Automatic application program interface to a data base

W. D. Haseman* and A. B. Whinston†

As the availabilities of data base management systems become greater the need to interface already existing programs with the data base will become important. This paper presents a technique whereby these programs can be interfaced automatically with the data base through the use of a query/extraction interface, thus providing the capability to retrieve, modify and use data in various application programs without knowing the physical data structure and without using the data manipulation language.

(Received March 1976)

1. Introduction

As the availabilities of data base management systems become greater, the need to interface already existing programs with the data base will become important. These programs may include such things as standard report generators, statistical models, or special output routines such as plots or histograms. When a planner wishes to perform data analysis, he would like to have a collection of programs available in a library which can be used to process the data stored in the data base. Some of these programs may be used many times, while others will only be used occasionally, and in most applications, with different elements of the data base.

This paper presents a technique whereby the planning system (Haseman, 1975) will provide a capability for automatically interfacing these programs with the data base through the use of a query/extraction interface. This process will relieve the planner from the need to write the appropriate data manipulation language to perform the desired data extraction from the data base. This extension provides the planner with the capability to retrieve and modify data and use this data in various application programs without knowing the physical data structure and without using the data manipulation language (DML).

It should be noted that the cost for this added capability is that the time required to fetch the data using this general approach is greater than if specific DML was written for a specific application program. This trade-off is similar to the trade-off between higher level languages such as FORTRAN and the use of assembly language. It is likely that routine programs which will be used many times on a regular basis should be modified to include the specific DML, rather than use this query/extraction interface technique. If we are to assume that the user is not a programmer, then the automatic generating of the queries would be an important feature. With this complete system, the user will not be involved at all with the data base-application program interface.

The Data Management System used for this work was the GPLAN Data Management System (Haseman, Nunamaker and Whinston, 1975; Bonczek, Cash, Haseman, Holesapple and Whinston, 1975) which was developed at Purdue University under the sponsorship of the National Science Foundation. The GPLAN/DMS system is a partial implementation of the CODASYL DBTG report of 1971 (CODASYL, 1971). The system was designed to use FORTRAN as the primary host language, however it will also support other high level languages. Using the GPLAN system the user defines the logical structure with the data description language, and this structure is analysed and stored as a schema. The application program accesses the data base through the use of a data manipulation language. The actual data is stored within the data base. It is this interface between the application program and the data base system that we are concerned with in this paper.

2. Interface approaches

The real thrust of the interface problem is to replace the Input/Output (I/O) statements in the application program with the statements required to access the desired data from the data base. The trade-off to be considered in this process is the efficiency with which the data access will occur versus how data structure independent the code which performs this access will be. If the code must be regenerated every time the data base structure is modified, the cost to perform this regeneration is important. On the other hand, if the code is extremely inefficient, this extra cost will occur every time the code is used. The real resolution of this fixed versus variable cost problem is how often the program is going to be used versus the number of times the data structure will be modified.

There exist three basic approaches by which this program data interface can be performed automatically. These approaches are:

1. Replace all input/output statements within the application program with the appropriate data manipulation language.
2. Develop a separate specific program which will extract the data through the use of DML and store it in a file suitable for input into the application program.
3. Develop a general program which will generate an extraction file for each application program, through the use of data description (subschema) for the program.

These approaches are listed in decreasing order of efficiency and increasing order of data structure independence.

The first approach would involve modifying the application program. Using this approach, the user's program is modified to include the specific DML, which specifies the logical structure of the data base. If the data base is later restructured, each application program which uses that portion of the data base which was restructured would have to be modified to account for that restructuring. Although this approach is recommended for those applications which will be used repeatedly between data base modifications, it is not the best for the less often used application programs. This is the approach which is normally

*Assistant Professor of Accounting and Management Information Systems, School of Urban and Public Affairs, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.
†Professor of Industrial Administration and Computer Science, Krannert Graduate School, Purdue University.
This work was supported in part by Grant Number G5377-55 from the Office of Computing Activities of the National Science Foundation.

