as possible. This results in minimizing the average number of executed instructions (Montalbano [71]). An example of the delayed-rule method is shown in Table 20. Here, the row count matrix is searched for a conditional interrogation which will divide the table into two subtables as equal in size as possible. In Table 20 the original table is divided evenly into two subtables and their respective row count matrices. The flow diagram indicates the testing sequence of the conditional rows with respect to minimum row count occurrences in the subtables. Comparing Tables 19 and 20, it can be observed that fewer instructions will be required to isolate a rule using the delayed-rule method. The delayed-rule method minimizes the average number of executed instructions, so it will have less run time than the quick-rule method.

The foregoing network type algorithms can develop greater efficiency in translating a decision table into a computer program

TABLE 20 Delayed-Rule Decision Table

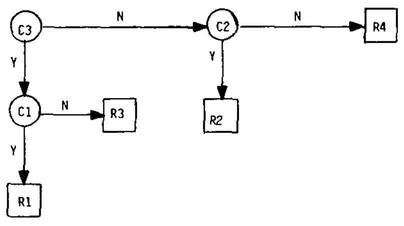
	R1	R2	R3	R4	ROW COUNT 1	ROW COUNT 0
C1	1	1	0	1	3	1
C2	0	1	0	0	1	3
C3	1	0	1	0	2	2

SUBTABLE 1

	R1	R3	ROW COUNT 1	ROW COUNT 0
C1	1	0	1	1
C2	0	0	0	2
C3	1	1	2	0

SUBTABLE 2

	R2	R4	ROW COUNT 1	ROW COUNT 0
C1	1	1	2	0
C2	1	0	1	1
C3	0	0	0	2



by using a more complex algorithm. If several pre-known conditions, such as rule frequency, dash count, delta count, and weighted dash count are available, more efficient algorithms can be used for minimizing core storage, and minimizing run time. Several terms needed to be defined before discussing the algorithms.

The *Column Count* (CC) for a rule is equal to 2^r, where r is the number of dashes (don't-care entries) in the rule. The *Delta Count* (Delta) for a row is the absolute value of the difference between the number of Y's in a row and number of N's in that row. In each row, the *Dash Count* (DC) is equal to the sum of the column counts of all rules that have a dash entry in the row. A *Weighted Dash Count* (WDC) for a row is equal to the sum of the products of rule frequencies and column counts of all rules that have a dash entry in the row.

The testing sequence for both algorithms is (Pollack [85]):

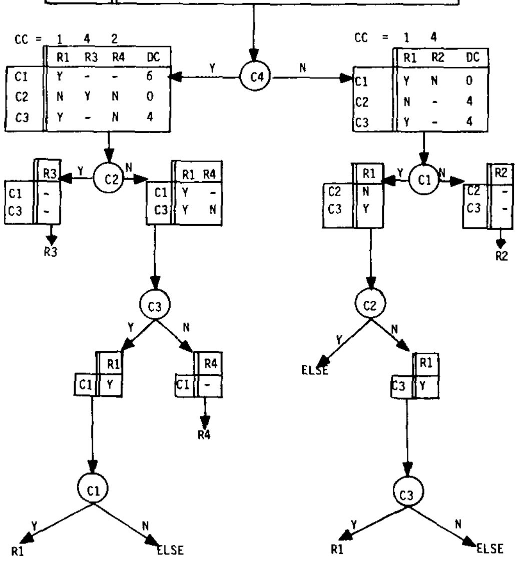
1) One row of the original decision table is selected—the criterion for selection differs for the two algorithms.
2) The original decision table is then decomposed into two subtables (containing one less row), one subtable and one rule, or two rules; each of these is associated with each branch of the comparison.
3) A row is then selected from each of these subtables, and a condition becomes attached to the previously selected condition; i.e., a single condition row is selected and becomes the next comparison of the testing sequence.
4) The process is continued until each rule of the original decision table or an ELSE rule appears as one of the branches of the condition, or a subtable indicates that the original table contained redundant or contradictory rules.

