"Decision" tables are introduced with reference to business data processing. A method of verifying both the completeness and consistency of a problem description is given.

☐ *The conversion of tables to computer programs is considered and a technique of obtaining a computer program which minimizes the branching requirements with respect to both memory and execute time is included. Program debugging and program modification are also discussed.*

Tables, flow charts, and program logic

by M. Montalbano

tables

The kind of table which forms the basis of tabular techniques has four parts so that the information displayed is sorted into four groups. The parts of the table are described as: the *condition stub*, which names logical variables; the *condition entry*, which lists permissible combinations of values for the logical variables; the *action stub*, which names action variables; and the *action entry*, which lists sequences of values for the action variables.

Each set of logical variable values in the condition entry is associated with a set of action variable values in the action entry. Such an association is called a *rule*. A rule is thus of the form: "If A *and* B *and* C *and* . . . are true, then take consecutive actions P *and* Q *and* R *and* . . ."

preliminary example

To illustrate, consider the billing procedure of a wholesaler with three product lines, several classes of customers, and a discount and payment structure which depends upon class of customer, product line, and dollar amount of invoice. These variables are as follows:

product lines	(1) engines (2) pumps (3) fans
classes of customers	(1) retail (2) government agencies (3) engine agents (4) pump agents (5) pump distributors (6) fan distributors
dollar ranges	(1) less than \$10.00 (2) \$10.00 to \$49.99 (3) \$50.00 to \$99.99 (4) \$100.00 or more

The information listed in the example thus far is the raw material for all decisions about discount and terms. It is also,

except for minor differences in arrangement, a completed condition stub.

Note the ready-made code by which reference can be made to the varying combinations determining the wholesaler's billing decisions. A three-digit number, whose positions each represent a value for one of the three kinds of variables listed (product, customer and dollar range, in turn) can now completely describe any set of factors: the code number 334, for example, designates an order from an engine agent for a fan costing $100.00 or more.

In analyzing a system, one next determines which *significant* combinations of logical variable values occur. The example has three product lines, six classes of customers and four dollar amount ranges. Thus, the total number of possible product-customer-amount combinations is 72. Generally, however, not all possibilities will occur. If, for example, no engine stocked costs less than $50.00, no combinations which include both engine and either code value 1 or 2 in the dollar range would ever occur in actual practice. All such combinations could either be omitted from consideration in the computer program, or included only to check clerical consistency.

Of the combinations which do occur, some may not be significant. Retail purchases, for example, may all be billed identically irrespective of product type or cost. The product-line and dollar-amount tests are thus not significant in this case, since, although different logical combinations do occur, they do not affect the action to be taken.

This requires a further extension of the coding scheme. In the case of tests which are not significant, X replaces one of the digits in the code. For example, $X1X$ will indicate that retail purchasers have only one rule applied to them whatever they order and however much it costs.

Table 1 Billing procedure for sample wholesale problem

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(1) engine (2) pump (3) fan	X	X	X	1	1	$\bar{1}$	2	$\bar{2}$	2	2	$\bar{2}$	3	$\bar{3}$	Else
(1) retail (2) gov't (3) eng. agt. (4) pump agt. (5) pump dist. (6) fan dist.	1	2	2	3	3	3	4	4	5	5	5	6	6	
(1) less than $10.00 (2) $10.00 to $49.99 (3) $50.00 to $99.99 (4) $100.00 or more	X	4	$\bar{4}$	3	4	X	X	X	1	2	X	X	X	
discount	0	15%	0	33%	40%	10%	25%	10%	30%	33%	15%	25%	10%	error
consignment	no	no	no	yes	yes	no	yes	no	no	no	no	no	no	
terms	c.o.d.	net 30	net 30	net 30	net 30	net 30	net 30	net 30	30 — 60 — 90 —	30 — 60 — 90 —	net 30	net 30	net 30	

One further convention completes the code for our present purposes. A bar ($\bar{}$) over a digit indicates that it is the only one not admissible in the position it occupies. Government agencies, for example, may get discounts only on purchases totalling $100.00 or more, irrespective of product class. The corresponding coding would thus be $X24$ if the discount applied and $X2\bar{4}$ if it did not.