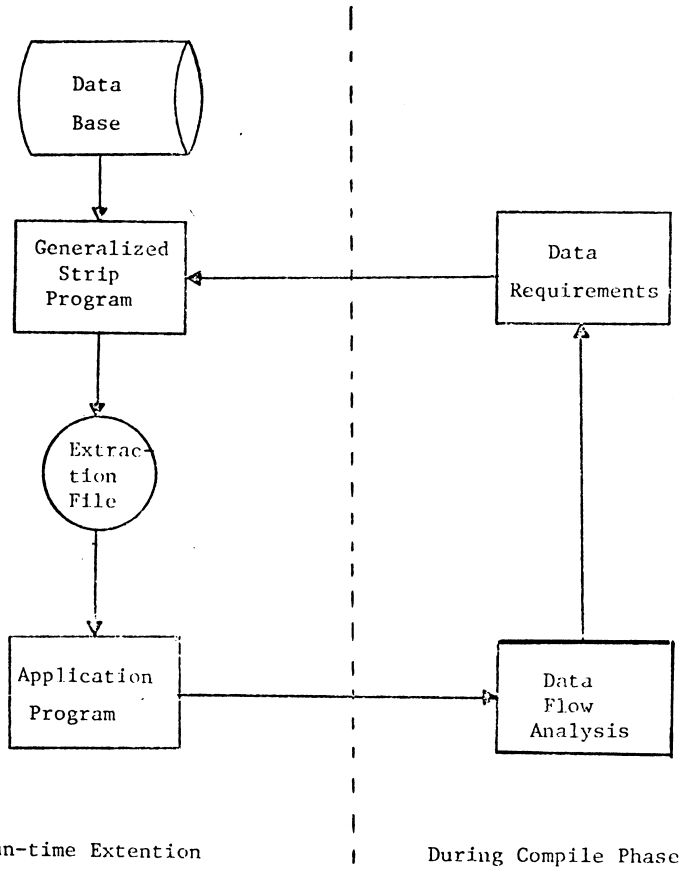


Fig. 1 Generalised strip program

used for a host language data management system, which does not include a query language capability.

The second approach would require writing the DML used in the first approach in its own data extraction program. This approach eliminates the need to modify the user's program, while also providing the capability to have several strip programs which can collect data for the same application program. This program performs the task of preparing the input file for the program in a manner similar to a user collecting the data deck to be used by the program. Although it does perform these nice extra features, the code is still data structure dependent. Although this approach varies in physical implementation, the first two approaches are conceptually the same.

The third approach involves determining the data requirements of the specific application program, and then matching these data requirements with the data base in a runtime environment. These data requirements can be determined prior to runtime and stored in the data base. They correspond to an extension of the subschema concept as proposed in the CODASYL DBTG Report. The subschema section of a DBTG program describes the logical portion of the data base that will be used by the program, and performs any renaming of item types and record types required. This description would also include conditions for selecting actual record occurrences within that logical structure. The diagram in **Fig. 1** demonstrates how this procedure is performed. This third approach was selected since it provided the maximum amount of data structure independence, and it had not been investigated in previous research. To determine the data requirements for a specific program, some sort of data flow and control analysis is required.

3. Data flow analysis

The question of looking at a program and determining its data requirements involves trying to partition the program into blocks and then determining the flow between these blocks. A lot of the early work in trying to optimise code generated by a compiler looked at this problem of determining blocks of code. The classic paper in this area was written by Fran Allen (1969) and was based in part from work by Lowery and Medlock (1967) and Prosser (1959). This work was concerned with separating a program into blocks, determining the loops formed by those blocks, and then performing local code optimisations as part of the compiling process. A recent dissertation by Nylin (1972) looked at the question of combining two program modules together by using some of the techniques presented by Fran Allen. None of these works were concerned with the question of the data requirements for the I/O statements, and in most studies, it was assumed that I/O statements were nonexistent. The following, then, is the approach we developed to determine the specific data requirements of a program.

The first step in the process involved analysing the program to determine the various blocks of code. A block of code as defined by Lowery and Medlock is:

> A block consists of a sequence of statements with the following properties:
>
> 1. Program control can only enter the sequence at the first statement;
> 2. Only the last statement of the sequence can contain a conditional or unconditional transfer of control.

The example in **Fig. 2** shows how a program would be divided into these blocks. Each immediate successor of a block is recorded, and this information is used to construct an incidence matrix. Using the following algorithm, this incidence matrix can be used to determine all the closed loops or paths within the program:

1. An $n \times n$ Boolean connectivity matrix C is constructed $C_{ij} = 1$ if block j is an immediate successor of block i, zero otherwise.
2. Successive powers, k, $(1 \leq k \leq n)$ are taken. If $C_{ii}^k = 1$ there is a path of length k containing i.
3. In order to separate out the closed paths imbedded in a given

```
1   DIMENSION CITY (5)
2   10 READ (5, 100) STATE,NOCITY
3      DO 20 I = 1, NOCITY
       READ (5, 200) CITY
       WRITE (6, 300) CITY, STATE
4   20 CONTINUE
5      IF (STATE.NE."WYOMING") GO TO 10   6
7      STOP
       END
```

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0
3	0	0	0	1	0	0	0
4	0	0	1	0	1	0	0
5	0	0	0	0	0	1	1
6	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0

Fig. 2 Example program with incidence matrix (a) Blocks for program (b) Incidence matrix

C^k, an integer distance matrix D, is kept. D_{ij} contains the length of the shortest path from block i to block j. Whenever $C^k_{ij} = 1$ and $D_{ij} = 0$, then D_{ij} is set to k.

