Decision tables

By P. J. H. King*

This expository paper outlines the essential features in decision table development and illustrates applications in various fields. A comprehensive bibliography is also presented.

There is at present a growing interest in decision tables among programming and systems professionals. For the past several years occasional articles and papers have been appearing in American magazines and journals pointing out the value of decision table techniques to the user and discussing translation to a lower level of program. This body of literature is reviewed and discussed below. Little work, as yet, seems to have been done on this subject in the U.K.

The purpose of this paper is to explain the basic ideas of the decision table approach, to illustrate its wide applicability by way of examples, and to present a bibliography. The use of decision tables in the context of more comprehensive systems is also mentioned.

The basic ideas

A decision table is a useful tool when the rules for handling a data record are more complex than a single simple discriminating test. Usual practice is to record and analyse this type of situation by means of a flowchart. This is then used for writing a program made up of a number of branches. Such programs, even though written in a high level language, are often not readily comprehensible in the absence of the accompanying flowchart or without constructing one.

Consider the flowchart of **Fig. 1.** This describes the rules for preparing lists of students divided into five categories on the basis of end of year exam results and other information in the case of those who have failed their main subject. The flowchart is drawn more formally in **Fig. 2.** We see that it represents a five way program branch on the outcome of tests on some of four distinct conditions. The five different actions may be thought of simply as program routes. MM and MA are marks obtained in main and accessory subjects respectively. C is set positive if special consideration is recommended and R is set positive if the student wishes to repeat the course. P is the pass mark.

If asked to program on the basis of Fig. 2 the ALGOL programmer would write something like:

```
if MM < P then (if C > 0 then go to R3
                else if R > 0 then go to R4
                else go to R5)
     else if (MA < P then go to R2
              else go to R1);
```

The COBOL programmer might write something like:

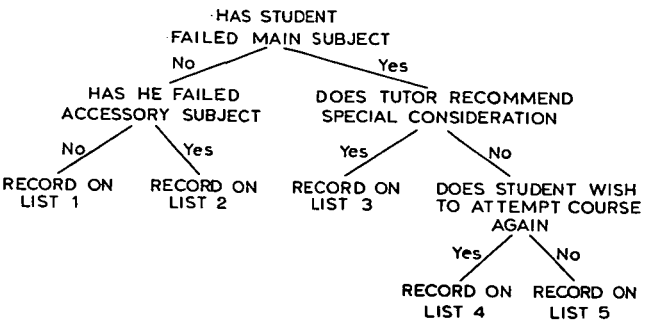

Fig. 1

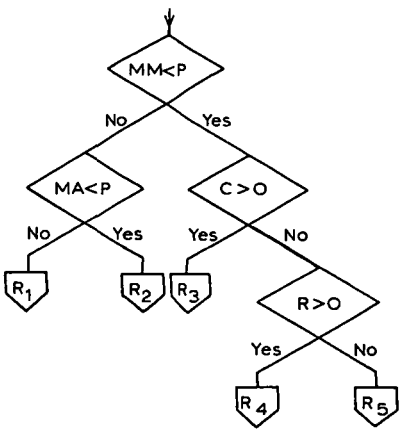

Fig. 2

IF MM IS LESS THAN P NEXT SENTENCE OTHERWISE IF MA IS LESS THAN P GO TO R2 OTHERWISE GO TO R1. IF C IS GREATER THAN ZERO GO TO R3 OTHERWISE IF R IS GREATER THAN ZERO GO TO R4 OTHERWISE GO TO R5.

The PL/1 programmer might write:

```
IF MM < P THEN IF C > 0 THEN GO TO R3;
                        ELSE IF R > 0
                             THEN GO TO R4;
                             ELSE GO TO R5;
          ELSE IF MA < P THEN GO TO R2;
                        ELSE GO TO R1;
```

The programmer with a decision table feature in his language would draw up a table similar to **Table 1.**

* *Computer Unit, University College of Wales, Aberystwyth, Cards.*

Table 1

MM < P	N	N	Y	Y	Y
MA < P	N	Y	—	—	—
C > 0	—	—	Y	N	N
R > 0	—	—	—	Y	N
GO TO	R1	R2	R3	R4	R5

Table 1 illustrates the generally accepted notation for decision tables. The table is physically divided into four quadrants by double lines. The upper two quadrants, the Condition Stub and Condition Entry, describe conditions which are to be tested. The lower two quadrants describe the actions to be taken depending on the outcome of these tests. A "rule" consists of a set of outcomes of the condition tests together with the associated actions. It is a single vertical column of the table to the right of the vertical double line. Thus the third rule in the above table says that if the first and third conditions listed in the condition stub are satisfied then the action to be taken is GO TO R3. The dashes in this column against the second and fourth conditions mean that the outcomes of the tests on these conditions are not relevant in deciding whether the action for this rule should be taken. Thus to decide whether the action is to be GO TO R3 these two conditions need not be tested. In this example there are five rules.

