November 1985 Vol. 64 No. 9

AT&T TECHNICAL JOURNAL

A JOURNAL OF THE AT&T COMPANIES

ANALYTICOL

An Analytical Computing Environment

AT&T TECHNICAL JOURNAL (ISSN 8756-2324) is published ten times each year by AT&T, 550 Madison Avenue, New York, NY 10022; C. L. Brown, Chairman of the Board; L. I. Christensen, Secretary. The Computing Science and Systems section and the special issues are included as they become available. Subscriptions: United States—1 year $35; foreign—1 year $45.

Payment for foreign subscriptions or single copies must be made in United States funds, or by check drawn on a United States bank and made payable to the AT&T Technical Journal, and sent to AT&T Bell Laboratories, Circulation Dept., Room 1E335, 101 J. F. Kennedy Pky, Short Hills, NJ 07078.

Back issues of the special, single-subject supplements may be obtained by writing to the AT&T Customer Information Center, P.O. Box 19901, Indianapolis, Indiana 46219, or by calling (800) 432-6600. Back issues of the general, multisubject issues may be obtained from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106.

Printed in U.S.A. Second-class postage paid at Short Hills, NJ 07078 and additional mailing offices. Postmaster: Send address changes to the AT&T Technical Journal, Room 1E335, 101 J. F. Kennedy Pky, Short Hills, NJ 07078.

AT&T
TECHNICAL
JOURNAL

VOL. 64 NOVEMBER 1985 NO. 9

ANALYTICOL

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

ANALYTICOL—An Analytical Computing Environment

By C. CHILDS* and C. R. MEACHAM†

(Manuscript received April 9, 1984)

A good workman is known by his tools.—Proverb

This paper is an overview of this special issue of the *AT&T Technical Journal* on ANALYTICOL, an analytical computing environment developed by AT&T and based on the *UNIX*™ operating system. ANALYTICOL was developed to provide specific capabilities of value to business analysts. The environment has the potential to provide substantial productivity gains and quality improvements for analysts working on ad hoc studies, and analysts and programmers developing applications. The other papers in this issue describe some of the individual software tools that make up the ANALYTICOL environment. This paper describes a set of generic tasks that need to be performed in solving business analysis problems, and how these tools can work together to perform the tasks and build an application system. An example of a typical business problem in the telecommunications industry is provided to illustrate the use of the tools in a business environment.

I. INTRODUCTION

In the early 1980s, the Bell System underwent a great deal of change, some directly under its control, some caused by external forces. The ability to respond rapidly to new business problems was of paramount importance. In response to this need, research was begun at AT&T Bell Laboratories on using the computer more effectively as a tool for business analysts performing modeling and financial studies. The

* AT&T Bell Laboratories. † AT&T Bell Laboratories; present affiliation Bell Communications Research Inc.

articles in this issue describe the individual tools that, when combined with the *UNIX* system, provide a highly flexible and evolving environment for such analytical studies.

In this overview we describe the nature of business analysis and its implications for analytical computing techniques. The scale of problems that are typically addressed by business analysts ranges from moderate to large, often with multimillion-byte data sets and equations with many variables. An example of a business problem, the modeling process that led to its solution, and the application program resulting from the work are provided to demonstrate the use of ANALYTICOL.

Analytical computing shares with many other computer applications a number of generic functions including interactive, form-oriented data entry and user-interaction management. As a result, many of the tools described here are in general use in a variety of applications. We describe how these tools are applied specifically to business analysis and the pros and cons of tightly integrated systems versus loosely integrated tools. Finally we describe the productivity gains that these tools provide when they are used as reusable modules for software application within the larger *UNIX* system development community.

II. A CHANGE IN APPROACH TO BUSINESS ANALYSIS AND SOFTWARE

AT&T Bell Laboratories and Bell Communications Research often address business problems that cannot easily be addressed by conventional approaches and therefore require development of new methods and techniques. New solutions to these problems are usually demonstrated through a prototype computer software implementation to determine their effectiveness. If the particular method is generic in nature, the successful prototypes are then given to other business analysts for experimentation. The prototype is then tailored to meet an individual company's needs, polished into business application software, and added to its repertoire of business tools. If the problem and solution are ad hoc and not expected to be readdressed, the results are reported and the understanding of capabilities needed to support such analysis is increased.