4. A possible path, PP, is constructed whenever $C^k_{ii} = 1$ by
   (*a*) Setting $PP_i = 1$
   (*b*) For all i^* in PP
      (i) for all $j \neq i^*$ $(1 \leq j \leq n)$
      if $D_{i^*j} \neq 0$ and $D_{ji^*} \neq 0$ then $D_{i^*j} + D_{ji^*} \leq k$
      then $PP_j = 1$
      where i^* is all those i currently 1 in PP.
5. PP is added to the list of paths P if it is unique. Step 4 is then repeated for all i.
6. Steps 2 through 5 are repeated for each power of k.

NOTE: A reasonably fast method of raising C^k to C^{k+1} can be developed as follows:

1. C^{k+1} is initially zero.
2. For each entry i in C^k and for each immediate successor j to i, then $C^{k+1}_i = C^{k+1}_i \vee C^1_j$.

   Since i and j are integer indices of the table, the ORing is fast. The number of operations is a linear function of the number of branches in a program. A theorem proving this method was presented by Warshall (1962).

An application of this algorithm to the program presented in Fig. 2 is shown in **Fig. 3.** As can be seen, the two closed paths that were determined are 3–4 and 2–3–4–5–6. Although this technique is effective for small graphs ($n < 25$), the computation time grows rapidly as the number of paths and the number of nodes grow large. In one such example, where the number of nodes was 110, and the number of paths was greater than 300, the CPU time required on the CDC 6500 was about 20 minutes. In light of this, the following are two alternatives which could be used to lessen the required CPU time.

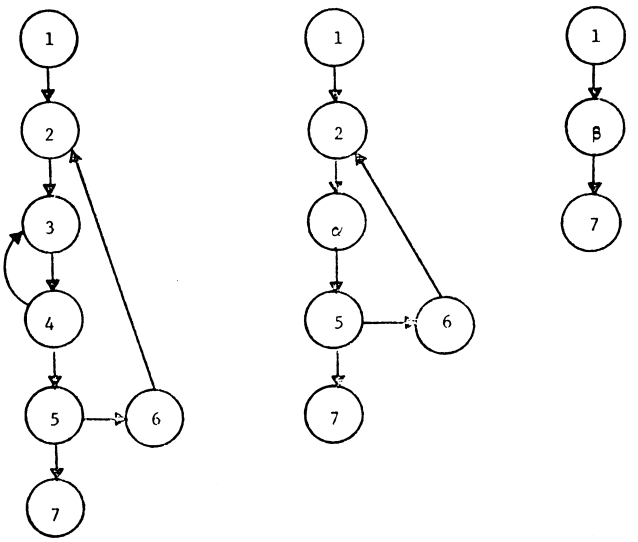
The first alternative is to try to eliminate the number of nodes in the network by eliminating those nodes which are not directly related to the I/O portion of the program. In most applications, the program inputs the required data in one section of the program, performs the calculation in the second portion, and generates the output in the final section. The section of the network which performs the calculation could be removed since it would not have any effect on the data requirements of the program.

The second alternative involves collapsing the network as various loops are detected. When a closed loop is detected, the entire loop can be replaced by a single node, assuming that this loop has the property of only one external entry and exit point. Using this approach, the network shown in Fig. 3 would first reduce by forming node α which contains node 3–4, and then by combining node 2–3–4–5–6 forming node β. This approach would rapidly reduce the size of the problem being solved as the various nodes would be combined.

4. Subschema generation

During the process of determining the various blocks of code, it is necessary to generate tables for the various statement numbers and variables used in the program. In fact, the program which performs this analysis is equivalent to the first pass of a two pass compiler. The only portion that is absent from this process is the actual machine language code generation. In light of this, although this process of data flow analysis appears to be time consuming, the same process is normally performed by an optimising compiler every time a program is compiled. It would therefore cost very little to add the necessary features to an already existing compiler to satisfy the requirements of the data flow analysis.