It should now be possible to interpret Table 1, which displays all of the relevant facts. This table is divided into four parts by intersecting vertical and horizontal double lines. The upper left quadrant is the condition stub; the upper right is the condition entry. The lower left is the action stub; the lower right the action entry. The columns to the right of the vertical double line describe the rules. These are "if . . . then . . ." statements in which the "if" portion is described above the horizontal double line and the "then" portion is described below it. For example, the rules corresponding to columns 1, 2, 6, and 14 of Table 1 are, respectively, as follows.

rule 1 If order is from a retail purchaser, then allow no discount, do not ship on consignment, ship C.O.D.

rule 2 If order is from government agency and totals $100.00 or more, then allow 15% discount, do not ship on consignment, terms are net 30 days.

rule 6 If order is from an engine agent, but is not for an engine, allow 10% discount, do not ship on consignment, terms are net 30 days.

rule 14 If no one of the previous rules applies, a coding error has been made. (The code 132, for example, would be an error, since this company does not stock engines which cost less than $50.00.)

condition entry usage

The remainder of this paper will focus largely on the portion of a table in which logical relationships are displayed—the condition entry. The condition entry, especially when characterizing a complex structure, is useful in:

programming to compile sets of branching instructions which occupy minimum space in computer memory and which require a minimum average number of executions,

analysis to make easy, comprehensive checks on the completeness and consistency of sets of logical alternatives,

debugging to maintain identifiers which will display the prior "branch history" of a program without breakpoint or statement-by-statement monitoring, and

modification to modify sets of branching instructions quickly, accurately and with a full realization of all the implications of such a modification.

For convenience, the codes identified in the previous section will be called *rule identifiers*. The condition entry of Table 1 is made up of rule identifiers: $X1X$, $X24$, $X2\bar{4}$, $133\cdots\bar{3}6X$, Else. (This last is in a special category discussed below.) The condition entry will now be regarded as a set of rule identifiers and de-

Figure 1 Quick-rule method of deriving flow charts from tables

I	II	III	IV	V	VI	VII	VIII	IX	X	1	2	3	4	5
1	1	1	1	1	2	2	2	2	2	5	5			
1	1	1	2	2	1	1	1	1	2	7	3			
1	2	3	3	3	1	2	3	3	4	2	2	5	(1)	
1	2	2	2	3	2	2	2	2	2	(1)	8	(1)		
1	2	2	2	2	1	1	1	1	1	6	4			
1	2	2	3	3	1	2	2	3	3	2	4	4		
3	4	4	5	5	1	1	2	2	3	2	2	2	2	2

I	V	X	II	III	IV	VI	VII	VII	IX	1	2	3	4	5
1	1	2	1	1	1	2	2	2	2	3	4			
1	2	2	1	1	2	1	1	1	1	6	(1)			
1	3	(4)	2	3	3	1	2	3	3	(1)	2	4		
(1)	(3)	2	2	2	2	2	2	2	2		7			
1	2	1	2	2	2	1	1	1	1	4	3			
1	3	3	2	2	3	1	2	2	3	(1)	4	2		
3	5	3	4	4	5	1	1	2	2	2	2		2	(1)

IV	VI	II	III	VII	VIII	IX	1	2	3	4	5
1	2	1	1	2	2	2	2	3			
(2)	1	1	1	1	1	1	5				
3	(1)	2	3	2	3	3		2	3		
2	2	2	2	2	2	2		5			
2	1	2	2	1	1	1	3	2			
3	1 √	2	2	2	2	3		4	(1)		
√ 5	1	4	4	1	2	2	(1)	2		2	

VII	IX	II	III	VIII	1	2	3	4	5
2	2	1	1	(2)	2	(1)			
1	1	1	1	1	3				
2	3	(2)	3	3		(1)	2		
2	2	2	2	2		3			
1	1	2	2	1 √	(1)	2			
2	(3)	2	2	2		3			
(1)	2	4	4	2 √		(1)		2	

tailed consideration will be given to each of the areas cited above.

