LANGUAGES FOR DECISION SUPPORT SYSTEMS: AN OVERVIEW

Edward A. Stohr

and

Norman H. White

June 1982

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #43

GBA #82-64(CR)

Published in, Management and Office Information Systems, S.K. Chang (Editor), Plenum Press, New York, (1984), pp. 71-89.

ABSTRACT

In this paper we survey some issues relating to the language interfaces provided by DSS. We do this from the point-of-view of the designer of generalized software for building DSS. We first describe a fairly general architecture for such software. This is followed by a brief examination of the language interface components. We list the functions that must be provided through the language facilities of a DSS together with a number of implementation issues such as user-friendliness, degree of procedurality and interpretation versus compilation. This provides a useful framework for the comparative evaluation of DSS packages. Finally we discuss some possible directions for future development including specialized formal languages and the potential of English-like 'semi-natural' languages.

Keywords: Decision Support Systems; user interfaces; language facilities.

1. INTRODUCTION

The accelerating complexity and size of modern private and public institutions and their increasing dependence on environmental factors such as multi-national trade, world politics, and government regulations point to a need for computer-based decision support systems (DSS). Several surveys (Naylor,[1976] and Hayes and Nolan,[1974]) have indicated an increase in the use of such models especially in applications such as pro-forma financial statement projection and budgeting. These systems can vary widely in both scope and objectives (see Figure 1 which is based on Lorange and Rockart, [1976]). On the one hand a number of comprehensive corporate planning systems have been developed that attempt to model the firm's complete production process in time and across geographic locations. Examples are the Potlatch Corporation (Boulden,[1975]) and Xerox (Seaberg and Seaberg, [1973]) models. Such Systems may contain 50 or 60 submodels and 1000's of lines of code. At the other extreme are the currently popular micro-computer-based 'accounting spread sheet' packages that are used in small financial and budgeting applications.

Although a number of successful DSS have been reported in the literature there have been a number of failures as well. According to Boulden [1975], approximately one-third of the (larger) systems fail in the first two years after implementation and another one-third perform indifferently. Surveys by the Financial Executive Search Foundation (Traenkle et al,[1975] and by Hayes and Nolan, [1974] indicate both human and technical causes for these failures. On the

ORGANIZATION LEVEL	STRATEGIC OBJECTIVES SETTING	SPECIFIC MULTI-YEAR PLANS	BUDGETING
Corporate	Econometric models, judgemental models	Pro-forma financial statements, Capital investment models	Financial Budgeting (top-down, bottom-up)
Divisional	Industry/Local Econometric models	Pro-forma financial statements, Capital investment models	Budgeting (financial, physical)
Project or Department	----	Marketing models, R&D models, Plant/ Location, Capital investment models	Operations Management Models (scheduling, distribution)

FIGURE 1

TYPES OF PLANNING MODEL

human side there was inadequate communication between managers and developers leading to: (1) a lack of fit between the model and the organization's planning and control procedures, and/or (2) models that were too complicated for managers to understand and use. The technical difficulties included inadequacy of the hardware and software tools leading to: (1) long development times, (2) expensive use characteristics and (3) the inability of the model to evolve in response to rapidly changing needs.

Our objective is to examine how recent software advances can facilitate the development of DSS systems that are more easily used by staff personnel and managers and have increased functionality combined with an ability to adopt to changing assumptions and needs. We will do this by examining the language interfaces provided by generalized DSS software. A DSS Generator is a set of tools which can be used to build a wide variety of DSS. Some commercial examples include EMPIRE [1983], EXPRESS [1983], IFPS [1983], PLATO [1983], SIMPLAN [1983] and XSIM [1983]. Although there are particular environments that need a specialized DSS possessing features not normally included in a DSS Generator, we argue that the similarity of organizational structure and operations makes these the exception rather than the rule.

The discussion is limited to a consideration of the user interfaces provided by DSS generators. In particular we will be primarilly concerned with the language facilities provided. Other aspects of the user interface such as hardware devices, communication speed, graphics and dialogue style are covered more fully in a companion paper (Stohr and White, [1982]).

We attempt to define the language functions that must be provided by a DSS. The range of these functions is broader than in most other types of software. In addition the nature of the operations that must be performed by users is qualitatively different. A major thesis of the paper is that these two facts will necessitate the development of new languages that can manipulate quite complex data objects, processes and abstractions. At present we do not even have a coherent and broad-based terminology for the concepts and operations involved in interacting with a DSS. However a number of actual DSS's have developed a limited set of special functions at the requisite level of abstraction and aggregation.

In the next section we describe the DSS environment and derive some general software requirements. In Section 3 we describe a software architecture that serves as a framework for much of the discussion. Section 4 lists the types of language that should be made available in a DSS Generator. Section 5 discusses some of the trade-offs involved in designing DSS languages. Finally, in Section 6 we outline the major functions that have to be performed by DSS languages for data manipulation and query, model definition, model execution, sensitivity analysis and specification of output requirements.

2. THE DSS ENVIRONMENT

A DSS is used to support decision-making in ill-structured or poorly structured decision situations. A decision situation is unstructured to the extent that cause-effect relationships are unknown; there is uncertainty with regard to the possible actions that might be taken and their consequences; important variables are qualitative or immeasurable; there are multiple conflicting objectives and decision-makers can not express their trade-offs in terms of a higher level goal. The 'support' in the above definition implies that human judgement is a necessary ingredient in the resolution of the decision-making process. Thus we see a division of labor between the 'structurable' part of the process (which is relegated to the computer in the form of data base retrievals and/or more complex simulation or optimization models) and the 'non-structurable' part of the problem (which relies on human judgement). The important point here is that the DSS Generator should provide an environment that will facilitate both facets of the decision process.