Quick-Rule Algorithm

The objective of the quick-rule algorithm to be discussed is to minimize the number of storage locations (Pollack [85]). This procedure is illustrated in Table 21, and the

TABLE 21. Minimum Core Storage

COLUMN COUNT =	2	4	4	2			
	R1	R2	R3	R4	ELSE	DC	DELTA
C1	Y	N	-	-		6	0
C2	N	-	Y	N		4	1
C3	Y	-	-	N		8	0
C4	-	N	Y	Y		2	1



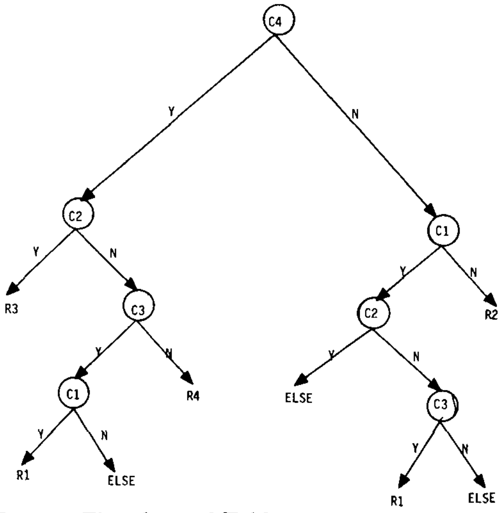


FIG 8. Flowchart of Table 21.

resulting test sequence shown in Figure 8. The steps in the algorithm are:

(1) Check the table for redundancies and contradictions. If two rules do not contain at least one row where one rule has a Y entry and the other has an N entry, the two rules are either redundant or contradictory: they are redundant if they have the same action and contradictory if they do not.

(2) Calculate the column count (CC) and dash count (DC).

(3) Determine the row that has the minimum dash count. If two or more rows have the minimum dash count, select the row that has the maximum Delta.

(4) Taking the row selected in (3), use the YES-NO branch to create two subtables, each containing one or more rules, with one row less than the original row.

(5) If the subtable contains more than one rule return to (1).

(6a) If the subtable has exactly one rule that contains only dashes, that rule has been isolated.

(6b) If the subtable has exactly one rule that contains only dashes, choose any non-dash row and discriminate on it. This will yield a subtable from the satisfied condition and an ELSE rule isolation from the opposing branch.

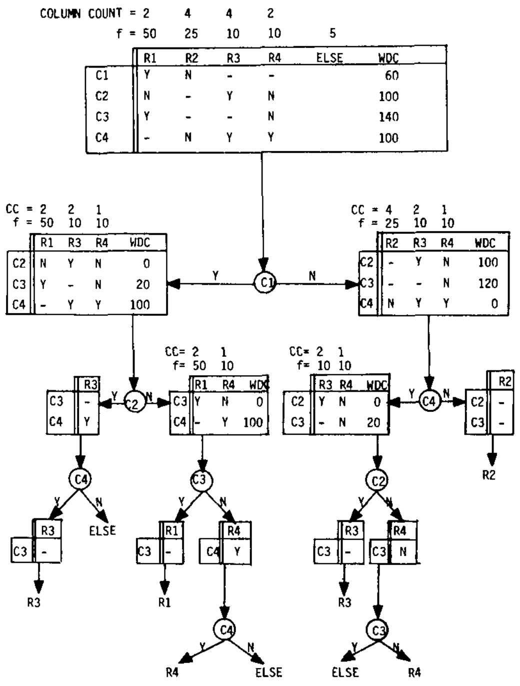
(6c) If no subtable is produced, an ELSE rule isolation is indicated.

(6d) If the subtable has exactly one rule that has one condition with a Y or N entry, discriminate on the condition. The satisfied branch isolates the rule, while the opposing branch isolates the ELSE rule.

Delayed-Rule Algorithm

The objective of the second algorithm is to convert a decision table to a program whose comparisons can be executed in minimum time (Pollack [85]). Some of the assumptions in this algorithm are: a) any rules not specified or implied in the table are assumed to be part of an ELSE rule; b) systems analysts can provide estimates of how often each rule in the table will be satisfied by an average batch of transactions to be tested; and c) relatively few transactions will satisfy the ELSE rule. Table 22 illustrates the procedure for the sec-

TABLE 22. MINIMUM RUN TIME



	R1	R2	R3	R4	ELSE	WDC
COLUMN COUNT =	2	4	4	2		
f =	50	25	10	10	5	
C1	Y	N	-	-		60
C2	N	-	Y	N		100
C3	Y	-	-	N		140
C4	-	N	Y	Y		100