2.1 The old way

Many business analysts have limited computer experience. The conventional approach has been for an analyst to rely on programmers to obtain potentially useful data, manipulate it under the analyst's direction, and eventually code an analytic model. Or, the analyst programs in a traditional language, such as Fortran or COBOL, to develop models directly. Frequently model changes or "what-if" modeling studies are difficult and time-consuming to implement in the

typical "waterfall" model of software development. (For example, it has taken over six months for some tax changes in depreciation to be included in one capital asset analysis program.) Even if the time frame is acceptable in a business sense, these resources can be justified only for software that has major effect on or repetitive use within a corporation. Exploratory interactive analysis or ad hoc studies cannot be done effectively in this type of environment.

2.2 The new way

New analytical application systems can be built far more productively by rapid prototyping with appropriate tools. If analysts are given an environment where they can directly use a computer and high-level languages—frequently referred to as fourth-generation languages—then many more exploratory and ad hoc studies can be undertaken, and understanding of the decision-making process can be increased. But how is it possible to bring about fundamental change in the way business analysts approach their tasks? First, it is necessary to create an alternative. We propose ANALYTICOL as an alternative.

The work of ANALYTICOL began when an expiring mainframe lease created the opportunity to develop a new analytical computing environment within an AT&T organization containing both business analysis and computer science research activities. Development began with a fundamental decision to use the *UNIX* operating system to support the environment. Another critical decision was to provide sufficient hardware, including high-quality graphics printers and terminals, high-speed communications, and necessary computing power in the form of super minicomputers and work stations networked together.

In the beginning stage of the research, the proximity of the computer scientists to a willing test population of business analysts provided early identification of system requirements and rapid information feedback. This contributed substantially to the selection and character of the tools and the quality and usability of the software.

III. CURRENT STATUS

Use of the ANALYTICOL system within AT&T is growing both through demonstration of the productivity gains by analysts and application developers and by word-of-mouth reports by enthusiastic observers. Each of the ANALYTICOL tools exists either as a working prototype or as a fully supported tool, and each has been effectively applied to several problems. The tools are implemented in the portable *UNIX* operating system environment and have been executed on a variety of hardware including work stations, minicomputers, and mainframes under *UNIX* System V, Releases 1 and 2, and the Uni-

versity of California Berkeley Releases 4.1 and 4.2 BSD. To use all of the tools effectively, a typical configuration has at least 2 megabytes of main memory and 20 megabytes of disc memory plus high-speed tape drives. Because of the full-screen nature of some of the terminal interfaces, 1200 baud or higher terminal access is recommended.

In addition to the basic ANALYTICOL environment that we describe here, many specialized analytical environments also exist that include ANALYTICOL tools, commercially available database managers and modeling systems, and data filters translating data formats between tools. Section V describes a specialized tool that was created with ANALYTICOL tools to analyze pricing options for telephone services in a marketing analytical environment.

IV. THE ANALYTICOL APPROACH

The computing model selected for this environment was a set of independent, high-level, efficient, and functionally specific tools addressing generic analytical tasks that could be loosely integrated into more specialized application packages. This model enhances the user's tool development many times over by creating reusable software. In the *UNIX* system environment, capabilities such as redirection, pipes, and the shell language facilitate this loose integration. Other *UNIX* system commands extend the environment, especially when ASCII flat files form the data interfaces. Furthermore, the *UNIX* operating system provides users with flexibility in the selection of terminals (Termcap/Terminfo) and computers. This is important because, as tool developers, we cannot foresee who our eventual users will be or what their hardware environments will be.

Determining functionally generic tasks common across a variety of ad hoc studies took the combined and concerted efforts of a group of computer scientists and a group of business analysts. It was the close interaction of these two groups that enabled the identification of a set of generic analytic tasks that could be addressed by a small set of tools which, in turn, effectively could be used to solve a large set of problems. The five generic tasks that were identified, and the tools that address them, are described in the rest of this section. Figure 1 shows the data and work flow among the tasks.

4.1 Data acquisition

The first step in exploratory data analysis is typically the acquisition and preparation of data. Usually data from many sources—often not computerized—must be integrated to provide the foundation for analysis. Examples of sources of data are previously compiled operations data about customer calling patterns, cost data for providing a service using new and existing technologies, demographic data from the cen-

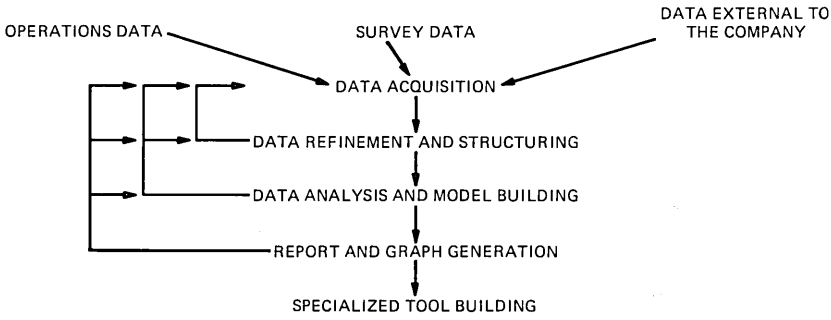

Fig. 1—Flow of an analytical project.