We will distinguish the following classes of user of a DSS system: (1) Managers - the 'ultimate' decision-makers, (2) Intermediaries - staff personnel who interact with the DSS in a hands-on mode and form a channel of communication between the manager and the 'builder' of the DSS, and (3) Builders - technically oriented personnel who program DSS applications (build models), perform the DSS data administration function and so on. It is of course possible that all three roles might be filled by the same person. In terms of Schneiderman's [1980] semantic-syntactic model of computer users the managers can be expected to have a high 'semantic' knowledge of the application domain and a relatively low 'syntactic' knowledge of the DSS system itself. Conversely, the builders might have low semantic and high syntactic knowledge. Finally, the intermediaries might lie somewhere in between the managers and builders in terms of both syntactic and semantic knowledge.

The DSS Generator should provide a variety of interfaces and language types to suit the functions performed by all three classes of user. Moreover, since DSS use is often voluntary, the interface should not only be effective in terms of the direct benefits to the decision- making process, it should be easy to learn and remember and pleasant to use. Moreover, empirical evidence (Zmud [1979]), supports the idea that individuals differ significantly with respect to their perceptual processes and patterns of problem-solving. To support these different 'cognitive styles' we should provide a variety of interface styles and language types. Moreover we should provide 'extensible' languages where new terms and synonyms can easilly be introduced.

According to Keen [1980], a DSS is characterized by an evolutionary development process involving a three-way dialogue between the end-user(manager or intermediary), the builder and the system itself. To facilitate this process the DSS Generator should provide languages that are powerful enough to allow the rapid development of models and efficient exploration of the consequences of these models via sensitivity analyses. At the same time the software code produced should be readable and well-structured to facilitate maintenance and allow successful models (or at least certain sub-components of them) to be used in other applications or incorporated in the operational systems of the organization.

Finally, the language interfaces provided by the DSS Generator (and/or built using the tools provided by the Generator) should have the properties of any good man-machine interface namely: (1) easy to learn, use and remember, (2) forgiving when mistakes are made, (3) suitable for both novice and expert use, (4) provide immediate response for most user actions and inform the user of likely time delays on complex tasks, (5) provide a means for 'undoing' all actions that affect the system integrity. In particular it is always helpful for users of computer languages if the previously entered command is retained and presented back to the user for on-line editing to eliminate mistakes or to allow modification of the command in an incremental fashion. This saves typing and provides a useful short-term memory aid.

The array of DSS language requirements that has so far been enumerated is certainly quite imposing. However we feel that there are other aspects that get closer to the heart of the DSS design problem. Returning to the definition of DSS at the beginning of this section we see a need for the development of higher level languages more suitable to the problem-solving environment. These languages would differ from conventional languages in that they would operate at a higher level of abstraction manipulating global data objects and computational processes in a manner more closely resembling human problem-solving processes. Modern data retrieval languages are already approaching this level of sophistication. To give another concrete example, we cite the DSS Generators that support the sensitivity analysis process by providing a 'what-if' language - 'what if x is increased by 10 percent', 'what does x impact?' and etc. Other areas where there is a potential for such high level languages will be mentioned below.

Borrowing from the ROMC methodology for DSS development (Sprague and Carlson, [1982]) we obtain useful insights for DSS language development efforts. According to this paradigm we need to provide the user with: (1) alternative <u>r</u>epresentations (eg. graphic, textual, tabular) for a single concept, (2) a useful set of <u>o</u>perations to allow exploration of alternatives and the construction of yet more complex concepts, (3) a set of <u>m</u>emory aids to help users overcome their own cognitive limitations and (4) a set of control mechanisms that allow users to utilize the three preceding sets of tools. We will use this framework in our subsequent discussion.

3. AN ARCHITECTURE FOR A DSS GENERATOR

We commence with a brief description of the major software components shown in the idealized system architecture shown in Figure 2. This is based on Stohr and White, [1982] (see also Sprague and Watson, [1976]).

Data Conversion System (DCS): this sub-ystem is used to transfer data between the DSS and the external environment including the corporate transaction processing systems.

Data Base Management System (DBMS): this provides the traditional data management and retrieval functions. In a DSS environment some additional capabilities should be provided as discussed later.

Data Base (DB): the repository of DSS data - time-series, productivity coefficients, geographic locations etc.

System Directory (SD): a repository of 'meta-data' concerning the structure and contents of the data base and 'model base' as well as intelligence about how the various software and data components relate to each other.

Model Management System (MMS): this system allows procedures and models to be defined, documented, stored, retrieved, loaded and executed. It provides an environment in which more complex models can be constructed from primitive procedures and other models stored in the 'model base'.

Model Base (MB): a procedure or model library containing useful building blocks from which more complex models can be built.

User Interface System (UIS): a layer of software placed between the user and the other components of the DSS. This provides (1) 'device independence' by relieving programmers and users of the necessity to know anything about physical device addresses and characteristics, line speeds and communication protocols, (2) a uniform set of interface conventions for users, and (3) menu and screen management facilities.

Language Interface System (LIS): a layer of software providing translation (interpretation and compilation) services and message switching facilities that allow the various components of the system to communicate with one another.