	R1	R3	R4	WDC
CC =	2	2	1	
f =	50	10	10	
C2	N	Y	N	0
C3	Y	-	N	20
C4	-	Y	Y	100

	R2	R3	R4	WDC
CC =	4	2	1	
f =	25	10	10	
C2	-	Y	N	100
C3	-	-	N	120
C4	N	Y	Y	0

	R3
C3	-
C4	Y

	R1	R4	WDC
CC=	2	1	
f=	50	10	
C3	Y	N	0
C4	-	Y	100

	R3	R4	WDC
CC=	2	1	
f=	10	10	
C2	Y	N	0
C3	-	N	20

	R2
C2	-
C3	-

	R3
C3	-

	R1
C3	-

	R4
C4	Y

	R3
C3	-

	R4
C3	N

ond algorithm, and Figure 9 shows the resulting test sequence The algorithm's procedure follows:

(1), (2) Same as the previous algorithm.

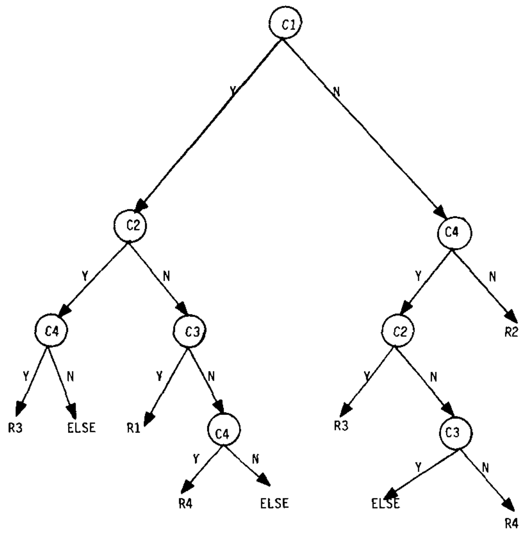


FIG 9. Flowchart of Table 22

TABLE 23. COMPARISON OF TABLES 21 AND 22

RULE NUMBER	TABLE 21 a	TABLE 21 b	TABLE 22 a	TABLE 22 b
1	4 x 50 =	200	3 x 50 =	150
2	2 x 25 =	50	2 x 25 =	50
3	2 x 10 =	20	3 x 10 =	30
4	3 x 10 =	30	4 x 10 =	40
ELSE	(3.6_c) x 5 =	18	(3.6_c) x 5 =	18
		318		288

a = number of comparisons.
b = expected frequency.
c = average number of comparisons for 3 ELSE branches

(3) Determine those rows that have a minimum weighted dash count (WDC).

(3a) If two or more rows have a minimum WDC, select from among them the row that has the minimum Delta. If among these there still exist two or more rows, select the row with the minimum dash count. If there are two or more such rows, select any one of them. The test on dash count does not affect running time, but can save memory space without adding to running time.

(4), (5), (6) Same as the first algorithm.

A comparison of the execution times of the two algorithms depicted in Tables 21 and 22 is shown in Table 23. The execution time is based on processing 100 transactions which have a frequency distribution as indicated in Table 22. Assuming each conditional interrogation takes one time unit, the total test times are indicated in Table 23. It can be seen that the second algorithm is more efficient in run time because it uses 288 time units to process the 100 transactions, whereas the first algorithm uses 318 time units.

A summary of the various decomposition and conversion algorithms is given in Table 24. This table contains both the short (common) and the long class name, as well as major references.

Ambiguities

An aspect that is often ignored in decision table processing is that of ambiguities in the tables. These need to be somehow

TABLE 24 CLASSES OF CONVERSION ALGORITHMS

Short Class Name	*Long Class Name*	*Sub-Algorithms*
Masking Technique	Scanning and Rule Mask Technique	Rule Mask Algorithm Kirk (62)
		Interrupt Rule Mask Algorithm King (55)
Tree Structure Technique	Conditional Testing and Network Techniques Press (89)	Quick-Rule Algorithm Montalbano (71), Pollack(85
		Delayed Rule Algorithm Montalbano (71), Pollack (85) Shwayder (98)

reported to the analyst or the programmer (Muthukrishnan, et al. [73]). King [58] suggests that the rules concerning redundancy, contradiction, and completeness, on which the diagnostic facilities of processors for translating decision tables to programs are based, are unsatisfactory. He states that the important aspect of checking a table is to eliminate ambiguities. He asserts that a check-out of decision table input (in checking for ambiguities) should consist of two parts: 1) if no ambiguity is possible in a particular table, this should be noted; 2) if there are ambiguities then all outcomes in which they occur should be produced, leaving it to the decision table originators to check these facets of the table.