To examine programming, another example which requires identification of ten pipe products is introduced in Table 2. One with redundant information was chosen intentionally to illustrate the elimination of redundancy by means of rule identifiers.

The two programming objectives are to minimize (1) the number of branching instructions in memory and (2) the average number of executed branching instructions.

For simplicity, assume binary branching, though the arguments given would be equally valid for other types. Since differentiation among the ten products is required, the minimum number of binary branching instructions will be nine. If the ten products occur with equal frequency, the theoretical minimum average number of branching instructions executed would be $\log_2 10 = 3.32$.

Figures 1 and 3 illustrate two different methods of converting the condition entry of Table 2 to a set of branching instructions. Figures 2 and 4 display the resulting flow charts. Both flow charts have nine branchpoints (the minimum number); but one will require an average of 5.4 executed branch steps compared to 3.4 for the other.

The procedure followed in Figure 1 will be called the *quick-rule* method. In the quick-rule method, the objective is to make as soon as possible those tests which will isolate a rule. The procedure of Figure 3 will be called the *delayed-rule* method. In the delayed-rule method, the objective is to delay as long as possible the tests which isolate rules.

Consider Figure 1. At the top of the page is the condition entry portion of the original table, represented now as a set of ten rule identifiers labelled, as they are in Table 2, with the Roman numerals I-X. To its right is a seven-by-five array which displays a row-by-row count of digit occurrences in the condition

Table 2 Condition stub and condition entry for sample pipe problem

	I	II	III	IV	V	VI	VII	VIII	IX	X
(1) black (2) galvanized	1	1	1	1	1	2	2	2	2	2
(1) single length (2) double length	1	1	1	2	2	1	1	1	1	2
(1) plain end (2) threaded only (3) threaded and coupled (4) threaded one end	1	2	3	3	3	1	2	3	3	4
(1) light wall (2) standard wall (3) heavy wall	1	2	2	2	3	2	2	2	2	2
(1) unoiled (2) oiled	1	2	2	2	2	1	1	1	1	1
(1) uniform (2) semi-random (3) random	1	2	2	3	3	1	2	2	3	3
(1) 1-inch (2) 1½-inch (3) 2-inch (4) 2½-inch (5) 4-inch	3	4	4	5	5	1	1	2	2	3