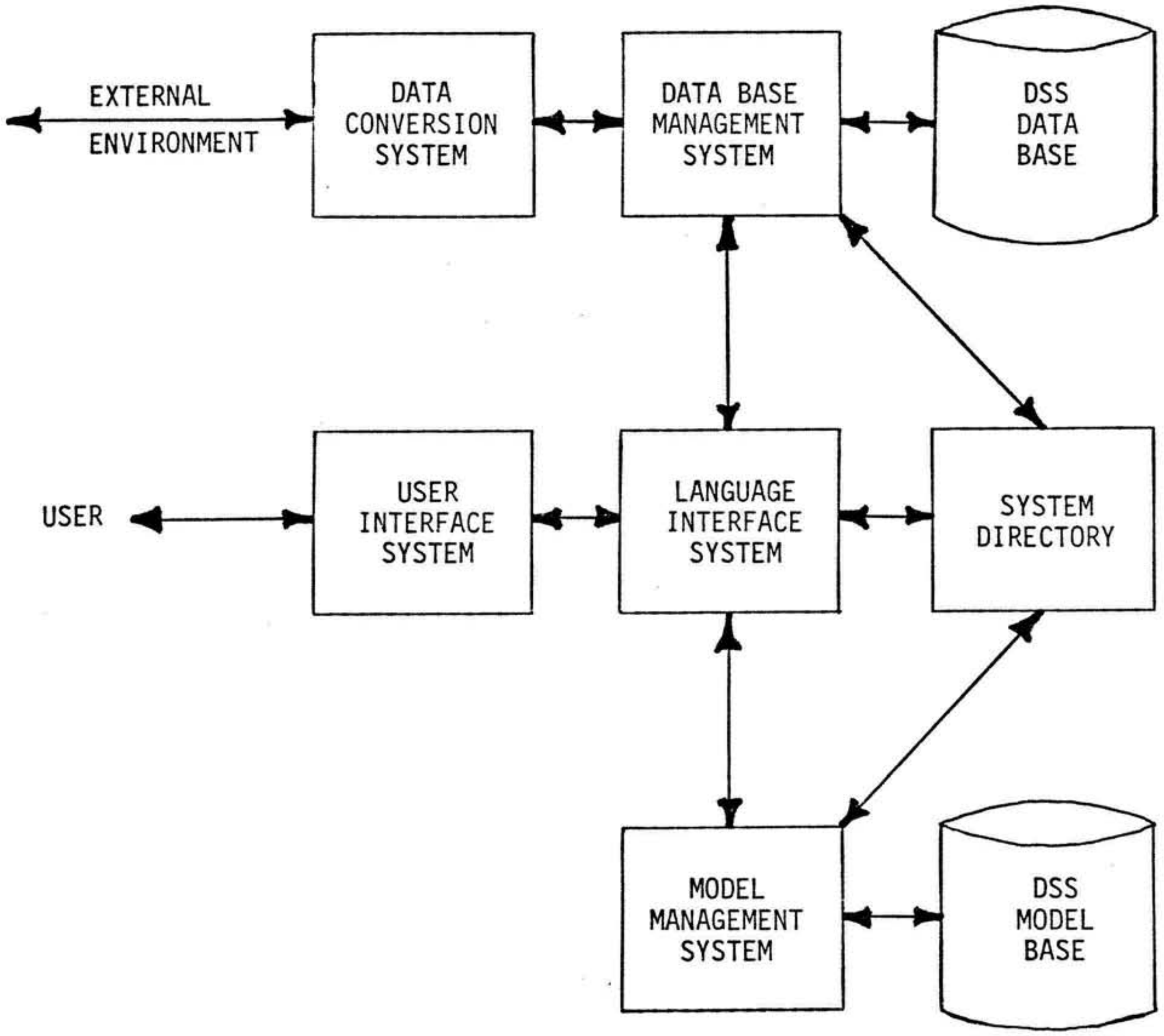


FIGURE 2

MAJOR COMPONENTS OF DSS GENERATORS

In practice the actual division of functions between the various components of DSS Generators will vary from case to case. One possible design would involve a loosely coupled system in which, for example, the DBMS and MMS have well-developed, powerful and user-freindly languages. In this case the LIS would need minimal translation abilities. Alternatively, the DBMS and MMS might support only terse mathematically-oriented 'target' languages suitable for system programmers. In this case the LIS would need to contain language translators to provide more natural languages for the end-user.

Our separation of the UIS and LIS serves to emphasize the many non-language aspects of the man-machine interface. These range from physical characteristics such as line speed and quality of CRT displays to 'help' features, graphics and menu and screen management facilities. Even the most powerful and user-freindly of languages will fail to satisfy users if these aspects of the interface are unsatisfactory (Turner et al [1982]).

A DBMS relieves programmers of many tedious and difficult data management and manipulation chores. In a similar fashion the UIS will provide higher-level languages for designing screens, menus, reports and graphic displays. At present these functions can only be provided by separate report writer, screen manager and graphic packages purchased from separate vendors. Access from available DSS Generators is difficult if not impossible. This results in inferior (e.g. prompt-response as opposed to full-screen) modes of interaction in many DSS systems. Obviously there are many advantages to the UIS concept. These include economies in number of lines of code required, reduced application development times, enhanced prototyping capabilities and the opportunity to present a coherent set of interface conventions to users.

The UIS and LIS jointly perform many complex transformations. A command issued by a user may be in the form of an English-like language, a terse formal command language or merely the touch of a light-pen on a CRT screen. Alternatively, graphics or even voice media may be used. All of these inputs must eventually be translated to activate a sequence of machine-level instructions.. Conversely, the raw output of, for example, the DBMS or MMS may be formatted into screens, transformed into graphs or formatted and paged by a report-writer. The LIS role in these transformations is to perform the language translations required.

As a more advanced feature the LIS might allow DSS builders to construct their own languages specially tailored to fit particular applications or particular user styles. To do this the LIS would contain a parser-generator (Aho and Ullman, [1978]). Note that some current natural language interfaces contain this feature to allow application specific vocabulary and grammar rules to be defined (Lehmann, [1978]).

Finally it is very important in terms of the flexibility and extensibility of the DSS that access to general-purpose languages (FORTRAN, PASCAL, APL etc.) be provided from the MMS.