King, in a later paper [61] presents a slightly different approach than the one-rule convention to ambiguities involving multi-rule decision tables. This approach retains the idea that the set of actions corresponding to only one rule is selected for a particular transaction. However, it does not insure it by allowing only one rule to be satisfied, but permits two or more rules to be satisfied provided they have the same action entries. Accordingly, if more than one rule can be satisfied by a transaction with identical action entries, the ambiguity is said to be "apparent," whereas if transactions specify different action entries, the ambiguity is said to be "real," requiring correction (King [61]).

Execution time diagnostics for these ambiguities, as opposed to compile time diagnostics, are implemented by condition tests that provide complete information about the conditions and their relation to the data. The tree methods can not really cope with real ambiguities, and the rule mask techniques do not consider them. Execution time diagnostics for these ambiguities enhance the value of decision table usage in programming (Muthukrishnan, et al. [73])

One of the latest techniques developed is that of Muthukrishnan and Rajaraman [73], which uses execution time diagnostics in pinpointing ambiguities in decision tables. They contend that execution time diagnostics are of immense value in checking out decision tables, because they precisely pinpoint the errors in logic, instead of passing the task to the systems analyst. Pollack [87] and King [61] have disputed this idea and state that it is better to find ambiguities during compilation rather than during execution. In their work, Muthukrishnan and Rajaraman developed two algorithms for programming decision tables, which have the merits of simplicity of implementation, and detection of ambiguities at execution time. The first algorithm is for limited entry decision tables, and clarifies the importance of proper coding in simplifying the mechanics of rule matching; the second algorithm programs a mixed entry decision table directly, without any intermediate conversion, to a limited entry form, which results in storage economy.

Automatic Versus Manual Translation

Effective programming efforts are required to convert decision tables into operational computer programs. This conversion means that tabular representation of information and data must be converted into machine language instructions. The techniques of program conversion have been well developed in the past few years There are four principal approaches which can be used: *manual, semiautomatic, interpretation,* and *automatic conversions* (Glans, et al. [39]).

Manual processing is accomplished by programmer rewriting of each decision table for more efficient and compact representation. This method offers flexibility, and allows the programmer to take advantage of testing certain rules or conditions in a particular sequence. In general, manual programming from decision tables is convenient, and leads to reasonably efficient object programs (Glans, et al. [39]). Any of the higher level languages or assembly language may be used for writing the programs.

Some people call the *semiautomatic conversion* a translator. Basically they are the same thing, so the two terms will be used synonymously. This method converts a decision table format into another programming language that is acceptable as a computer input language. One advantage of processing a table in this manner is that it can be converted into a language such as FORTRAN or COBOL. Thus a table can subsequently be run on any machine that accepts FORTRAN or COBOL. One disadvantage of this method is its relative inefficiency. It requires a two-step process, because the decision table has to be translated into a programming language, and then this source language has to be compiled or assembled into an object program.

The *interpretive conversion* allows for direct storage of the decision table, usually in a coded or compact form, thereby insuring easier maintenance (McDaniel [67]). It is necessary to have the interpretive program in core before inputting the source program. The main disadvantage is the slower solution speed. These programs usually have some restrictions on the type of format and vocabulary used in the source program While this method lacks the sophistication of the other conversion methods, it offers easier program maintenance.

Automatic conversion programs are those which will accept decision tables written in a user source language, and completely convert them to a fully acceptable input, usually at the machine language level (McDaniel [67]). Generally, an automatically converted decision table will require less execution time than interpretive and semiautomatic conversions. This method forces a higher degree of standardization, thus it may encourage more effective communication The disadvantages of automatic conversion are that it tends to be inflexible and is computer-oriented. Conversion of such a processor from one machine to another would require a considerable amount of reprogramming, unless, of course, the processor is written in a higher level language, such as COBOL; e.g., DETAP (Pollack, et al. [86]).

Before selecting the method of conversion, it is desirable to analyze the methods previously discussed, and then to select the one that best fits the situation. Some questions that should be asked during the evaluation are: (1) Is it possible to use the method of conversion? (2) What are the restrictions of the possible methods of conversion? (3) Does the processor produce an efficient code that satisfies the requirements? (4) With respect to the processors that satisfy (3), is the cost of running the processor worth the service it provides (Gildersleeve [34])? After answering these questions and comparing the different conversion methods, it may be found that the most economical solution is to hand code the programs from decision tables, and not to use a preprocessor at all.