On the other hand, the subschema generation, which uses the



Closed Paths

3-4

2-3-4-5-6

Fig. 3 Paths for example program

```
         NAME    TYPE      SIZE START FINISH INCR
RECORD   A
ITEM     STATE   REAL      1
ITEM     NOCITY  INTEGER   1
RECORD   B
ITEM     CITY    REAL      5
QUERY    C
RECORD   A
*GROUP                             1      NOCITY 1
RECORD   B
*END
```

Fig. 4 Subschema for sample program

data flow analysis as input, is not presently performed by compilers. This process involves determining which nodes or blocks of code contain I/O statements, and which closed paths or loops contain those blocks of code. By analysing these I/O statements and the conditional statements which determine the number of times a particular loop is executed, an explicit data requirement of the program can be generated.

There are two basic components of the subschema for the program. This subschema is designed to closely resemble the subschema discussed in the CODASYL Report (DBTG, 1971). The major difference is that information describing the loops is included. The subschema therefore appears much like the data description language used to describe the data base. The first component of the language describes the record structure. Since this analysis was performed on FORTRAN programs, each READ statement generated its own record description. As can be seen in **Fig. 4,** the record description includes the item name, its type, and its size (an array). It should also be noted that when an implied DO is used in conjunction with the READ statement this has the effect of creating a group variable. The program must then record the range of this DO loop to determine the required works of input. The information concerned with the type and size of the variable can be fetched from the tables generated by the compiler.

The second component of the subschema describes the control flow information which is used to determine the number of times a particular block containing an I/O statement is executed. This component also describes the order in which the various I/O statements will be executed. It should be noted that the control information for both this component, as well as the

control information for the implied DO in the I/O statements, may itself contain a data variable which will actually be input from the data base. This is why this subschema can only be satisfied at run time, since the data requirements may depend on the actual values of the data being input.

Once this subschema is generated, the data base administrator must supply the appropriate data variable matching between the names used in the program and its associated name in the data base, if there exists a one-to-one mapping. For some applications, for example, a regression model, this mapping is not actually performed until the program is actually requested for execution. In this case, the data base administrator defines the item types as being dummy variables, and the system will request the data base names at execution time.

All of this subschema information is stored in the data base. The system owns a collection of program descriptions which contains the program name and the names of any dummy variables required to execute that program. Each program in turn owns a collection of set descriptions which contain the information in the second portion of the subschema. Each of these, then, in turn contains a collection of the record descriptions they control. The only data base dependent relationship contained in this description is the actual item type names that have been matched with the program variable names.

5. Data extraction

The final step in the process of automatically interfacing the application program to the data base involves constructing a group of queries which will extract the data required for the program. These queries can be completely constructed from the subschema information stored in the data base. These queries can then be processed by the GPLAN query system (**Fig. 5**) described in Haseman, Lieberman and Whinston, (1974) and Haseman and Whinston (1975). It should be noted that this query description is data structure independent, in that it does not specify any of the data base record types or the data base set types.

6. Conclusion

The automatic interface program described in this paper was implemented to process FORTRAN IV programs and determine its input requirements. A simple extension of these techniques would be required to analyse the output requirements so that the program could store its results in the data base. The program which performed the subschema analysis was applied to several FORTRAN application programs, and was able to complete the analysis in every case. Although the approaches mentioned to decrease CPU time required to determine loops in the code were not implemented, they would probably be required in a production type of environment.

An overall view of this process of automatic program interface is shown in **Fig. 6.** Only the processes shown in the dotted lines are actually performed in the run time environment. In actual practice, the response time for an application program which is interfaced using this procedure is not noticeably longer than if the program was interfaced using the DML commands. This difference in time, however, does become larger as the amount of data required becomes larger.

The approach discussed in this paper does have some limitations. The first limitation involves the program which inputs