4. TYPES OF DSS LANGUAGE

Figure 3 lists some applications of languages in a DSS together with the user roles with which they are most likely to be associated. Although many different languages are listed it should be clear that we are only interested in the range of functions performed and that a common syntax and style across these functions would be advantageous. We must also emphasize that the languages in Figure 3 are those made available by the DSS Generator. This list ignores both the language(s) used by the designers of the software to build the DSS Generator and the 'languages'(or 'interfaces') that can be constructed using the tools provided by the Generator. In particular the major role of the DSS Builder is to use the languages of Figure 3 to derive new languages for use by intermediaries and managers. The ease with which useful 'derived' languages can be built provides a measure of the success of the DSS Generator.

Note that most language categories in Figure 3 have both 'definition' and 'command' languages. The definition languages generally describe data objects - the schema for the DBMS, specific types of graphs, reports, screen formats and so on. However in the case of models, the model definition language (MDL) describes

LANGUAGE	PURPOSE	PRIMARY USERS
CL - DSS COMMAND LANGUAGE	High level control of DSS processes	All
DATA BASE MANAGEMENT SYSTEM		
DDL - Data Definition Language	Define data and logical relationships	Builder
DML - Data Manipulation Language	Retrieval and update from application programs	Builder
QL - Query Language	Interactive data retrieval; limited update	
MODEL MANAGEMENT SYSTEM		
MDL - Model Definition Language	Program statements defining operations on data (the 'model')	Builder, intermediary
MCL - Model Command Language	Link, load execute modules; sensitivity analysis; store and analyse results	Intermediary, manager
REPORT MANAGER		
RDL - Report Definition Language	Define report formats and data	Builder, intermediary
RCL - Report Command Language	Display reports; interactively modify formats	Intermediary, manager
SCREEN MANAGER		
SDL - Screen Definition Language	Define screen formats and data; store in library	Builder
MENU MANAGER		
MnDL - Menu Definition Language	Define menus and heirarchical relationships; store in library	Builder
GRAPHICS MANAGER		
GDL - Graph Definition Language	Define graph type, scale, axes, labels, etc.	Builder, intermediary
GCL - Graph Command Language	Display graphs, interactively modify formats	Intermediary, manager
GPL - GENERAL PURPOSE LANGUAGE	High level language accessible from MDL; extend model base	Builder

FIGURE 3

DSS LANGUAGE FUNCTIONS

procedures as well as data objects. In fact MDL's may be general-purpose programming languages with conditional branching, looping constructs and so on. Usually they include specialized language features that help in building models - for example financial functions such as Present Value and Return on Investment. They tend to be formal keyword-oriented languages because of the need for precision and expressive power. They are generally more suited for use by DSS builders or intermediaries. If a definition language is interpreted (as in the case with QL's and some MDL's) it becomes more like a command language. However the latter operate at a more aggregate, 'meta' level.

The command languages are usually interpretive and, as their name implies, they cause processing to take place identifying both the data objects and procedures to be used. As an example, the DSS command language provides interactive high level access to other DSS components. The simplest form of such a language would be a query-response interface or a menu with options allowing the user to enter various components (cause them to execute). Alternatively the DSS command language may be a keyword language. Often these have the format: COMMAND parameter1,parameter2,.... A useful extension of a simple command language involves giving users the ability to store much-used sequences of commands in 'command files' for execution on an as--needed basis. Depending on the sophistication of the system it may be possible to pass symbolic parameters to the command files to allow for variations in data names and other specifications. Finally, the DSS command language may support looping and branching constructs in which case it resembles a full programming language. A second example of a command language is provided by the Model Command Language (MCL) which allows users to execute models and perform sensitivity analyses. When an MMS is present the MCL can be quite sophisticated as discussed more fully below.

Often the statements of a definition language must be typed in a file for later use by the DSS. When these processes cannot be initiated interactively from the main DSS command interface we will call the languages 'separate'. Compiled MDL's are often separate languages in this sense. The DDL, MNDL, and SDL are also usually separate languages. Since the latter three are used only by the builder, little is lost in terms of the interactivity of the interface. However separate MDL, RDL and GDL languages can cause time delays that decrease the effectiveness of the DSS.

Any DSS Generator must have the ability to perform at least some of the functions of the CL, DDL, (DML or QL), MDL, MCL, RDL and RCL. However these may exist only in rudimentary forms. In many ways the language functions in Figure 3 together with the type(s) of interface supported provide a good way of describing both the capabilities of a DSS Generator and its ease of use. For example one popular system provides a CL imbedded in a query/response type of interface which accesses a powerful MCL and somewhat less powerful DDL, QL, RCL, GDL and GCL languages. The MDL is a separate (compiled) language and the RDL and Command File facilities are separate also. The current version of this system supports the standard (text, dumb terminal, typing) interface only. No access is provided to a GPL to allow extension of the model base.