Adding, deleting, and restructuring tables is comparable to developing original tables and programs. If a change is to be made in a table, usually extensive hand conversion is needed, again adding more overhead to the desired change. (These overhead costs must be included when considering long-term maintenance.)

IV. CONCLUSION

Decision tables can be a powerful aid in programming, documentation, and in effective man-to-man and man-to-machine communications. Inherent in the design of a decision table is the visual presentation of complex programming logic with relative ease for modification, implementation, and automatic conversion into executable programs. Several such algorithms for converting decision tables to programs, by either manual or automatic techniques, were shown to be feasible, as well as practical for implementation.

An inevitable outcome of increasing use of decision tables in programming has been the development of a large number of package processors and translators for the conversion of decision tables to functional program form. These decision table processors are software programs, which are available for almost any language and hardware configuration. Each processor has standards as to size, format, words in the statement portion, and other required characteristics which must be met by tables prior to processing. Meeting these standards will usually require some manual checking of the tables prior to the preparation of the computer input. Then, the processor may reveal redundancies, missing situations, and contradictions within the table (McDaniel [68]).

Since each table generates a separate segment of coding, each segment can be traced back to the table that generated it; therefore, changes can be easily made by reworking one or more of the tables, and the effect of such changes observed.

Each processor generates straightforward coding, free of programming tricks; thus a programmer should be able to follow any program in a given installation, and make any necessary changes.

A detailed reference table is given in McDaniel's "Decision Table Software" [68], which examines characteristics of many of the processors, including the language of the processor and the output language; the hardware for which a processor has been implemented (in some cases the hardware for which it is being developed); the types of tables accepted as input by a given processor; the cost and availability of a given processor; the number of tables, rules, conditions, and actions allowed by a given processor; and other notable characteristics.

The algorithm and translator used in converting a decision table into a computer program will therefore be determined by the extent of the processing facilities and the constraints of the application program. To be able to use different algorithms and translators provides more flexibility for the users and yet works against the popularity of decision tables in programming because more individual effort is required in determining which algorithm and translation process to use.

ACKNOWLEDGMENT

The author is deeply indebted to the referees for their valuable suggestions and, in particular, to the referee who pointed out a major omission in the original manuscript and provided encouragement in overcoming this difficulty

BIBLIOGRAPHY

1. AINSLIE, R. J., AND KENNEY, A. A. "A tool for for tax practitioners." *The Tax Advisor* (June 1972), 336–345.
2. ARNOLD, H. O. "Utilization of a decision table translator for basic program creation." *SIGPLAN Notices* **6,** 8 (Sept. 1971), 12–19.
3. AUERBACH. "Decision tables—their general construction and acceptance in programming." *Auerbach Standard EDP Reports* (1968), pp 23:030:100–103.
4. ARMERDING, G. W. *"FORTAB:* a decision table language for scientific computing applications." Rand Corporation, **RM-3306-PR** (Sept. 1962), p. 39.
5. BARNARD, T. J. "A new rule mask technique for interpreting decision tables." *The Computer Bulletin,* Vol. 13 (May 1969), 153.
6. BJORK, HARRY. "Decision tables in ALGOL 60." *BIT,* **8**(1968), 147–153.
7. CALKINS, L. W. "Place of decision tables and *DETAB-X." Proceedings Decision Tables Symposium* (Sept. 1962), 9–12.
8. CANNING, R. G. "How to use decision tables" *EDP Analyzer* **4,** 5 (May 1966).
9 CANTRELL, N. H., KING, J., AND KING, F. G. H. "Logic structure tables." *Comm. ACM* **4,** 6 (June 1961), 272–275.
10. CANTRELL, N. H. "Commercial and engineering applications of decision tables." *Proceedings Decision Tables Symposium* (Sept. 1962), 55–61.
11. CHAPIN, NED. "A Guide to decision table utilization." *Data Processing Proceedings 1966,* Vol. 11 (1966), 327–329
12. CHAPIN, NED. "Parsing of decision tables." *Comm. ACM* **10,** 8 (August 1967), 507–512
13. CHAPIN, NED "An introduction to decision tables." *DPMA Quarterly* **3,** 3 (April 1967), 3–23
14. CHAPIN, NED *Flowcharts* Auerbach Publishers, Princeton, N J., 1971, pp. 20–21.
15. CHAPMAN, A. E., AND CALLAHAN, M. A. "A description of the basic algorithm used in the DETAB/65 Preprocessor." *Comm. ACM,* **10,** 7 (July 1967), 447–446.
16. CODASYL Systems Group and Joint Users of ACM, *Proceedings Decision Tables Symposium* (Sept. 1962).
17 CODASYL Systems Development Group. "Decision tables tutorial using DETAB/X" (1962).
18. CODASYL Systems Group, DETAB-X. "Preliminary specifications for a decision tables structured language" (1962)
19 CODASYL, Decision Table Task Force of Systems Committee "Draft of decision table standards" (March 1966).
20. DENOLF, H. "Decision tables an annotated bibliography." *IAG Quarterly* **1** (1968), 67–82.
21 DEVINE, D. J "LOBOC, logical business oriented coding," Insurance Company of North America, Oct 1962.
22. DEVINE, D. J. "Decision tables as a basis of a programming language." *DPMA Quarterly* **7** (1965), 461–466.
23. DIXON, P. "Decision tables and their application" *Computers and Automation* **13,** 4 (April 1964), 14–19