Figure 2 Flow chart derived from Table 2 by quick-rule method

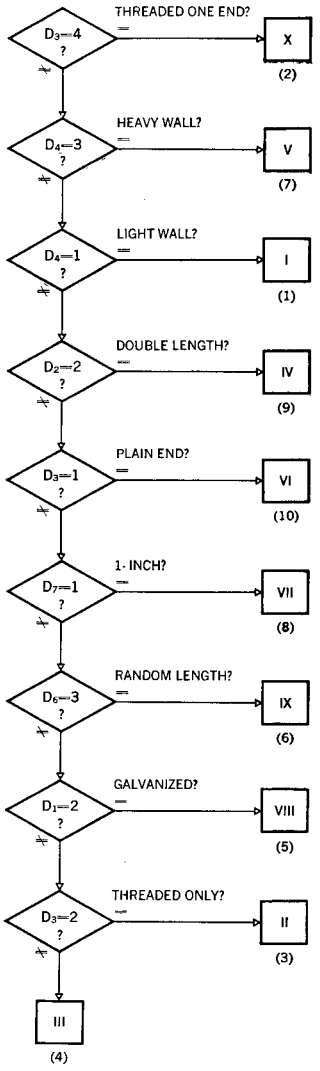
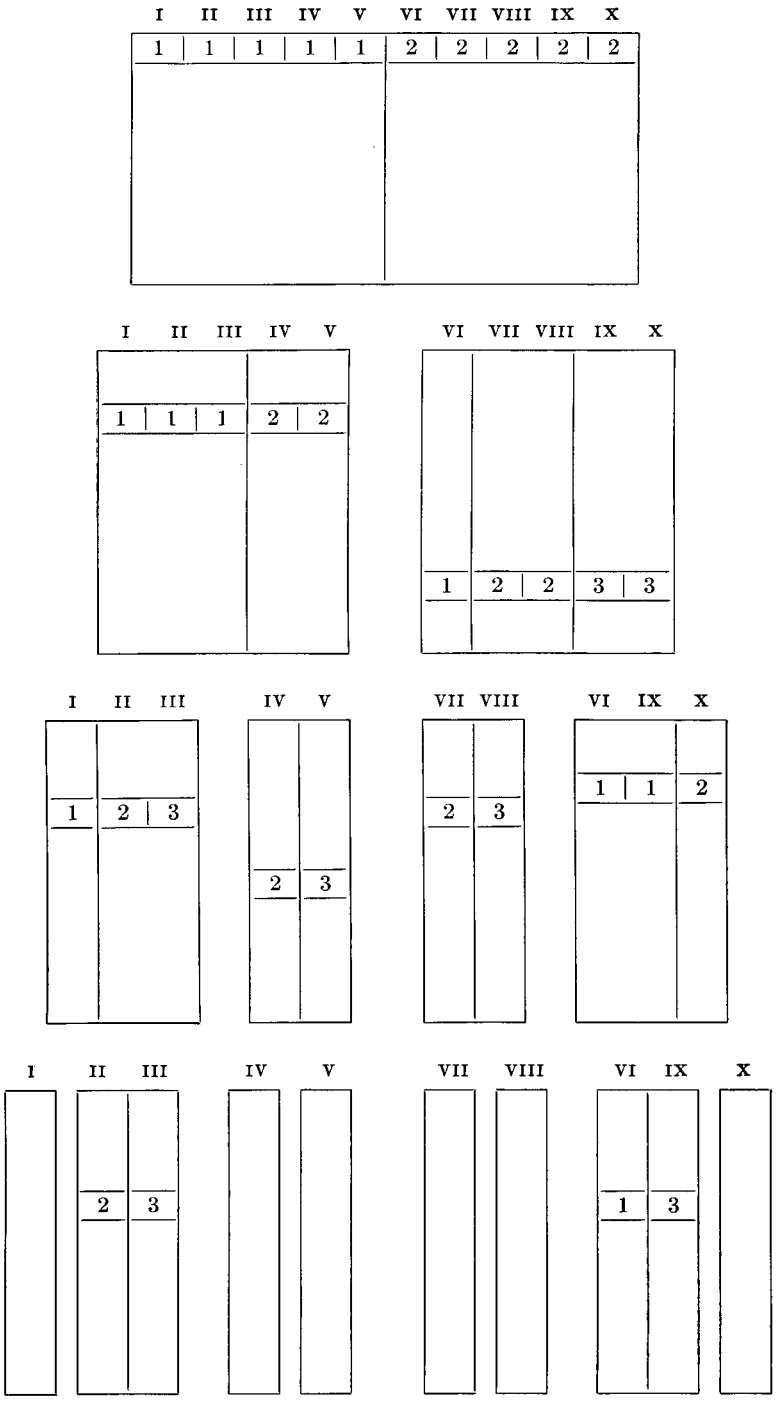


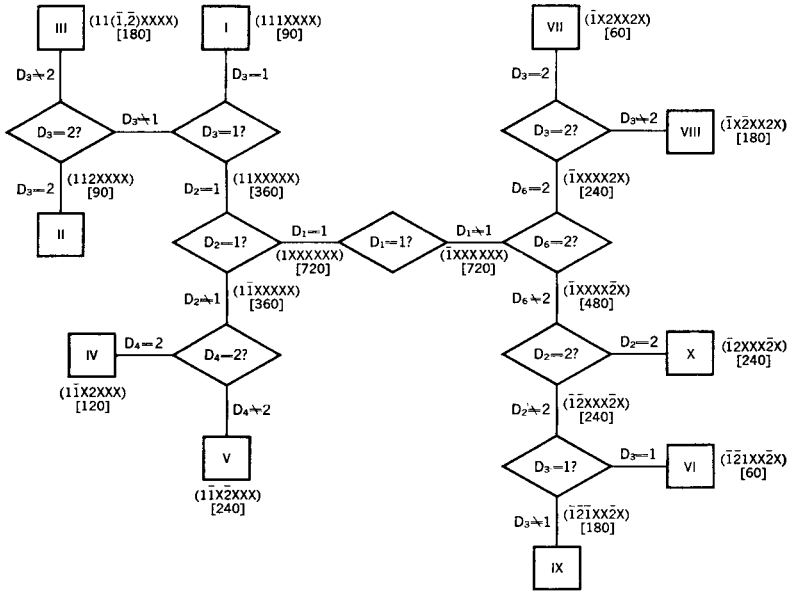
Figure 3 Delayed-rule method of deriving flow charts from tables