Diagrammatically the structure of a decision table is shown in **Fig. 3.**

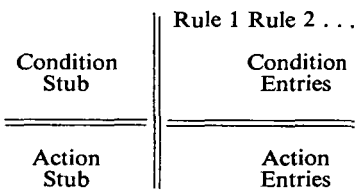

Fig. 3

The lines of a table may be written in either extended or limited entry form. In the example of Table 1 the condition section of the table is in limited entry form. This means that the condition is completely specified in the stub and the entry portion contains only Y's (for Yes) showing where a condition must be satisfied, N's (for No) showing where a condition must not be satisfied, or dashes which indicate that a condition is immaterial ("non-pertinent") for the particular rule. The Action Entries in Table 1, on the other hand, are in extended entry form. This means that the statement to be obeyed is only partially specified in the action stub and is completed by the action entry.

The second example shown in **Table 2** is taken from an input editing procedure. This has the condition part of the table in extended entry form. For example, R_3 will be satisfied if CODE = 13 and AMT > 50, the third condition, for which a dash has been entered, being non-pertinent for this rule. In

Table 2

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	E
CODE =	7	7	13	13	29	29	43	
AMT >	99	—	50	—	99	—	—	
AMT ⩽	—	99	—	50	—	99	—	
TYPE "ERROR 1"	—	—	—	—	—	—	—	×
TYPE "ERROR 2"	×	—	×	—	×	—	—	—
TYPE CODE, AMT	×	—	×	—	×	—	—	×
GO TO A	—	×	—	×	—	×	×	—
GO TO B	×	—	×	—	×	—	—	×

this table the action stub is in limited entry form. It lists five complete statements. If the conditions for R_3 are satisfied then the second, third and fifth of these will be executed. Limited entry format for the action part of the table comprises complete statements being entered in the stub and the entries being either × (the action *is* to be taken) or dash (the action *is not* to be taken). It should be noted that whilst in the condition stub no significance is attached to the order in which the conditions are stated, in the action stub the order in which the actions are stated is the order the action sub-set specified for a particular rule will be executed. The example of Table 2 contains the ELSE (or ERROR) feature not previously mentioned. This is the rightmost column which contains no entries in the condition part of the table. It specifies the actions to be taken if none of the rules are satisfied. It must always be included where the rules given do not cover all possibilities. Where they do it can be omitted. The column headings R_1, R_2, . . ., E in Table 2 are not part of the table but have merely been included for ease of reference.

Returning to the example of Table 1 the advocate of decision table use maintains that the table has a clearer and more readily apparent meaning than any of the usual programming language versions. Moreover, whilst the flowchart is a necessary adjunct to the preparation and understanding of any one of these programs it is quite unnecessary to the construction or understanding of the corresponding decision table.

Probably the most important aspect of the decision table notation is that it contains *less* information than a corresponding flowchart. A flowchart contains the logical rules of the problem and *also* specifies the procedure by which the outcomes are to be arrived at. Several flowcharts can usually be drawn for the same problem, all in a sense "correct". They are the same from the point of view of embodying the logical decisions. They are different procedures for giving effect to these. A decision table specifies only the logical rules. As the order in which the conditions in the condition stub are given has no significance, and neither is there any significance in the order in which the rules are stated, the condition section of the table contains no implications about the procedure for its implementation. This aspect is discussed in more detail below.

Thus Decision Table notation provides the means of passing over to the compiler (or perhaps more correctly to the compiler writer) the construction of a procedure for implementation. According to the criteria considered important some procedures will be better than others. This choice of procedure has hitherto been regarded as the province of the applications programmer. In a general way it has been left to his good sense to ensure that the most important and frequently required tests are done first. In effect he is required to see that the flowchart corresponding to the program is a reasonable one. He uses available information on frequency of outcome of the various cases and whether core minimization or run time minimization is the more important, as best he can in achieving this. A further development in programming languages will be to hand this information along with the decision table to a compiler which will then be responsible for this. Thus decision tables not only offer a clearer way of stating the logic of a program but also provide the notational means of extending the scope of automatic programming.

The development suggested in the previous paragraph is not only required of business data processing languages but is also needed in the scientific field. At a lecture in London in January 1963, (10), Professor F. Hoyle when discussing the use of computers in his field referred to programming languages and said " . . . I would be much happier with an autocode if its main aim was to help with the logical structure of a program rather than with the arithmetical details. Mathematics already provides us with admirable notations for setting down equations. If an autocode were to provide equally powerful notations for the logical structure . . . My point is that at present we are offered assistance with the comparatively easy part of programming and not with the hard part . . .".