sus, and customer survey data on types of service selected. Two papers in this issue describe tools that provide this data-acquisition function. The paper by Belanger and Kintala describes two tools for data acquisition based on new program-generation techniques. These tools, TTU (Tape to *UNIX* system) and IMX (IMS to *UNIX* system), address the problem of transferring data residing in large mainframe databases to *UNIX* systems. TTU takes a description of an IBM-formatted tape and creates an efficient C language program that reads the selected records and files from that tape onto a *UNIX* system file. IMX takes a description of an IMS database and creates a COBOL program. This program can be transmitted by remote job entry or other methods (e.g., the *UNIX* system command `co`) to the system running IMS and then executed, with output data returned to the *UNIX* system environment by a route appropriate to its size.

The paper by Prichard explores the use of editors for forms entry and management. With FE (Form Editor) an analyst can paint a form and describe attributes of data to be collected, including intraform data validation and computation. For those more proficient in programming, fields can be filled by *UNIX* system commands (e.g., `date`) and user-supplied C language functions. When selecting FE to do data management, a file cabinet model is provided. Forms, as objects, can be manipulated by the user with a command language similar to that of the `ed` file editor for actions such as insert, append, move and delete. When a form is displayed, the `vi` full screen editor is the model for cursor positioning and data entry and update. Both features were designed to build on *UNIX* system editor training. A less powerful full screen editor is also available for users who do not know `vi`. For applications that need to interface with another data manager and need only the interactive form display and editing features, a C language library exists.

4.2 Data refinement and structuring

The *UNIX* operating system file system is adequate for much of the data storage and retrieval needs. All of the data-acquisition processes mentioned above provide flat *UNIX* system files (with the capability of building implicit hierarchies). White space and new lines are enough to meet minimal structuring needs, but some problems require hierarchical or relational structures and query capabilities. Two tools described by Swartwout and Yanofchick have been created within ANALYTICOL as experiments in database management aimed at analysis.

The paper by Swartwout describes Datastream, a simple language with expression handling, control structures, and built-in I/O. It makes it reasonable to query large (multiple megabyte) flat files interactively for data selection, refinement, and computation.

The paper by Yanofchick describes a hierarchical data manager, T. One of its most interesting capabilities is that a node can consist of a shell script. Access to such nodes can cause programs to be executed whose outputs appear as virtual data in the hierarchy.

4.3 Data analysis and model building

Data analysis and model building are the essence of the analyst's work. Such work requires a diverse and flexible set of functions and may include the interactive application of numerical or statistical procedures and exploratory graphical techniques. It may also include iterative development and solution of a series of equations—some of which may be simultaneous sets—that model financial or corporate problems. Often the analysis has multivariable equations. Two tools are available to meet these problems, S and HEQS.

The paper by Becker and Chambers describes S, a language and system for interactive data analysis. S provides a comprehensive set of statistical and graphical functions for interactive use, and mechanisms for users to extend its functional capabilities and language. Because its data structures are rich, yet easy to manipulate, it can also serve the data-refinement and structuring needs of many applications.

The paper by Derman and Sheppard describes HEQS (Hierarchical Equation Solver). This tool provides algebraic and logical expertise normally implicit in human model solving. Users can define, debug, change, and interactively solve nonprocedural models described as sets of linear and nonlinear multivariable equations. It provides for what-if, goal-seeking, impact analysis, and sensitivity analyses.

4.4 Report and graph generation

Reports and graphs are important not only to summarize and document the results of a project, but also for information updating

during the analysis stage. A report describing the project and its conclusions must be produced efficiently and with high quality. Often it is a mixture of text, graphics, filled-in forms, and tables. Many tools support this need, including FE, S, D, and *UNIX* system commands (e.g., `mm` and `nroff`). D (Display system for reports and graphs) is part of ANALYTICOL and is built on the S graphics capabilities. It provides a nonprocedural language for describing tabular reports and business graphs (pies, bars, and two-dimensional graphs). Although its capabilities are similar to those of many Database Management Systems (DBMS) that create reports and graphs, its principal strength is that it is an independent tool and offers output flexibility advantages to applications and studies not otherwise using a DBMS. Since it represents commonly available technology, no further mention is included in this series.