entry. This array is called the *row count matrix*. The entries in this matrix tell, for the row in which they appear, how many times a 1 occurs in the condition entry, how many times a 2 occurs, etc.

Looking at the first row of the row count matrix, observe that 1 occurs five times in the first row of the condition entry, as does 2.

Figure 4 Flow chart derived from Table 2 by delayed-rule method



Similarly, the third row is seen to be made up of two 1's, two 2's, five 3's and one 4.

In the procedure illustrated in Figure 1, those questions are asked which will determine a rule as quickly as possible. This is accomplished by looking for the smallest number in the row count matrix and asking the question associated with this number. In Figure 1, the smallest number in the row count matrix is 1 which occurs three times. These 1's indicate that there is one occurrence of a 4 in row 3; one occurrence of a 3 in row 4, and one occurrence of a 1 in row 4. The corresponding questions are: "Is this product threaded on one end?", "Is this product heavy-wall?" and "Is this product light-wall?" An affimative answer to one of these questions gives corresponding identification of the product as X, V or I. The first three branchpoints of Figure 2 ask these questions.

These products are now eliminated from further consideration. The condition entry is thus reduced to seven columns. The row count matrix for this reduced condition entry shows four 1's. In this case, however, the 1's occur in pairs, so only two rules can be isolated at this stage; rules IV and VI. Rule IV can be selected on the basis either of a 2 in the second position or a 5 in the seventh position. Similarly, rule VI can be isolated on the basis either of a 1 in the third position or a 1 in the sixth position. The circles in selected rules IV and VI show the tests actually made in the flow chart of Figure 2; the checkmarks show the alternative tests which could have been made. The remaining steps follow in the same manner. The complete flow chart (Figure 2) is the end result of the process.

As previously noted, this flow chart is efficient with respect to storage but not efficient with respect to average execution time. Let us consider Figure 3 to see how the tests can be scheduled so as to minimize average execution time.

In Figure 3 (in which we omit the row count matrix and the untested rows in the condition entry), tests are scheduled to delay rule identification as long as possible. The procedure employed might be described as "Ask those questions first which will make the two differentiated groups of rule identifiers as *similar* in size as possible." This procedure is illustrated in Figures 3 and 4. If the rules are of equal frequency, the flow chart of Figure 4 will result in an average number of 3.4 branch-instruction executions per product. Like its predecessor, the flow chart of Figure 4 also requires minimum memory space. (The numbers in brackets and parentheses which are shown in Figure 4 will be discussed later.)

If the rules were not of equal frequency, but their relative frequencies were known, the "minimum-average-path" principle just described would require only minor modification. Each rule would have its relative frequency associated with it as a "weight." Instead of a row count matrix, one would have a *row weight count matrix*. The objective would then become to divide the condition entry into groups of as nearly equal weight as possible.

analysis

Descriptions of complicated sets of interacting decisions are frequently inconsistent or incomplete so that analysis is mandatory. The rule identifiers provide a ready means to check sets of such statements for both completeness and consistency. This kind of checking can be done: first, by the system analyst to establish his own understanding; second, by the programmer to check the system analysis; and third, by the compiler to check the program. A limited discussion supporting this statement is given in the remainder of the paper. A more thorough discussion would require a paper in its own right.

completeness

Consider the occurrence of a "don't-care" indication in the condition entry and its effect on the "table-to-flow-chart" procedure discussed above.