It is my view that decision tables provide in part the notational advance for which Professor Hoyle was asking.

Further examples

Three further examples are given in this section. The reader is recommended to ignore these where he is not familiar with the subject-matter.

In (2) an ALGOL procedure is given. This contains the following assignment.

$t :=$ (**if** $xw > 0$ **then** $-xx/(2{\cdot}0 \times xw)$ **else if** *phi* [*upper*] $>$ *phi*[*lower*] **then** $3{\cdot}0 \times vt[lower] - 2{\cdot}0 \times vt[mid]$ **else** $3{\cdot}0 \times vt[upper] - 2{\cdot}0 \times vt[mid]$);

It is my view that **Table 3** expresses the required action more clearly and is more readable. A worthwhile extension of ALGOL would be to allow the use of this type of tabular form.

Computers are now being used in the field of investment as an aid to portfolio management (e.g. see "Computer Service Speeds Investment Choice", *The Times*, 19 January 1967). The essential feature in such use is that a file is maintained with detailed records of possible investments. The investment manager specifies what he considers important criteria, the file is then interrogated and a list of those investments conforming to the criteria is produced. It is suggested that the facility to specify the requirements in decision table form would be useful in this context.

Table 3

$xw > 0$	Y	N	N
$phi[upper] > phi[lower]$	—	Y	N
$t := -xx/(2{\cdot}0 \times xw)$	×	—	—
$t := 3{\cdot}0 \times vt[lower] - 2{\cdot}0 \times vt[mid]$	—	×	—
$t := 3{\cdot}0 \times vt[upper] - 2{\cdot}0 \times vt[mid]$	—	—	×

Table 4

P.E. ratio $<$	12	12	12	12	10	10	
Div. yield $>$	5	5	4	4	6	6	
Cash/market value $>$	$2\frac{1}{2}$	$2\frac{1}{2}$	5	5	0	0	
No. of div. increases $\geqslant$	1	2	1	2	1	2	
No. of div. reductions $=$	0	1	0	1	0	1	
List A	×	—	×	—	×	—	—
List B	—	×	—	×	—	×	—
Ignore	—	—	—	—	—	—	×

Suppose the basic requirement is a price/earnings ratio of less than 12, a dividend yield of more than 5%, cash in the last balance sheet of at least $2\frac{1}{2}$% of market value, no dividend reductions, and at least one increase within the last six years. However, a dividend yield of 4% is acceptable provided the cash in balance sheet is 5% of market value or alternatively no cash in the balance sheet (but no bank overdraft) is acceptable provided the yield is more than 6% and the price/earnings ratio is less than 10. As an alternative to the no dividend reduction criterion one dividend reduction is allowed provided there have been two or more increases, but shares in this category are required to be listed separately. **Table 4** expresses the requirements. Moreover the general format is suitable for use as input to a generalized program to perform this type of interrogation.

The third example shows how a decision table feature can be of value in non-numeric programming. A familiar example used when introducing list processing and its applications is that of the formal differentiation of an algebraic expression; see for example Wilkes (29). Usually, as in the cited example, the result will be produced in a form involving expressions including zeros and ones, for example $0 + y$ and $1 \times y$. It is then observed that a further program will be required to perform the tidying-up process. A decision table is helpful in writing this program.

Following the cited paper of Wilkes we assume the expression is stored as a list structure representation of the Polish notation. We write a subroutine *TIDY* (*R*) where the argument *R* will always be a list. This list

will consist of a single cell the CAR of which is either an atom, or a list which is a triad. Thus the expression y will be as shown in **Fig. 4**(*a*), the expression $by + a$ as shown in **Fig. 4**(*b*).

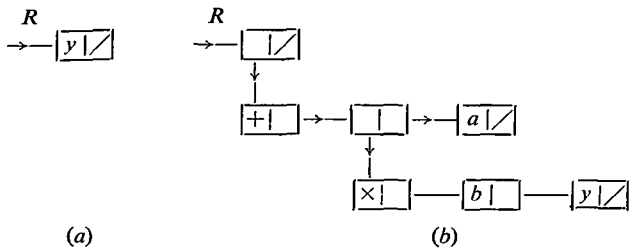

Fig. 4

The program given below and in **Table 5** is written in an informal ALGOL-like notation. We assume the existence of suitable facilities for recursion, i.e. the local variables A, B, C, the argument R, and the return address, are held in nesting stores which are pushed down when values are assigned and popped up on exit from the subroutine by the RETURN statement. The reader is recommended to work through this program with R having the value $ay + bz$ say, and then again after making the substitution $y = 0$, $z = 1$.