5. LANGUAGE TRADE-OFFS

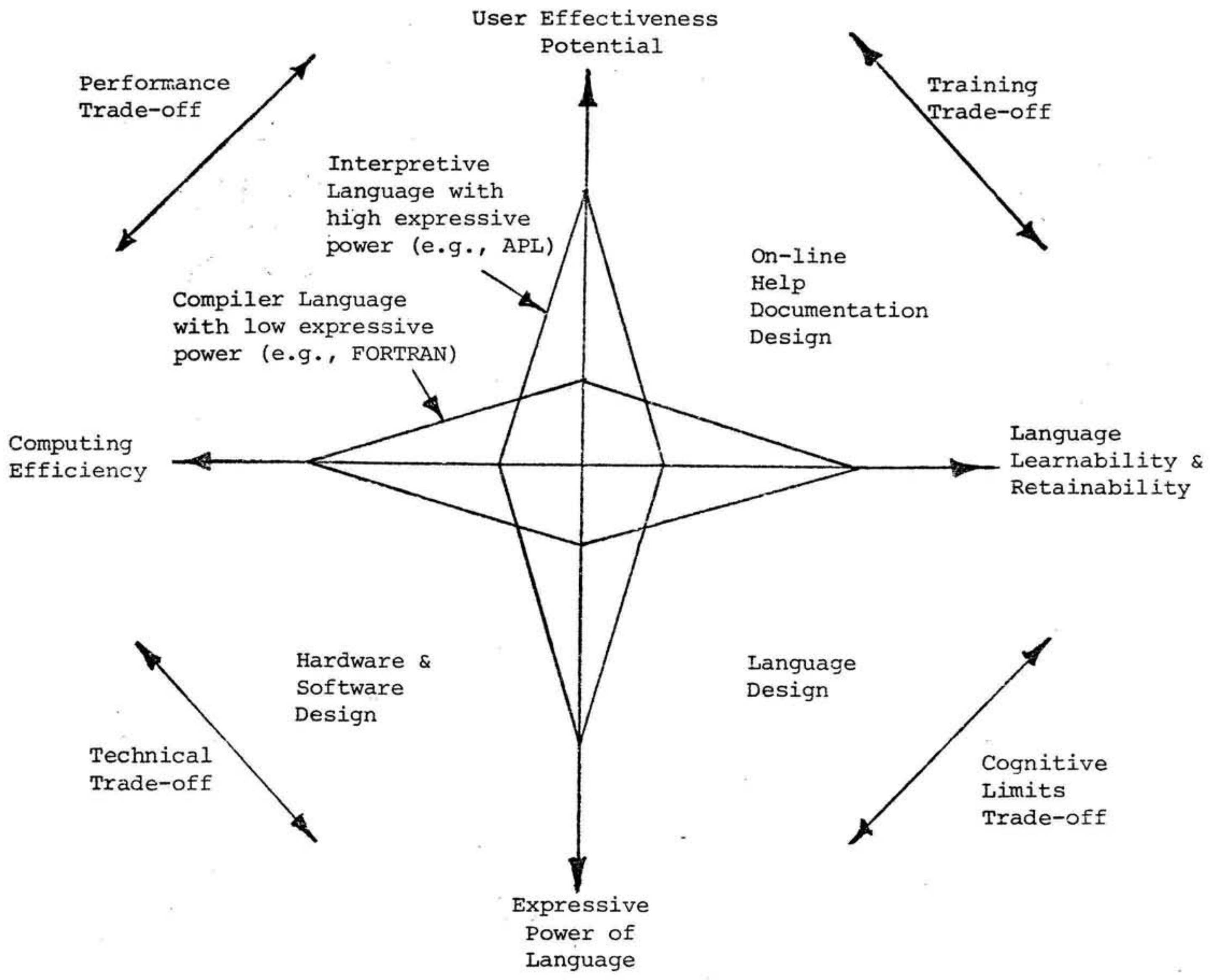
In this section we discuss some major implementation issues concerning the form of DSS languages without regard to the domain over which the languages are defined. The latter (semantic) issues are addressed in Section 6 below. Our discussion is in terms of some of the major design choices faced by DSS language designers: compilation versus interpretation; language power versus ease of learning and use; 'host-language' versus 'self-contained' systems; menu-driven versus command languages; formal command languages versus 'natural' languages.

A major consideration in DSS languages concerns the choice between interpretation and compilation. In compiled languages the definition and execution phases are separate processes and unless the transition is well-handled the interface loses the immediacy property mentioned in Section 2. Compiled software executes more rapidly but is more time- consuming to develop. Statements in an interpretive language on the other hand are executed as soon as they are typed thereby providing prompt feedback to the user and aiding the evolutionary DSS development process. Furthermore, it is often possible in interpretive systems to allow users to dynamically define new commands and to create and store new data objects. This provides some measure of language extensibility and allows users to develop more personalized systems. Finally, some systems allow users to develop and test procedures in an interpretive mode and then to translate the developed code using a compiler to obtain an efficient execution. This provides some of the advantages of both schemes.

A DSS is used to accomplish decision-making tasks efficiently and effectively. The ability to do this is closely connected to the 'power' of the languages used. A language is powerful, with respect to an application domain if: (1) it has high expressive power (operations in the domain can be expressed clearly and succinctly), (2) it has reasonable computing efficiency in terms of both response times and computing resources used. Figure 4 is a Kiviat Star diagram that illustrates some major language trade-offs and relationships. Two languages, APL and FORTRAN, are plotted on the diagram as examples. If the application involves a mathematical application such as linear programming then the shapes of the two plots might be as shown. Notice that the relative positions of the two languages on one or more axes might be reversed if another application domain were considered.

We now discuss the trade-offs associated with the quadrants in Figure 4.

Technical Trade-off: There is a tendency for languages with high expressive power to be less efficient in execution time and memory usage. Thus a well-written assembler program may be more efficient than the same program written in a high level language. Similarly


User Effectiveness Potential
Performance Trade-off
Training Trade-off
Interpretive Language with high expressive power (e.g., APL)
On-line Help Documentation Design
Compiler Language with low expressive power (e.g., FORTRAN)
Computing Efficiency
Language Learnability & Retainability
Hardware & Software Design
Language Design
Technical Trade-off
Cognitive Limits Trade-off
Expressive Power of Language

Note: Application domain and non-language aspects of the interface design are considered fixed.

FIGURE 4

DSS LANGUAGE RELATIONSHIPS

CODASYL DML (DBTG, [1971]) is likely to be more efficient than a high level relational language for most retrieval tasks. Interpretive languages are, ceteris paribus, less efficient than compiled languages.

Performance Trade-off: Often higher user-effectiveness is associated with lower computer efficiency. Usually the effectiveness of users in problem solving will be the most important factor in a cost-benefit analysis. Potential effectiveness will be increased with a language of high expressive power since the necessity for detailed programming is eliminated and prototype modifications will be faster. Again interpretive languages give higher user effectiveness, ceteris paribus, because they provide immediate feedback.