Rule 2 in the wholesaling example (Table 1) has $X24$ as its rule identifier. The X—the "don't care" indicator—in this case signifies that any permissible digit in the first position will lead to rule 2; in other words, 124, 224 and 324 are equivalent rules—as long as our order is from a government agency and is for a total amount of $100.00 or more, the 15% discount will apply, whether the article purchased is an engine, a pump, or a fan. Thus, the effect of a "don't-care" indicator is to consolidate several columns into one—the number of columns depending upon the number of alternatives possible for the logical variable to which the "don't-care" applies. If two "don't-care" indications occur in the same column, the number of columns consolidated into one is $m \times n$, where m and n are the number of alternative values for the first and second logical variables, respectively. The extension to three or more "don't-care" entries is done similarly.

Table 3, and Figures 5 and 6 may make this clearer by illustrating the relationship between compound rules (those in which "don't-care" entries occur) and simple rules (those in which each variable is specified exactly).

Table 3 is an uncoded table or, rather, the uncoded condition

Table 3 Condition stub and condition entry of sample table containing "don't care" entries

	1	2	3	4	5	6	7	8
A eq. 2.5	Y	Y	Y	N	N	N		Else
B vs. 19	$<$	$<$	$=$	$=$	$=$	$=$	$>$	
C				$=P$	$=P$	$=Q$		
D is pos.	Y	N		Y	N			

stub and condition entry of a table. The "don't-care" condition is indicated by the absence of an entry in any cell where the test is not significant. Note that one of the rules in this table is a catch-all rule called "Else." This is the rule that applies if no other rule holds.

Figure 5 shows just the condition entry portion of the same table, first as a coded set of compound rules in which "don't-care" is indicated by the presence of an X in a cell and, second, as the equivalent set of simple rules into which the compound rule table can be analyzed.

In the simple rule table, we have allowed four columns for rule 8—the Else rule. The total number of simple rules possible is the same as the total number of rule identifiers we can write. In this case, the rule identifiers are of the form $a\ b\ c\ d$, where a can be either 1 or 2; b can be either 1, 2, or 3; c can be either 1 or 2; and d can be either 1 or 2. The total number of different rule identifiers is thus $2 \times 3 \times 2 \times 2 = 24$. We get 20 of these 24 from the decomposition of compound rules 1-7 into simple rules. The remaining four must therefore make up the simple rules combined into rule 8.

Figure 6 illustrates some of the further analysis possible. Part I of Figure 6 is merely a copy of the compound rule table of Figure 5 written in more compact form, with a number under each compound rule which tells how many simple rules it represents.

Part II is a row count matrix for rules 1-7. Note that the count is not the actual number of occurrences of X, 1, 2, or 3 in Part I,

Figure 5 Coded condition entry portion of Table 3

		1	2	3	4	5	6	7	8
A	(2)	1	1	1	2	2	2	X	E
B	(2)	1	1	2	2	2	2	3	
C	(2)	X	X	X	1	1	2	X	
D	(2)	1	2	X	1	2	X	X	

	1		2		3				4	5	6		7								8			
A	1	1	1	1	1	1	1	1	2	2	2	2	1	1	1	1	2	2	2	2				
B	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3				
C	1	2	1	2	1	1	2	2	1	1	2	2	1	1	2	2	1	1	2	2				
D	1	1	2	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2				

Figure 6 Analysis of "Else" rule in Table 3

Part I

	1	2	3	4	5	6	7	8
A	1	1	1	2	2	2	*X*	*E*
B	1	1	2	2	2	2	3	
C	*X*	*X*	*X*	1	1	2	*X*	
D	1	2	*X*	1	2	*X*	*X*	
	(2)	(2)	(4)	(1)	(1)	(2)	(8)	(4)

Part II

	(1–7)		
X	1	2	3
8	8	4	
0	4	8	8
16	2	2	
14	3	3	

Part III

(1–7)		
1	2	3
12	8	
4	8	8
10	10	
10	10	

Part IV

(8) (Else)	
	4
4	
2	2
2	2

but the weighted number of occurrences—each occurrence of a digit adds the column weight (the number at the foot of the column in Part I) to the row count.