SUBROUTINE *TIDY(R)*
IF *CAR(R)* = *ATOM* **THEN RETURN**
OTHERWISE
(*A*:=*CAR(R)*, *B*:=*CDR(A)*, *C*:=*CDR(B)*)
TIDY(B)
TIDY(C)
TABLE
RETURN

Note that the *TABLE* embodies the fact that the numerator when the operand is / will be the second member of the triad and that the exponent when the operator is ↑ will be the third member of the triad. Note that there is no provision for monadic operators so that $0 - y$ will be unchanged. Extension of the program to cover monadic operators replacing, for example, cos (0) by 1 is straightforward.

Flowcharts and decision tables

The purpose of this section is to introduce and discuss aspects of translating decision tables into procedures. We do this using the example of **Table 6** as illustration.

Table 6

C_1	Y	Y	—	N
C_2	Y	—	N	—
C_3	N	Y	N	—
GO TO	R_1	R_2	R_3	R_4

Note that there is apparent ambiguity in this table. If, for a particular set of data, all three conditions give negative outcomes on testing, then both R_3 and R_4 are satisfied. Such apparent ambiguity is usually of no consequence, as relationships exist between the conditions which automatically resolve it. This situation is illustrated by **Table 7** which is ambiguous if both conditions are true. We see from the nature of the conditions that this is impossible.

Table 7

Age < 18	Y	—	N
Age > 65	—	Y	N
GO TO	R_1	R_2	R_3

The matter of ambiguity in decision tables is a topic requiring further work but is not of consequence in the discussion which follows.

Without use of the decision table concept the logic embodied in Table 6 would be presented as a flowchart. There are a number of possible flowcharts that could be

Table 5

TABLE

CAR(A) = '×'	Y	Y	Y	Y	—	—	—	—	—	—	—	—	—	—	—	
CAR(A) = '+'	—	—	—	—	Y	Y	Y	—	—	—	—	—	—	—	—	
CAR(A) = '−'	—	—	—	—	—	—	—	Y	—	—	—	—	—	—	—	
CAR(A) = '/'	—	—	—	—	—	—	—	—	Y	Y	Y	—	—	—	—	
CAR(A) = '↑'	—	—	—	—	—	—	—	—	—	—	—	Y	Y	Y	Y	
CAR(B) = '0'	Y	—	N	—	Y	Y	N	—	Y	N	N	Y	—	N	N	
CAR(B) = '1'	—	—	—	Y	—	—	—	—	—	—	—	—	Y	—	—	
CAR(C) = '0'	—	Y	—	N	Y	N	Y	Y	—	Y	—	—	—	Y	—	
CAR(C) = '1'	—	—	Y	—	—	—	—	—	—	—	Y	—	—	—	Y	
CAR(R) := '0'	×	×	—	—	×	—	—	—	×	—	—	×	—	—	—	—
CAR(R) := *CAR(B)*	—	—	×	—	—	—	×	×	—	—	×	—	—	—	×	—
CAR(R) := *CAR(C)*	—	—	—	×	—	×	—	—	—	—	—	—	—	—	—	—
CAR(R) :='∞'	—	—	—	—	—	—	—	—	—	×	—	—	—	—	—	—
CAR(R) := '1'	—	—	—	—	—	—	—	—	—	—	—	—	×	×	—	—

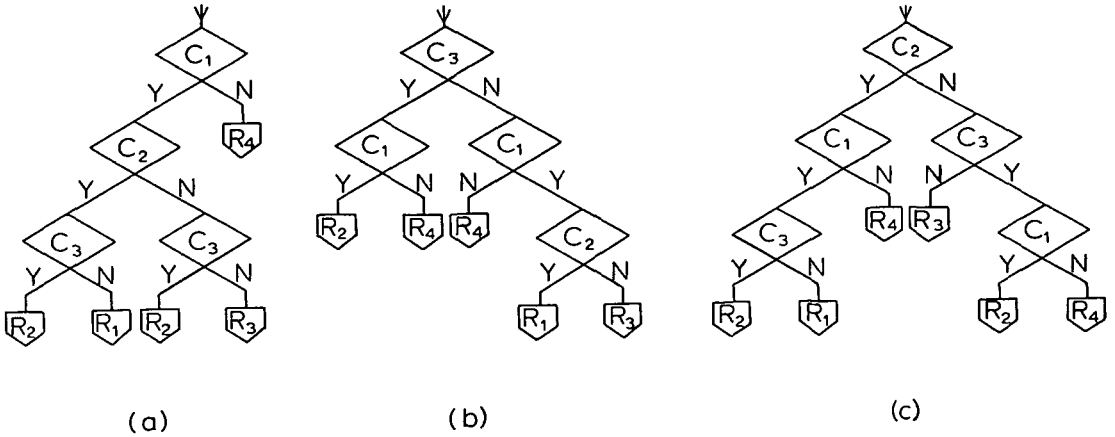

Fig. 5

drawn to give the required logical outcomes. Three of these are given in **Fig. 5.**