24. EGLER, J. F. "A Procedure for converting logic table conditions into an efficient sequence of test instruction." *Comm ACM* **6,** 8 (Sept. 1963), 510–514.
25. ELLIS, J. "Decision tables, a users' guide." Western Electric Company, June 1967.
26. EVANS, O Y. "Reference manual for decision tables." IBM, Sept. 1961.
27 FERGUS, R. M. "Decision tables—an application analyst/programmer's view." *Data Processing* **12** (1967), 85–109
28. FERGUS, R. M. "An introduction to decision tables." *Systems and Procedures Journal* (July–August 1968), 24–27.
29. FERGUS, R. M. "Good decision tables and their uses." *Systems and Procedures Journal* (Sept.–Oct. 1968), 18–21.
30. FIFE, R. C. "Decision tables, UNIVAC application report " *Spring Joint Computer Conference of Systems and Procedures Association* (April 1965)
31 FIFE, R. C "Decision tables." Systems Programming Dept, UNIVAC, 1966
32. FISHER, D. L. "Data Documentation and Decision Tables " *Comm. ACM* **9,** 1 (Jan. 1966), 26–31.
33. FLETCHER, G R. "Seminar on decision tables." Bureau of the Census, Sept. 1969.
34. GILDERSLEEVE, T. R. *Decision Tables and Their Practical Application in Data Processing.* Prentice-Hall, Englewood Cliffs, N.J., 1970.
35 GENERAL ELECTRIC COMPANY. "GE-225 TABSOL reference manual and GE-224 TABSOL application manual." **CPB-147B,** June 1962
36. GLANS, R. B , AND GRAD, B. "Tabular descriptive languague." *IBM Technical Report 245* (Jan. 1962).
37. GLANS, R B., AND GRAD, B. "7080 decision table system preliminary manual." *IBM Technical Report 2D1 (*April 1962).
38. GRAD, B "Structure and concept of decision tables " *Proceedings Decision Tables Symposium* (Sept 1962), 19–28.
39. GRAD, B "Engineering data processing using decision tables " *Data Processing* **8** (1965), 467–476.
40. GRINDLEY, C B. B. "The use of decision tables within systematics " *The Computer Journal* **11,** 2 (August 1968), 128–133.
41. GRINDLEY, C B. B. "Systematics—a nonprogramming language for designing and specifying commercial systems for computers." *The Computer Journal* **9,** 2 (August 1966), 124–128.
42 HARRISON, W J. "Practically complete decision tables: a range approach. *SIGPLAN Notices* **6,** 8 (Sept. 1971), 89–93.
43 HAWES, M K "The need for precise definition." *Proceedings Decision Tables Symposium* (Sept. 1962), 13–18, 20–21.
44. HAWES, M K. "The use of decision tables for problem specifications." *Proceedings UNIVAC Users Association* (April 1965), 55–61.
45. HIRSCHHORN, E. "Simplification of a class of Boolean functions." *J. ACM* **5,** 1 (Jan. 1958), 67–75.
46. HONEYWELL, INC. *An Introduction to Decision Tables—A Programmed Text,* 1st Ed., Oct. 1969.
47. HUGHES, M. L., SHANK, R. M., AND STEIN, E. S. "Decision tables." MDI Publications, 1968.
48. IBM CORPORATION. "1401 decision logic translator H20-0063," and "1401 decision logic translator H20-04921." *Decision Tables—Practice Problems and Solutions,* 1963.
49. IBM CORPORATION. "Decision tables—a systems design and documentation technique." *IBM,* **F20-8102** (1962), p. 21.
50 KAVANAGH, R. F. "TABSOL—a fundamental concept for systems oriented languages." *Eastern Joint Computer Conference,* Vol. 18 (Dec. 1960), 13–15, 117–136.
51. KAVANAGH, R. F. "TABSOL—the language of decision making." *Computers and Automation* **10,** 9 (Sept. 1961), 15, 18–22.
52. KAVANAGH, R. F., AND ALLEN, M. "The use of decision tables " *Proceedings 1963 Conference of International DPMA,* 318.
53 KAVANAGH, R. F., AND SCHMIDT, D. T. "Using decision structure tables, Part I: Principles and preparation; Part II. Manfacturing application." *Datamation,* Vol. 10, (Feb.–March, 1964).
54. KING, J. E "LOGTAB: a logic table technique." *General Electric* March 1959.
55 KING, P. J. H. "Conversion of decision tables to computer programs by rule mask techniques " *Comm. ACM* **9,** 11 (Nov. 1966), 796–801.
56. KING, P J. H. "Some comments on systematics." *The Computer Journal* **10,** 1 (May 1967) 116–119.
57. KING, P J. H. "Decision tables." *The Computer Journal* **10,** 2 (August 1967), 135–142.
58. KING, P. J. H. "Ambiguity in limited entry decision tables." *Comm. ACM* **11,** 10 (Oct. 1968), 680–684.
59. KING, P. J. H. "The interpretation of limited entry decision table format and relationships among conditions." *The Computer Journal* **12,** 4 (Nov. 1969), 320–326.
61. KING, P. J. H., AND JOHNSON, R. G. "Some comments on the use of ambiguous decision tables and their conversion to computer programs." *Comm. ACM* **16,** 5 (May 1973), 287–290.
62 KIRK, G W. "Use of decision tables in computer programming." *Comm. ACM* **8,** 1 (Jan. 1965), 41–43.
63. KLICK, D. C. "TABSOL." Preprints of Summaries of Paper presented at National ACM, Paper 10, **B-2** (Sept. 1961).
64 LOMBARDI, L. A. "A general business-oriented language based on decision expressions." *Comm. ACM* **7,** 2 (Feb. 1964), 104–111.
65. LONDON, K. *Decision Tables: A Practical Approach for Data Processing.* Auerbach Publishers, Princeton, N.J., 1972.
66. LUDWIG, H. R. "Simulation with decision