Part III shows the row count matrix for rules 1-7 after the *X*'s have been converted into the 1's and 2's, or the 1's, 2's, and 3's they represent. In the first row of Part II, for example, the count in the *X*-column is 8. Since *A*, the variable in the first row, takes on only the values 1 and 2, half of the simple rules covered by the *X* have 1's in this digit position and half have 2's. Thus 4 added to the 1-count and 4 to the 2-count gives the Part III entries in that row; twelve 1's and eight 2's. (Part III could, of course, have been obtained directly from the simple rule table of Figure 5.)

Part IV of Figure 6 displays the row count matrix for rule 8, the Else rule. This is obtainable directly from the Part III row count matrix. In the digit positions corresponding to two-valued variables, the 24 simple rules will divide into twelve 1's and twelve 2's. In the three-variable position, the simple rules will divide into eight 1's, eight 2's, and eight 3's.

An inspection of Part III shows which digit values are missing in each row. These missing digit values must occur in the Else column. This information is enough to enable writing out the Else row count matrix. (We could also go on to write out all the simple rules which make up the Else column, if we chose.)

At the end of a procedure such as this—one which can be carried out easily by the system analyst, the programmer, or the computer—the complete set of alternatives implicit in the set of logical variables specified in the condition stub has been verified. For every possible combination, the action to be taken has been specified.

consistency

Consistency as well as completeness can be checked as part of the same procedure. Suppose the original Table 3 had contained a rule *R* with entries as shown in Table 4. The rule identifier for *R* is 1212. Rule *R* is thus seen to be inconsistent with rule 3, since the rule identifier of rule 3 is 12*XX*—which includes 1212 as one

of the simple rules of which it is made up. In other words, the statement that one action is to be taken whenever $A = 1$ and $B = 2$ is inconsistent with the statement that a different action is to be taken if $A = 1$, $B = 2$, $C = 1$, and $D = 2$.

Table 4

Y
$=$
$=P$
N

The "don't-care" entries modify somewhat the flow-chart optimization procedure described above. The additional principle required can be described briefly as: "Delay as long as possible any test in a row in which X's appear."

Applying this principle and the delayed-rule principle previously described we get Figure 7: a flow chart for the problem originally described by Table 3. It contains seven branchpoints, which is the minimum to distinguish among eight rules. The branch segments are labelled with partial or complete rule identifiers. (Note that the rule identifier for the Else rule is $21XX$—which is in keeping with the row count matrix information.) These branch segment labels appear above or to the left of the branch segment which they label.

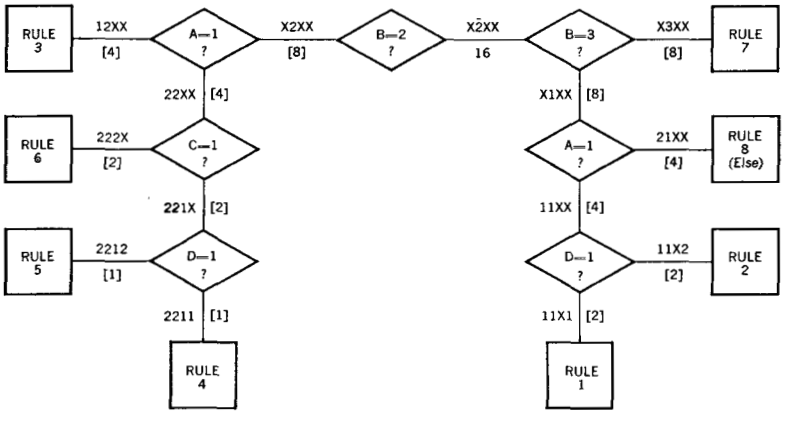
The number in square brackets (below or to the right of each branch segment) is the number of rules included in that segment. There are 24 rules to apportion. The first test splits these 24 into groups of 8 and 16. In the 8-branch, the second test splits these into groups of 4 and 4. (One of these groups turns out to be the compound rule, 3.) In the 16-branch, the second test splits the rules up into groups of 8 and 8. (One of these groups turns out to be rule 7.) The splitting process continues until all rules have been identified. If rules 1 through 8 occur with equal frequency, the average is 3.25 branch instruction executions per rule.