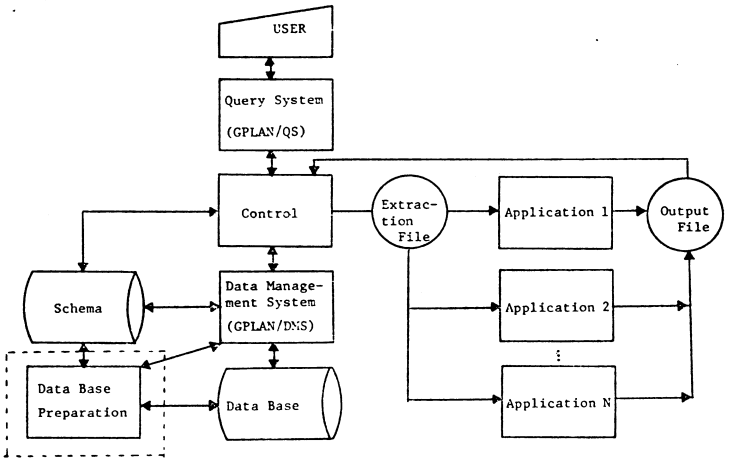


Fig. 5 GPLAN query system

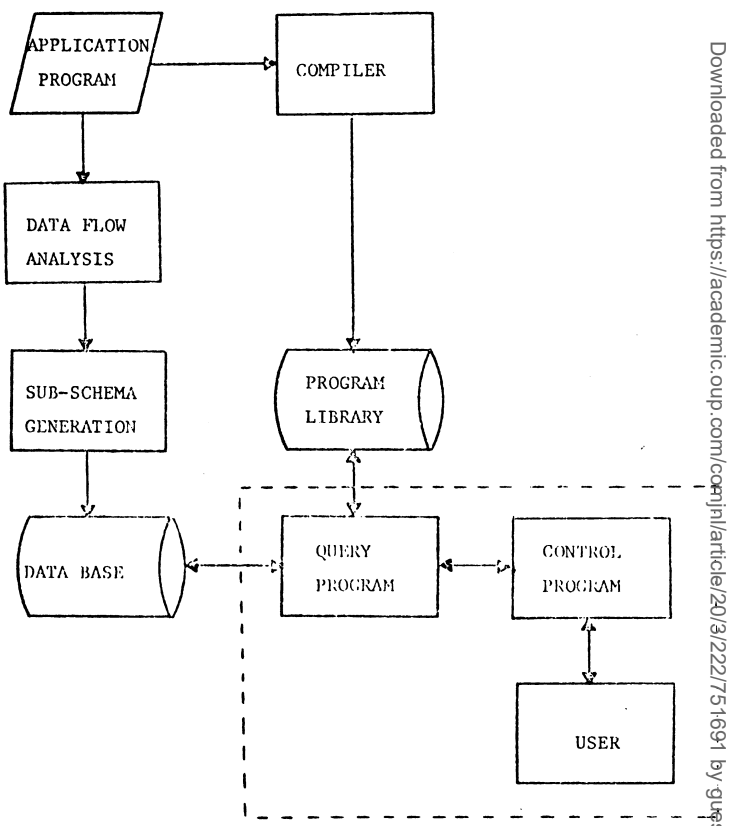


Fig. 6 View of automatic program interface

data from the card reader as well as the data base (i.e. possibly an update program). There is no possible way to predict required data requirements *a priori*, for this reason only programs which input strictly from the data base can be used. A second problem deals with programs which because of their structure contain certain logic which cannot be analysed and completely defined prior to execution. An example might be a program which solves problems using data in an interactive or recursive nature. Fortunately most application programs do not fit into the above mentioned areas, however these limitations are not insignificant.

References

ALLEN, F. E. (1969). Program Optimization, *Annual Review in Automatic Programming*, Vol. 5, Pergammon, New York, pp. 239-307.

BONCZEK, R. H., CASH, J., HASEMAN, W. D., HOLESAPPLE, C., and WHINSTON, A. B. (1975). *Generalized Planning System/Data Management System (GPLAN/DMS) User's Manual, Version 2.0*, Krannert Graduate School of Industrial Administration, August 1975.

CODASYL COMMITTEE. (1971). *Data Base Task Group Report*, Association for Computing Machinery, April 1971.

HASEMAN, W. D. (1975). Framework for a Planning Information System, Ph.D. dissertation, Purdue University.

HASEMAN, W. D., LIEBERMAN, A. Z., and WHINSTON, A. B. (1974). *Generalized Planning System/Query System (GPLAN/QS) User's Manual*, Krannert Graduate School of Industrial Administration, November 1974.

HASEMAN, W. D., NUNAMAKER, J. F., and WHINSTON, A. B. (1975). A Partial Implementation of the CODASYL DBTG Report as an Extension to FORTRAN, *Management Datamatics*, October 1975.
HASEMAN, W. D. and WHINSTON, A. B. (1975). A Data Base for Non-Programmers, *Datamation*, May 1975, pp. 101-107.
LOWERY, E. S. and MEDLOCK, C. W. (1967). Object Code Optimization, *CACM*, Vol. 12 No. 1.
NYLIN, W. C. (1972). Structural Reorganization of Multipass Computer Programs, Ph.D. Dissertation, Purdue University.
PROSSER, R. T. (1959). Applications of Boolean Matrices to the Analysis of Flow Diagrams, *1959 Proceedings of the Eastern Joint Computer Conference*, No. 16, pp. 113-138.
WARSHALL, S. (1962). A Theorem on Boolean Matrices, *JACM*, Vol. 9, January 1962, pp. 11-12.