They embody the same logical consequences but differ in the procedures they specify for arriving at the outcomes. Whilst all giving the correct result these procedures may not be equally good from the point of view of store requirements and program run time. For example, suppose C_1, C_2 and C_3 are tests which each requires the same amount of store, then the implementation of 5(*a*) or 5(*b*), each of which requires the coding of four tests with five transfers of control, will be more economical of store than 5(*c*) which requires five tests to be coded, i.e. a 25% increase. If, however, economy in the use of store is not vitally important but it is essential to minimize run time, and it is known that R_3 will occur much more frequently than any of the other outcomes, then Fig. 5(*c*) must be the preferred implementation. In this case only two tests (C_2 and C_3) will have been carried out in deciding that R_3 is the condition that holds, whereas implementation of either 5(*a*) or 5(*b*) means that three tests will be carried out. 5(*c*) can therefore give approaching 33% economy of run time over 5(*a*) or 5(*b*). If the times to test the data for the conditions are not equal then 5(*c*) could give an even better saving. This, for example, would be the case if some of the conditions were not simple arithmetical comparisons between quantities in core store but asked whether codes were members of reference lists some of which, perhaps, are not immediately available in core. Similarly, although making the same demands on the store for program space, 5(*a*) will be a better procedure to implement than 5(*b*) if R_4 is the most frequent outcome. On the other hand 5(*b*) will very probably be better than 5(*a*) if R_2 is a very frequent outcome.

It is not the purpose of this paper to attempt an analysis of the implementation of a decision table in terms of selecting the best flowchart, but only to attempt to exhibit the problems and considerations involved. What does seem clear is that there is scope for software development which will achieve this in a much more precise and effective way than hitherto. The burden on the Computer Applications Scientist (for want of a term which does not introduce artificial distinctions into the profession!) would then be to provide the logical rules for processing in decision table form together with expected frequencies of outcomes. Note that the best implementation of a particular table in one installation may not be the best in another. This is because the suitability of the implementation is a function of the statistical properties of the data on which it is to operate. These will differ in different installations even though the logical requirements may be identical.

The transformation of a decision table with associated information into the "best" (in some sense) flowchart must still be regarded as a field for research. It should be emphasized, however, that this need in no way inhibit the every day use of decision tables by programmers. These tables give clarity and precision and are an aid to arriving at a near optimum procedure. For example, suppose the outcomes in Table 5 occur with frequencies of 10%, 40%, 45% and 5% respectively, and that all three conditions take an equal time to test. Suppose also that run time is all important. The state of the art is such that a formal algorithm cannot yet be stated which guarantees to produce the best flowchart. Most programmers, however, could produce a reasonably good practical solution. It is suggested that the reader should try this example. Note that a flowchart can be found that is better than all three of Fig. 5.

Rule mask techniques of implementation

The previous section discusses the implementation of a decision table in terms of conversion to a conventional form of program. A fundamentally different approach which might be regarded as semi-interpretive has been described by Kirk (14) and discussed further by King (12). This involves the expansion of the condition section of the table to limited entry form (if not already

in that form) and its representation in the program as two binary matrices, the *mask matrix* and the *decision matrix*. The program operates by representing the outcomes of the condition tests as a binary vector. This is multiplied logically, element by element, with a column of the mask matrix and the result tested for equality with the corresponding column of the decision matrix. There will be equality if the rule is satisfied and inequality if it is not. The rules are tested successively until one is found which is satisfied. If none are satisfied then the Error or Else outcome is obtained.

The rule mask technique is illustrated using Table 6. The mask matrix, M, is obtained by representing pertinent conditions (the Y's and N's) by 1 and non-pertinent conditions by 0; the decision matrix, D, by representing the yes conditions (the Y's) by 1 and all others by 0. Thus for Table 6 we would have the matrices shown in **Fig. 6.**

$$M = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Fig. 6

Suppose the first condition, C_1, held but C_2 and C_3 did not hold when testing a particular set of data: then we would obtain the vector shown in **Fig. 7**(*a*).