Training Trade-off: In general the languages with the highest performance potential will be the hardest to learn. Human factors studies show that on-line help and documentation aids can effectively reduce the cost and time to train users.

Cognitive Limits Trade-off: Languages with higher expressive power have both larger vocabularies and more complex grammars; they will therefore be more difficult to learn and retain. Reisner, [1981] has suggested that languages should be 'layered' in difficulty. This means that the most common functions should be expressable in simple forms that can be easily learned by casual users while less common functions can have more complex syntax.

The choice of the 'best' language for a given application depends on a cost-benefit analysis taking into account computer and training costs for different levels of user effectiveness.

The designers of a DSS generator must decide whether it is to be constructed as an extension of an existing general purpose <u>'host-language'</u> or be <u>'self-contained'</u> in the sense that all language interfaces and other functions are coded into the software of the DSS itself. The simplest form of a host-language DSS Generator would consist of a package of subroutines or procedures callable from a language such as FORTRAN, PL/1 or APL. A more sophisticated approach is to extend the syntax of one of these languages to provide DSS functions embedded in a unified syntactic scheme. Often this is done by precompiling the augmented language to produce intermediate code in the original host language. Self-contained languages are specially built for the purpose at hand and can be designed to perform efficiently and to present a uniform interface to the user. Their major disadvantage in comparison with host-language systems is that communication with other, more general purpose, programming tools may be severely restricted. This can limit the ability of DSS builders to develop new kinds of applications not envisaged by the designers of the system.

As pointed out by Vassiliou and Jarke,[1982], (and as is evident from our previous discussion), the concept of a computer language has expanded in two directions: (1) methods of communicating with computers are beginning to embrace more of our physical senses (touch, hearing, etc.) and (2) formal keyword languages are becoming more English-like and even 'natural'. Both of these trends appear to have potential in DSS applications.

Interfaces employing devices such as light-pens, touch-screens and joy-sticks fall into the first of the above categories. Often these are used to provide an alternative to typing as a means of menu selection. For this purpose they are useful interfaces for managers and other casual users. However the expressive power of such interfaces is limited to a discrete set of predetermined choices. Empirical evidence, Gilfoil [1982], indicates that users tend to prefer command languages to menu-driven interfaces after they have gained some initial experience with a system. Furthermore, the complex data retrieval and procedure coding tasks faced by DSS builders require a full language capability. Thus we see a need for both levels of 'language' in a DSS. Menu or simple prompt-response interfaces can be used at the higher levels of the DSS to allow all classes of users to select the major subsystems or modes of operation they wish to enter. These simple interfaces can then be extended downwards two or three levels to allow casual users to perform useful tasks such as running predefined models and reports. Full language capabilities must be provided for the builders and intermediaries to build the models, format output reports and so on.

The second important trend in computer languages is the attempt to make them more like English. To the extent that this can be achieved it is generally believed that managers will be encouraged to interact personally with the computer. In addition, learning and retention properties should be improved. More importantly perhaps, progress in the field of 'natural' computer languages will, of necessity, pave the way for more intelligent (forgiving) interfaces since human speech involves many short-cuts and abbreviations that require a built-in intelligence to understand. Eventually our progress in understanding natural language will be coupled with voice-entry of commands and queries to give a truly different dimension to the man-machine interface. At the present time there is at least one successful 'natural' database query language on the market (Harris, [1977]) and progress in voice recognition has reached a point where voice commands could be used for simple menu-selection tasks and restricted forms of data-entry.

To summarize our discussion so far, we believe that DSS Generator MDL's and QL's should be interpretive, have high expressive power, be 'layered' and be augmented by a range of help and documentation aids. In addition all language functions shown in Figure 3 should be accessible from the main DSS interface (no separate languages).

6. DSS LANGUAGE FUNCTIONS

We turn now to a brief discussion of the semantics of DSS data retrieval and modeling languages. We are interested in what these languages can do for DSS users or, more precisely, in the range of functions that must be performed by DSS languages. Lists of the most important functions are given (Figures 5 through 7), together with some summary comments. However it is our intention that these functions should be incorporated into the vocabulary of the various DSS languages. In some cases this can be done by simply storing callable procedures in the Model Base. In other cases, particularly for command languages such as the MCL, the functions and the objects on which they operate might form the verbs and nouns (respectively) of a DSS language. The discussion will be limited to database and modelling functions only since these are closer to the end user and are also most likely to differ from conventional MIS applications.

We will not discuss DBMS languages in great detail here since they have been the subject of many articles and books. However DSS databases have their own unique data management problems that should be reflected in language facilities. First the unanticipated nature of many requests to the DSS and the generation of new data by DSS models implies that the DBMS should allow new data relationships to be added dynamically at execution time. This facility is provided most easily by relational systems (Codd, [1970]). Secondly a number of conceptual objects and relationships occur in DSS that can not easily be represented in a conventional DBMS. Figure 5 lists some of these together with possible representations and operations (following the ROMC method - see earlier).

There are a number of aspects of DSS data manipulation that need special attention:

(1) Many, perhaps most, DSS applications involve planning and forecasting giving rise to a need for the DSS to handle time-series. There are a number of problems here: (a) conventional MIS usually maintain only recent transactions and store data on a cross-section rather than time-series basis making data acquisition by the DSS difficult, (b) conventional DBMS often do not handle variable time length data well, (c) there is often a need to store many versions of the same time series with different periodicities and ranges of time.