Book review

Quantitative Methods for Business Decisions, by L. Lapin, 1976; 770 pages. (*Harcourt Brace Jovanovitch*, £9·15)

The Role and Effectiveness of Theories of Decision in Practice, edited by D. J. White and K. C. Bowen, 1976; 419 pages. (*Hodder and Stoughton*, £15·00)

At first sight these two books have nothing in common except the word 'Decision' in their titles. They are written with entirely different objectives. Lapin is 770 pages long. It aims to provide 'as complete a treatment as possible of the basic management science methodology . . . for the average college student with only an algebra background'. Subsidiary objectives appear on page 13 where the importance of a knowledge of quantitative methods to a manager is discussed. 'It is very important to know enough about this subject to guide those high-powered analysts (who often stray into a mathematical never-never land). As a bare minimum, any exposure to quantitative methods will certainly help future managers to ask the right questions and to recognise when outside help might be useful'.

The treatment is painstakingly thorough, leaving nothing to the imagination of the reader. One gets the impression that a thoroughly prepared course has been published; if so it would be a very good introductory course. The book would be valuable for any student having difficulty with basic concepts and is better than most of this genre.

Preference is given to organising topics in increasing difficulty rather than for homogeneity of subject. The 23 chapters do form a logical progression from 1, Introduction, and 2, Probability concepts, to 21, Simulation, 22, Dynamic programming, and 23, Markov processes.

The attempts made to compare techniques and place them in perspective would help the student. For example, after working the same case on a decision theoretical approach and also from a traditional statistical hypothesis the different results lead, on page 424, to a fundamental comparison of the two approaches. Problems are given at the end of each chapter with selected results at the end of the book, although there is no guidance for those in difficulty. An instructor's manual gives more detail and provides worked examples to 300 problems.

Your reviewer is unable to guess what proportion of English students would persevere to the end of the book. At some points the audience addressed would find the going hard without personal supervision.

White and Bowen have edited 31 papers given to a conference held in Luxembourg in August 1973 under the aegis of the NATO Science Committee. The book was published in 1975; the papers were given in six sections; Special theories of choice (7 papers); Prescriptive and descriptive choice (2); Practical determination of preferences, values and uncertainties (6); Information and decision (7); Problem formulation (1); and Choice of models and techniques (8).

It is tempting to divide the papers crudely into the philosophical, the computational and the practical, especially since the classification will not be subject to the rigorous scrutiny reported as discussion after each section in the book. Much would have been lost to the reader had the papers merely been rewritten to take account of the discussion.

Professor Rivett provides an elegant example of the philosophical in a paper entitled 'Behavioural problems of utility theory'. 'One is tempted therefore to use utility as a pedagogic device and to approach managers with it in the manner of missionaries approaching savages'. Examining nine basic axioms Rivett arrives at a modified form of utility which should map onto the credibility of the executive. In the ensuing discussion doubt was expressed concerning the relevance of a mathematically logical and tight set of axioms.

Basically computational seems a fair description of the paper 'Stochastic dominance: theory and applications' by Professor Fishburn. He reviews notions of stochastic dominance and their use in decision analysis. In discussion he agreed that the way he had chosen to deal with an individual's discriminatory and perfect judgement problems was probably not the best.

'Choice through dominance' is predominantly practical. Written by H. Benham of the Civil Service Department it refers to the Location of Government Exercise. One point raised in discussion of this paper was whether the solution techniques for finding the set of nondominated strategies were as simple as the author made them appear. One can always argue about the choice of papers but the coverage in this case is wide. The book can be thoroughly recommended to anyone with some knowledge of decision theory who wishes to gain a real insight into the implications of using it in practice in a wide variety of situations.

On second thoughts the books can usefully be compared by considering what a student completing the first would make of the second! A number of the papers would be incomprehensible. A few would be readily understandable and some could be read with great benefit. There is, I fear, a large gap between the end of Lapin and the knowledge assumed in White and Bowen. Armed merely with Lapin some trouble would be experienced 'guiding those high-powered analysts'.

P. A. LOSTY (Bedford)