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

(*a*) (*b*)

Fig. 7

Logical multiplication with the first column of M would leave this unaltered, and we do not get equality on comparison with the first column of D so that the first rule is not satisfied. Similarly logical multiplication with the second column of M would leave the vector unaltered, there is not equality with the second column of D and so the second rule is not satisfied. However, logical multiplication with the third column of M produces the vector of **Fig. 7**(*b*). This is equal to the third column of D and so the third rule is satisfied. Note that at each stage it is the original vector derived from the data which is used in the logical multiplication.

The principal disadvantage of the technique as described by Kirk (14) is that all conditions have to be evaluated to construct the data vector before any rule can be tested. King (12) describes how this disadvantage can be circumvented and the run time of such "rule-mask" programs thereby materially improved. The relative value of the rule mask technique as opposed to the more conventional approach remains an open question. It does appear to have advantages where there is apparent ambiguity and may prove very satisfactory for large tables. Also economy of object program size will be achieved if a program contains a number of tables since the binary matrices can be concisely stored and will serve as arguments to a section of program to carry out the procedure described above.

Bibliography

There is some very readable introductory and illustrative material (3, 4, 8) which the reader not familiar with the concept of decision tables is recommended to consult. In particular the article by Grad (8) contains two interesting examples and, although published as long ago as 1961, some remarks on future developments which are still relevant. The article by Dixon (4) is a good introductory presentation of the case for using decision tables in preference to flowcharts whenever practical.

Two of the earlier references (4, 11) give an introduction in terms of TABSOL which was an experimental tabular language developed for GE machines. This uses a different notation from that now generally recognized and presented in this paper. Evans (6), although not very readable as an introduction, gives considerable detail of how tabular methods were being used in his company at the time of writing, including some actual examples of the documentation.

The reader considering applying decision tables in an engineering context is strongly recommended to consult Nickerson's paper (18). This describes an actual application using the LOGTAB language which was developed for the IBM 704. The introductory paper (3) also introduces the LOGTAB notation and could usefully be read as a preliminary to Nickerson's paper. There is a natural use of decision tables in the context of Information Retrieval by Larsen (15). It is worthy of note, and a sign of progress, that this is the first publication (known to the author) making use of decision tables and not specially commenting on the fact.

Anyone familiar with FORTRAN is very strongly recommended to read the Rand publication by Armerding (1). This describes FORTAB which essentially is the FORTRAN language with a quite extensive and sophisticated decision table facility added to it. This publication is not only descriptive but is also a manual for writing FORTAB programs. A processor exists for FORTAB for the IBM 7090 which operates on the basis of translating the decision table features in the language into normal FORTRAN from which there is then the usual compilation.

Two Rand Corporation publications (20, 21) describe and discuss DETAB-X which is a version of COBOL with extensive tabular features. These, however, suffer from some irksome rules concerning completeness and redundancy. The rationale for these rules is given by Pollack (22) in an early attempt to do some rigorous analysis on the condition section of decision tables. DETAB-X was designed by the Systems Group of the CODASYL Development Committee in the early sixties. The work was regarded as experimental, and a

revised version has now been developed and is known as DETAB-65. A DETAB-65 processor for the IBM 7090 is available. This is written in COBOL, translation being from DETAB-65 to normal COBOL which is then compiled.

There are a number of papers useful to those interested in converting decision tables to conventional branching programs of the type discussed previously, whether the intention is to include the facility in a compiler, to write a pre-processor of the kind used for FORTAB, or to set up a routine clerical process for use by the programmer in performing the conversion manually. Egler (5) gives a fairly simple algorithm which ignores any statistics on rule frequencies but nevertheless was thought by its originator to minimize both store requirements and processing time. It was pointed out by Montalbano (17) that the procedure does not always achieve this. Pollack (19) has presented two algorithms of a type similar to Egler's, one aimed at the problem of minimizing store requirement, the other at minimizing object program run-time. In a letter (which could well be entitled "Pollack's procedures refuted") Sprague (25) has pointed out that neither of these algorithms achieve their objective in every case. In an earlier paper Montalbano (16) also gives similar types of algorithm. There is also an attempt by Press (23) whose paper includes an interesting discussion of Decision Table Languages and some comments on ambiguous tables.

It now seems clear that in general it is not possible to achieve both minimum run-time and minimum store requirement by the same procedure as Egler attempted. Except for small fairly straightforward decision tables the program which achieves minimum run-time will be different from that which minimizes store requirement. It also seems clear from the recent work of Reinwald and Soland (24) that algorithms which are guaranteed to produce the optimum (in some sense) flowchart from a given decision table are likely to be fairly complex and certainly not suitable for manual processing. Procedures of the type proposed by Egler, Pollack, Montalbano and Press should not, therefore, be dismissed out of hand because they do not always achieve the absolute best. It may be that the relative simplicity of these procedures compensates for their not always producing the optimum, for even when they produce a non-optimum solution it is likely to be a fairly good one.