CONCEPT	**REPRESENTATIONS**	**SPECIAL DATA BASE OPERATIONS***
Time-series	Variable length array	Extraction from MIS or external data bases
		Periodicity Conversion
	Line graph	Missing values
	Scatter plot	Period ranging Concatenation of series Grouping of series Aggregation over series
Multi-dimensional Information	Multi-dimensional array	Aggregation Multi-key sorts
Hierarchical Groups	Trees, Confluent heirarchies	Aggregation Tree search
Meta Data	Dictionary format definitions, cross-reference maps	Multi-key, Partial-key and context search; Dictionary maintenance

*In addition to the usual retrieve, display, store, add, delete and modify operations

FIGURE 5

DATA MANAGEMENT FUNCTIONS

(2) Accounting data are often conceptualized by managers as having many different <u>dimensions</u> - for example sales-by product, by region, by customer class. Hence there is a need to be able to store create and manipulate multi-dimensional arrays.

(3) <u>Hierarchical classification</u> schemes (accounting systems, organization structures, project task break-downs, product explosions, etc.) are a common means for dealing with complexity. The DSS Generator languages must therefore allow the user to create, and manipulate alternative hierarchically organized data objects.

(4) The DSS System Directory (Figure 2) contains meta data concerning the meanings and relationships of both data and model objects. Language concepts that can be used to create and retrieve such mete data should also be provided.

Many of the representations and operations shown in Figure 5 have been provided by different systems. For example statistical packages provide facilities for handling time series and some report writers handle the concept of multi-dimensional objects well. However research on the semantic and syntactic structure of DSS languages for manipulating these concepts is much needed.

The concept of 'model management' has been an innovative feature of DSS research. We have previously mentioned that the MDL (Model Definition Language) should be (1) interpretive, (2) have capabilities (in terms of control structures and input/output) of a full programming language, (3) allow procedural access to a general-purpose programming language to allow additions to the model base to be constructed and (4) provide a number of commonly used operations useful in planning. Figure 6 indicates the major classes of functions that should be provided in the MDL. Many of these can be implemented via procedure calls. Others would be better implemented by incorporating them into the syntax of the MDL. Ormancioglu,[1982], describes a mathematically based high level language that provides a number of operations that are useful in data retrieval and model building and would allow DSS models to be stated in a concise form. Another interesting possibility is to build specialized languages containing keywords corresponding to commonly used operations such as 'Regress', 'Forecast' and 'Consolidate'. Current DSS Generators generally provide these functions only through standard procedure or subroutine calling conventions.

Turning now to the MCL (Model Command Language) there are a number of concepts and operations that can be defined over models and their inputs and outputs. Figure 7 lists these operations in three groups: housekeeping, construction/execution and sensitivity analysis.

The Model Base (Figure 2) contains a large number of procedures that can be used in models (see for example Figure 6). The <u>housekeeping</u> functions are concerned with the maintenance both of these procedures and their descriptions. The <u>construction</u> functions combine these building blocks into executable modules. While the housekeeping and construction facilities are a necessary part of any

Data Transformation Operations

Simple calculations
Vector and matrix operations
Recoding of data values
Concatenation
Missing value transformations
Grouping of time-series

Financial Functions

Net present value, return on investment, etc.
Accounting conventions
Consolidation
Depreciation routines.
Tax routines

Management Science Techniques

Mathematical programming
Simulation support facilities
Simultaneous equations

Statistical Functions

Descriptive statistics
Exploratory data analysis
Time-series forecasting
- Exponential smoothing
- Box-Jenkins

Regression forecasting
- Multiple regression
- Simultaneous equation techniques

Multivariate analysis

Sensitivity Analysis

Impact analysis
What-if analysis
Goal seeking
Monte Carlo analysis

FIGURE 6

ANALYTIC AND MODELING CAPABILITIES

CONCEPT	REPRESENTATIONS	OPERATIONS
1. HOUSEKEEPING (BUILDER):		
Functions, subroutines, and procedures as building blocks	Source code, object code, data abstractions	Store; retrieve; add; delete; modify
Model dictionary	Input-process-output description, cross-reference maps	Multiple key, partial key and context search; dictionary maintenance
2. CONSTRUCTION (BUILDER, INTERMEDIARY):		
Model built from building blocks	Source code referencing library functions and procedures	Store; retrieve; add; modify; execute (compile, link, load, go)
3. SENSITIVITY ANALYSIS (INTERMEDIARY, MANAGER):		
Assumptions	Model code	Modify
Model parameters	Data items	Modify, range values
Base case	File of data item values	Store, retrieve
Case	Operations on base case data items	Execute, store, retrieve, undo
Sensitivity	How does x vary with y?	Marginal analysis
Goals	What value of y is required to obtain x = ___?	Target analysis
Causality	What impacts x? What is impacted by x?	Impact Analysis

FIGURE 7

MODEL MANAGEMENT FUNCTIONS

application development system much DSS research has been aimed at providing more powerful systems for building models. Data base techniques have been used to coordinate the inputs and outputs of cooperating models at different organization levels (Sprague and Watson, [1976]). Knowledge representation techniques from Artificial Intelligence might be used to give the system greater self-knowledge in order (1) to assist users in learning the capabilities of the Model Base and (2) to help builders in the construction of models (Elam and Henderson, [1980]). Continuing this direction of research, Bonczek et al [1982] have shown how Predicate Calculus and the resolution principle can, in principle be used to automatically construct models from more basic building blocks. This would allow users to state information results in a non-procedural manner by specifying <u>what</u> is to be accomplished rather than the procedural details of <u>how</u> the computation is to be performed.

The ability to explore the implications of models under various assumptions concerning their structural form and parameter values and different management policies lies at the heart of the DSS idea. The MCL interface should support the decision-making process by providing short and long-term memory aids. These might help the user keep track of sets of parameter values and assumptions cases) and their corresponding results (Stohr and Tanniru, [1981]). In this context we need to develop language constructs that would allow users to create and modify 'base-cases' (trial settings of parameters and other model elements), to retrieve the results of previous runs, to compare different solutions and to modify the underlying assumptions.