tables." *Journal of Data Management* **6** (Jan 1968), 20–27.

67. McDaniel, H. *Applications of Decision Tables*. Brandon/Systems Press, Princeton, N.J., 1970.

68 McDaniel, H *Decision Table Software*. Brandon/Systems Press, Princeton, N.J., 1970.

69. McDaniel, H *An Introduction to Decision Logic Tables*. John Wiley, New York, 1968.

70. Meyer, H I. "Decision tables as an extension to programming languages." *Data Processing* **8** (1965), 477–483.

71. Montalbano, M. "Tables, flowcharts and program logic." *IBM Systems Journal* (Sept. 1962), 51–63.

72. Morgan, J. J. "Decision tables" *Management Services* (Jan.–Feb. 1965), 13–18.

73. Muthukrishnan, C. R., and Rajaraman, V. "On the conversion of decision tables to computer programs." *Comm. ACM* **13,** 6 (June 1970), 247–251.

74. Naramore, F. "Application of decision tables to management information systems" *Proceedings Decision Tables Symposium* (Sept 1962), 63–74.

75. Nickerson, R. C. "An engineering application of logic structure tables." *Comm. ACM* **4,** 11 (Nov. 1961), 516–520

76. Peel, Roger "Decision table translation" *The Computer Bulletin* **13,** 12 (Dec. 1969).

77. Pollack, S. L. "What is DETAB-X?" *Proceedings Decision Tables Symposium* (Sept. 1962).

78. Pollack, S L "DETAB-X: an improved business-oriented computer language." Rand Corporation Memo RM-3273-PR (August 1962).