The most rigorous work done on converting decision tables to conventional programs is that of Reinwald and Soland (24). They assume (as most other workers have implicitly taken for granted) that all conditions are testable whichever rule holds, and in the cited paper give an algorithm which produces the program having minimum execution time. The only drawback is that their algorithm requires not just the probabilities of outcome for the various rules but a complete knowledge of the probability structure involved. In practice this may not always prove readily obtainable.

A different approach to translating decision tables to program is the rule mask technique discussed previously. This is simply explained by Kirk (14) and a proposed development of it is given by King (12). Veinott (27, 28) makes some proposals similar to Kirk's (and suffering from the same drawbacks) for programming using the common languages. He also includes some observations on the relative merits of limited entry and extended entry tables.

An interesting paper by Fisher (7) uses decision tables as one element in the context of a comprehensive method of documentation. It is claimed that this method greatly facilitates amendments and alterations by maintenance programmers. Anyone concerned with documentation problems is strongly recommended to read this paper. Grindley (9) includes a restricted form of decision table as an element in a proposed program specification system. The restriction arises from attaching significance to the order in which the conditions are stated. The notation used is also non-standard. Grindley's paper has been discussed in detail in (13).

Bibliography and References

(1) ARMERDING, G. W. (1962). "FORTAB: A Decision Table Language for Scientific Computing Applications", Mem. RM-3306-PR, Rand Corp., Santa Monica.

(2) BROYDEN, C. G. (1965). "Solving Nonlinear Simultaneous Equations", *Maths. of Comp.*, Vol. 19, p. 577.

(3) CANTRELL, N. H., KING, J., and KING, F. E. H. (1961). "Logic Structure Tables", *Comm. ACM*, Vol. 4, p. 272.

(4) DIXON, P. (1964). "Decision Tables and their Application", *Comput. Automat.* (April 1964), p. 14.

(5) EGLER, J. F. (1963). "A Procedure for Converting Logic Table Conditions into an Efficient Sequence of Test Instructions", *Comm. ACM*, Vol. 6, p. 510.

(6) EVANS, O. Y. (1961). "Advanced Analysis Method for Integrated Electronic Data Processing", IBM General Information Manual F20-8047.

(7) FISHER, D. L. (1966). "Data, Documentation and Decision Tables", *Comm. ACM*, Vol. 9, p. 26.

(8) GRAD, B. (1961). "Tabular Form in Decision Logic", *Datamation* (July 1961), p. 22.

(9) GRINDLEY, C.B.B. (1966). "Systematics—A non-programming language for designing and specifying commercial systems for computers", *The Computer Journal*, Vol. 9, p. 124.

(10) HOYLE, F. (1963). "The Relative Abundances of Ni^{58} and Ni^{60}", Transcript of IBM Symposium on Computers in Research and Education, IBM (UK).

(11) KAVANAGH, T. F., and ALLEN, M. "The Use of Decision Tables", *Proc.* of 1963, Conf. of International Data Processing Management Assn. (Data Processing VI), p. 318.

(12) KING, P. J. H. (1966). "Conversion of Decision Tables to Computer Programs by Rule Mask Techniques", *Comm. ACM*, Vol. 9, p. 796.