debugging

Figures 4 and 7 display digital codes which are associated with the lines connecting branchpoint symbols. These codes record the branch history of the line segment next to which they appear. A code like $\bar{1}\bar{2}XXX\bar{2}X$ (Figure 4), for example, means that variables one, two and six have been tested prior to the point in the flow chart at which this number appears, and that the tests have determined that the current value of the first variable is not 1, the second variable is not 2, and the sixth variable is not 2.

These codes are, in effect, incomplete rule identifiers when they

Figure 7 Flow chart derived from Table 3 by modified delayed-rule method



occur between successive branchpoints, and complete rule identifiers when they occur between branchpoints and action boxes. If a computer program were written so as to maintain the current rule identifier (complete or incomplete) in a standard location available to a debugging program, the information provided by such a rule identifier would be a powerful debugging aid. It would also provide assistance in systematic program checkout and diagnostics, since an automatic means to cycle through all permissible rule identifiers could be devised for each program so constructed.

As previously noted, the numbers in square brackets which accompany the rule identifiers merely record how many simple rules are contained in them. The large numbers associated with the rules of Figure 4 measure the large amount of redundancy in the system.

program modification

The point about modification can readily be made. Consider the pipe program described by Table 2. Suppose it is wished to discontinue manufacturing light-wall pipe and substitute a new product: galvanized, double-length, threaded-one-end, heavy-wall, oiled, semi-random, 4-inch pipe. All that is needed is to strike one column from the table and add another—an easy transition compared to the difficult redrawing of a flow chart which this same problem would require.

the use of tables

The preceding discussion, in which tables and their corresponding flow charts have been prepared in conjunction, serves to indicate the relative merits of the two forms of display of logical structure. The table is superior to the flow chart in displaying computer-independent information; the flow chart is superior in displaying computer-dependent information. If the problems discussed above were to be programmed for machines which did branching by some other method than binary choice, the flow charts would be different but the tables would be unchanged. In this sense, tables are problem-oriented; flow charts are computer-oriented.

There is another aspect in which tables are problem-oriented. We have been considering primarily the condition entry portion of the table, since this is the point of greatest difference with past practice. But the division of the table into its four sections is, in itself, a useful aid in problem description. The condition stub is a list of all the questions and permissible answers pertinent to a particular problem. The condition entry is a list of all permissible combinations of answers. The action stub is a list of all the actions pertinent to a particular problem. The action entry is a list of the permissible sequences of actions. The rules serve to associate a specific set of answers with a particular sequence of actions.

The question arises as to the type of problem for which tabular techniques are effective. In some types of problems the stages of the accompanying procedural study can be characterized by the following four kinds of questions.

1 "How do you know when to or how to do such-and-such?"
2 "Do this condition and that condition ever occur together?"
3 "What steps might you have to take in doing such-and-such?"

4 "When this condition and this condition and this condition, etc., occur together, which steps do you actually take?"

Since the four sections of a table correspond functionally to these four kinds of questions, tabular techniques should be advantageous in the associated types of problems.

concluding remarks

There are, of course, many problems meriting further investigation: tables in their present form can become unwieldy when problem segments are prefaced by one or two simple decisions rather than six or seven complicated ones; it would sometimes be convenient to have rules in a table refer to other rules in the same table; rule identifiers in which the variable values are connected by "or" rather than "and" would sometimes be a convenience, and so on. Such further investigation would be desirable, since it would enhance the already considerable merit of tables as a means to implement program logic.

BIBLIOGRAPHY

Grad, Burton, "Tabular Form in Decision Logic," *Datamation*, July, 1961.

Kavanagh, Thomas F., "TABSOL—A Fundamental Concept for Systems Oriented Languages," *Proceedings of the 1960 Eastern Joint Computer Conference.*

Evans, Orren Y., "Advanced Analysis Method for Integrated Electronic Data Processing," *IBM General Information Manual*, #F20-8047.