79 Pollack, S. L., and Wright, K. R "Data description for DETAB-X." Rand Corporation Memo RM-3010-PR (March 1962).

80. Pollack, S. L. "Analysis of the decision rules in decision tables." Rand Corporation Memo RM-3669-PR (May 1963)

81 Pollack, S. L "How to build and analyze decision tables." Rand Corporation Memo P-2829 (Nov. 1963).

82. Pollack, S. L. "CODASYL, COBOL, and DETAB-X." *Datamation* **9,** 2 (Feb. 1963), 61.

83. Pollack, S. L. "The development and analysis of decision tables." *Ideas for Management*, 1964, International Systems Meeting, Systems and Procedures Association, Philadelphia, 1964

84. Pollack, S. L. "Decision tables for systems design" *DPMA Quarterly* **8** (1965).

85. Pollack, S. L "Conversion of limited entry decision tables to computer programs." *Comm. ACM* **8,** 11 (Nov. 1965) 677–682.

86. Pollack, S. L., and Harrison, W. J. "DETAP version III user's guide." *IMI,* July 1969.

87. Pollack, S. L. "Comment on the conversion of decision tables to computer programs." *Comm. ACM* **14,** 1 (Jan. 1971), 52.

88. Pollack, S L, Hicks, H., and Harrison, W. J. *Decision Tables: Theory and Practice*. Wiley, New York, 1971.

89. Press, L. J "Conversion of decision tables to computer programs" *Commun. ACM* **8,** 6 (June 1965), 385–390.

90 Quine, W. B. "The problem of simplifying truth functions." *American Math. Mon*, Vol. 59 (1952), 521–531.

91. Quine, W. B "A way to simplify truth functions." *American Math. Mon.,* Vol. 62 (1965), 627–631.

92. Reinwald, L T, and Soland, R. M. "Conversion of limited entry decision tables to optimal computer programs, I: Minimum average processing time." *J. ACM* **13,** 3 (July 1966), 339–358.

93. Reinwald, L. T. "An introduction to TAB40." Research Analysis Corporation, Nov. 1966.

94. Reinwald, L. T, and Soland, R. M "Conversion of limited entry decision tables to optimal computer programs, II: Minimum storage requirements." *J. ACM* **14,** 4 (Oct. 1967), 742–758.

95 Robinson, F. "Processing of decision tables in COBOL." *Computer Weekly* No. 222/223 (Dec. 1970).

96 Shaw, C. J. "Decision tables—an annotated bibliography." *S D C*, TM-2288/000/00, Dec 1965.

97. Shaw, C J. (Ed) "Decision tables." *SIGPLAN Notices* **6,** 8 (Sept. 1971), 1–111.

98. Shwayder, K "Conversion of limited entry decision tables to computer programs—a proposed modification to pollack's algorithm." *Comm. ACM* **14,** 2 (Feb. 1971), 69–73.

99. Slagle, J. R. "An efficient algorithm for finding certain minimum cost procedures for making binary decisions." *J. ACM* **11,** (1964), pp. 253–264

100. Sprague, V. G. "Letters to the Editor" (On Storage Space of Decision Tables). *Comm. ACM* **9,** 5 (May 1966), 319.

101. St. Clair, P. R., Jr. "Decision tables clear the way for sharp selection" *Computer Decisions* **12,** 2 (Feb. 1970), 14–18.

102. Strunz, H "The development of decision tables via parsing of complex decision situations." *Comm. ACM* **16,** 6 (June 1973), 366–369.

103. Taylor, H. *Decision Table Technique for Computer Systems*. Hirschfeld Press, Philadelphia, Pa, 1968

104. Veinott, C G "Programming decision tables in FORTRAN, COBAL, or ALGOL", *Comm. ACM* **9,** 1 (Jan 1966), 31–35.

105 Verhelst, M. "Procedures for finding optimal and near optimal test sequences for applying rule mask techniques in object programs derived from decision tables." *IAG Quarterly* **1,** (1968), 47.

106. Verhelst, M. "A Technique for constructing decision tables." *IAG Quarterly* **2,** 1 (1969), 27.

107. Williams W. K. "Decision structure tables." *NAA Bulletin,* No. 9 (1965), 58–62.

108. Wright, K. R. "Approaches to decision table processors" *Proceeding Decision Tables Symposium* (Sept. 1962) 41–44.