In the area of sensitivity analyses we have already mentioned the emergence of languages for 'what if?' analyses. At a more advanced level Blanning [1982], defines a formal grammar for decision-making based on six functions commonly provided by a DSS - selection and aggregation of data, estimation of parameters, solution of simultaneous equations and optimization. The grammar contains four variables: a decision to be made, a sensitivity analysis, a performance measure and an environment. Sentences in the language correspond to common sequences used in problem solving tasks.

7. CONCLUSION

In this paper we developed a set of general requirements for DSS languages. We then listed an array of languages that should be included in a DSS Generator. These were differentiated both according to the function performed (example model definition or data base retrieval) and according to their type (derived, command, definition, separate, etc.). Some of the major implementation issues faced by DSS software designers were also discussed. Next we provided a detailed analysis of some of the special functions that a DSS should perform in the areas of data base and model management. For each area we described the desired capabilities from the point of view of the end user or model builder. The next step will be to design specialized, high-level languages that will assist the user in performing these functions. This is seen as an important area for research and development by behavioral scientists, operations researchers and computer scientists.

REFERENCES

1. Aho, A.V. and J.D. Ullman, Principles of Compiler Design, Addison-Wesley Reading, Mass., 1978.

2. Blanning, Robert W., 'A Decision Support Language for Corporate Planning', Int. Journal of Policy Analysis and Information Systems, December, 1982.

3. Bonczek, Robert H., Clyde W. Holsapple and Andrew B. Whinston, 'A Generalized Decision Support System Using Predicate Calculus and Network Data Base Management', Operations Research, pp. 263-281, Vol. 29, No. 2, March-April, 1982.

4. Boulden, J.B., 'Computer-assisted Planning Systems: Mangement Concept, Application and Implementation, McGraw-Hill, 1975.

5. Codd, E.F., 'A Relational Model for Large Shared Data Banks', Communications of the ACM, vol 13, no 6, June 1970.

6. DBTG: Data Base Task Group April 1971 Report, CODASYL Programming Language Committee, ACM, New York, 1971.

7. Elam, Joyce J., John C. Henderson and Louis W. Miller, 'Model Management Systems: An Approach to Decison Support in Complex Organizations'. Proc. 1st International Conference on Information Systems, Philadelphia, PA, December, 1980.

8. EMPIRE: Applied Data Research Inc., Princeton, N.J., 1983.

9. EXPRESS: Management Decision Systems, Waltham, Mass., 1983.

10. Gilfoil, D., 'Warming-up to Computers: A Study of Cognitive and Affective Interaction Over Time', Proc. Conference on Human Factors in Computer Systems, Gaithersburg, Maryland, March 1982.

11. 'Harris, L. R., 'User Oriented Data Base Query with the ROBOT Natural Language Query System', International Journal of Man-Machine Studies,Vol 9, 1977, pp 679-713.

12. Hayes, Robert H. and R.L. Nolan, 'What Kind of Corporate Modeling Functions Best?', Harvard Business Review, May-June, 1974.

13. IFPS: EXECUCOM Systems Corporation, Austin, Texas, 1983.

14. Lehmann, H., 'Interpretation of Natural Language in an Information System', IBM Journal of Research and Development, 22, September, 1978.

15. Lorange, Peter and John F. Rockart, 'A Framework for the Use of Computer-Based Models in the Planning Process', Alfred P. Sloan School of Management, Working Paper WP 860-76, 1976.

16. Naylor, T.H. and H. Schauland, 'A Survey of Users of Corporate Planning Models', Management Science, pp 927-936, Vol. 22, No. 9, May, 1976.

17. PLATO DSS Reference Manual, OR/MS Dialogue, New York, 1982.

18. Reisner, P., 'Human Factors Studies of Data Base Query Languages: A Survey and Assessment', ACM Computing Surveys, 13, 1981.

19. Schneiderman, Ben, Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers Inc., 1980.

20. Seaberg, R.A. and Seaberg, C., 'Computer Based Decision Systems in Xerox Corporate Planning', Management Science, pp 575-584, Vol. 20, 1973.

21. SIMPLAN: SIMPLAN Systems Inc., Chapel Hill, North Carolina, 1983.

22. Sprague, R.H. and E.D. Carlson, Building Effective Decision Support Systems, Prentice Hall Inc., Englewood Cliffs, N.J., 1982.

23. Sprague, R.H. and H.J. Watson, 'A Decision Support System for Banks', OMEGA Vol. 4, pp. 657-671, 1976.

24. Stohr, E.A. and N.H. White, 'User Interfaces for Decision Support: An Overview' Int. Journal of Policy Analysis and Information Systems, vol 6, December 1982.

25. Stohr, E.A. and M. Tanniru, 'A Data Base for Operations Research Models', Int. Journal of Policy Analysis and Information Systems, Vol. 4, No. 4, December 1981.

26. Traenkle, J.W., Case, E.B., and J.A. Bullard, Jr., The Use of Financial Models in Business, Financial Executives Research Foundation, New York, 1975.

27. Turner, J., M. Jarke, E.A. Stohr, Y. Vassiliou and N.H. White, 'Using Restricted Natural Languages for Data Retrieval: A Laboratory and Field Evaluation', Proc. NYU Symposium on User Interfaces, New York, May, 1982.

28. User Interfaces: Proc. NYU Symposium on User Interfaces, Graduate School of Business Adminstration, New York University, May, 1982.

29. Vassiliou, Y. and M. Jarke, 'Query Languages: A Taxonomy', in Proc. NYU Symposium on User Interfaces, Graduate School of Business Administration, New York, 1982.

30. XSIM: Interactive Data Corporation, Waltham, Mass, 1983

31. Zmud, Robert W., 'Individual Differences and MIS Success: A Review of the Literature', Management Science, vol 25, 1979.