(13) KING, P. J. H. (1967). "Some comments on Systematics", *The Computer Journal*, Vol. 10, p. 116.
(14) KIRK, H. W. (1965). "Use of Decision Tables in Computer Programming", *Comm. ACM*, Vol. 8, p. 41.
(15) LARSEN, R. P. (1966). "Data Filtering Applied to Information Storage and Retrieval Applications", *Comm. ACM*, Vol. 9, p. 785.
(16) MONTALBANO, M. (1962). "Tables, Flowcharts and Program Logic", *IBM Systems Journal* (Sept. 1962), p. 51.
(17) MONTALBANO, M. (1964). "Letter to Editor (Egler's procedure refuted)", *Comm. ACM*, Vol. 7, p. 1.
(18) NICKERSON, R. C. (1961). "An Engineering Application of Logic Structure Tables", *Comm. ACM*, Vol. 4, p. 516.
(19) POLLACK, S. L. (1965). "Conversion of Limited Entry Decision Tables to Computer Programs", *Comm. ACM*, Vol. 8, p. 677 (also as Mem. RM-4020-PR, Rand Corp., Santa Monica, May 1964).
(20) POLLACK, S. L., and WRIGHT, K. R. (1962). "Data Description for DETAB-X", Mem. RM-3010-PR, Rand Corp., Santa Monica, March 1962.
(21) POLLACK, S. L. (1962). "DETAB-X: An Improved Business-Oriented Computer Language", Mem. RM-3273-PR, Rand Corp., Santa Monica, Aug. 1962.
(22) POLLACK, S. L. (1963). "Analysis of the Decision Rules in Decision Tables", Mem. RM-3669-PR, Rand Corp., Santa Monica, May 1963.
(23) PRESS, L. I. (1965). "Conversion of Decision Tables to Computer Programs", *Comm. ACM*, Vol. 8, p. 385.
(24) REINWALD, L. T., and SOLAND, R. M. (1966). "Conversion of Limited Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time", *Jour. ACM*, Vol. 13, p. 339.
(25) SPRAGUE, V. G. (1966). "Letter to Editor (On Storage Space of Decision Tables)", *Comm. ACM*, Vol. 9, p. 319.
(26) SCHMIDT, D. T., and KAVANAGH, T. F. (1964). "Using decision structure tables", *Datamation*, Vol. 10 (Feb. and March 1964), p. 42 (Pt. I) and p. 48 (Pt. II).
(27) VEINOTT, C. G. (1966). "Programming Decision Tables in FORTRAN, COBOL, or ALGOL", *Comm. ACM*, Vol. 9, p. 31.
(28) VEINOTT, C. G. (1966). "Letter to Editor (More on Programming Decision Tables)", *Comm. ACM*, Vol. 9, p. 485.
(29) WILKES, M. V. (1965). "Lists and why they are useful", *The Computer Journal*, Vol. 7, p. 278.

Book Review

Computer Simulation Techniques, by T. H. Naylor, J. L. Balintfy, D. S. Burdick and Kong Chu, 1966; 352 pages. (London and New York: *John Wiley and Sons Ltd.*, 72s.)

The literature on simulation is full of descriptive accounts of applied problems which have been successfully solved only when a digital computer has been used to simulate the real system. It also abounds with detailed technical articles on the generation of pseudo-random numbers and the efficient use of these to produce random variates from different probability distributions. But apart from descriptions of several specific computer simulation languages comparatively little has been written on *how* to simulate.

This book is an attempt to fill the gap by developing a "methodology for planning, designing, and carrying out simulation experiments". Chapter 1 presents an interesting account of simulation and its relationship to the scientific method from a philosopher's viewpoint, marred only by an unnecessary attempt to classify simulation models. In Chapter 2 a nine-step procedure for planning simulation experiments is proposed and discussed. The procedure suggested is admitted to be arbitrary, but does provide a suitable framework for the remaining chapters of the book which, the authors claim, place "particular emphasis on those aspects of computer simulation which are not treated in existing textbooks". Unfortunately the book does not fully substantiate this claim, though much credit is due to the authors for attempting to systematize some vague but important areas of simulation design and analysis.

Chapter 3 is concerned with the generation and testing of sequences of pseudo-random numbers. It gives a particularly good account of congruential methods and includes a brief but useful appendix on elementary number theory. Chapter 4 presents the usual methods of generating stochastic variates, *viz.* inverse transformation, rejection and the method of mixtures. Methods for sampling from all the standard distributions follow, each being preceded by a laboured description, which must surely be unnecessary since in the preface the authors assume some knowledge of mathematical statistics in their readers. FORTRAN subroutines are given for the generation procedures presented. It is unfortunate that so little is included on generating correlated variates.

For those who believe that simulation techniques were invented by, and exist to solve the problems of, frustrated queueing theorists, examples are given in Chapters 5 and 6 of simulation models applied to queueing, inventory and scheduling systems, and to the firm, industry, and the national economy as a whole. Several useful exercises for the reader are given at the end of both these chapters, although most assume a working knowledge of FORTRAN. The authors' preoccupation with FORTRAN throughout the book is, perhaps, the feature most likely to irritate ALGOL proponents. This is particularly true of the inevitable chapter on simulation languages which follows. The reviewer found this the most disappointing chapter in the book. In spite of detailed descriptions of several languages (all American in origin) no critical comparison is made; and whereas GPSS II is described in 30 pages (reprinted verbatim from the *IBM Systems Journal*), GSP, ESP and CSL are together dismissed in a single page.

Chapters 8 and 9 discuss the problem of verification and design of simulation experiments respectively. The former is in the nature of a philosophical diversion and is too brief to be really useful. Design of experiments receives more attention but the chapter is in effect little more than a valuable literature survey of the area.

The references and bibliography which follow each chapter are one of the strong points of the book; even if the reader may sometimes be disappointed in the content of a section, he is never ignorant of where to look for further guidance. This book is ambitious in its aim and in several respects falls short of achieving it. It is nevertheless a worthwhile addition to the potential simulator's bookshelf. J. C. WILKINSON