4.5 Specialized tool building

If the data analysis and model do not address an ad hoc need (or the solution can address a large set of problems and therefore is no longer ad hoc), then the analyst has reached the point where the analysis and model solution are known and ready to be "packaged" into a specialized tool for use by others in the future. In other words, the requirements are now understood and could be documented, and application software should be created.

However, by using ANALYTICOL in the prototype stages, major pieces of the application already exist. They need to be integrated smoothly. A user interface needs to be created that hides the individual tools and intermediate steps and places the user in control of his/her work environment. The last paper, by Vo, addresses this problem.

The Interpretive Frame System (IFS) provides a block-structured, high-level programming language for specifying both the structural information of a system and the interactions between its various subprocesses, coprocesses, and human users. The application builder can create polished user interfaces (menus, question/answer, forms, help, and other dialogue) for assortments of tools from scripts and, thus, separate program function from user interface in a consistent way. To end users, the application appears as a single program allowing them to analyze their problem within an interactive structured environment. Some users have described it as an application-specific shell.

V. AN EXAMPLE

In this section, we give an example of using ANALYTICOL to solve a problem. To put things in perspective, we briefly describe what the analysts are trying to do, and then we discuss the ANALYTICOL

tools that help in developing the model. Finally, we show how the applications developers can use the ANALYTICOL tools to build a system that contains this model for others to use.

One practical business problem is to estimate the profitability of services under alternative pricing scenarios. Analysts want to understand what will happen if alternative pricing plans are available to customers for certain types of telephone usage. In our example, the analysts investigate optional calling plans that feature discounts on intrastate or interstate direct-distance-dialed calls placed in the night or weekend period. A customer who elects an optional calling plan "buys" into the plan each month and pays for toll calls according to a discounted rate schedule.

To assess the profitability of the plan, the analysts model the demand response to the plan, as well as any potential changes in the calling behavior of all customers to whom the plan is initially offered. To do this, the analysts obtain pre- and post-usage measurements on the customers. Since this approach involves many usage measurements and large customer populations, the method of data management is an important consideration.

The analysts use two ANALYTICOL tools that are helpful in manipulating large amounts of data: TTU and Datastream. The TTU tape-reading facility can read at high speed the 35- to 80-megabyte data files that contain the monthly customer usage data collected on mainframes. TTU reduces the read time of this data to one-half hour from the roughly seven hours that other facilities would take. And it also provides checking, extracting, and summarizing of the raw data while it is being read from tape. The second tool the analysts choose is Datastream, which they use for more complex extraction, computation, and general clean-up for data residing in the very large flat files created by TTU. Datastream also is used to provide a link between individual customer-level information located on various data files. A master file is created, with one record per customer, of pointers to that customer's information in each of several data files.

S is used to generate graphs for graphical analysis to identify and analyze subscriber probabilities for the different calling plans. It helps determine empirical curves and to fit the curves with different functional forms, using both linear and nonlinear methods. This becomes the working model of customer choices.

The analysts then tailor the model for their use in predicting customer reactions to other alternative pricing options. For this they create an FE form on which to enter a pricing option. They add financial equations with the equations of the customer choice model to do revenue and profitability studies. HEQS is applied to the equations of the total model and the pricing options being studied in order

to solve the profitability model. They use the what-if and goal-seeking capabilities of HEQS to explore other scenarios, and they use the sensitivity analysis capability of HEQS to study how customers might react to different pricing schemes and thus help predict the profitability of the different options. Finally, they document their results in reports and graphs using D.

As a result of their work, the analysts now have a customer-calling pre/post database, a customer-preference model, a form to enter pricing options, a model of the financial equations, and a model of the reports and graphic output displays. And, in the process, the analysts have developed a predictive model that can be tailored by an individual telecommunications company to meet its specific needs. What is needed is an application system to be used by other market analysts in studying future pricing options.

A key point is that the only additional programming needed to build this application system is the end-user interface and on-line assistance (help) and the reformatting of the data between FE, HEQS, D, and the database. IFS integrates the tools and data interfaces and provides the end-user menus and help messages; the data formatters are programmed in shell or C language. The only interactions an end user must make with the resulting application system are entering the pricing plan on a form and executing the tailored system by choosing appropriate paths through the system via IFS menus to meet his or her needs. The new optional pricing tool is now added to the tool set making up the marketing analytical environment.

VI. INTEGRATION, ARCHITECTURE, AND FUTURE DIRECTIONS

The computing model for ANALYTICOL is a set of independent, functionally specific tools that can be used as building blocks, with the *UNIX* system and IFS providing the glue to bring them together. This model requires that the analyst or application builder be able to convert data from one data format (output of x) to another (input of y), as described in the example in Section V. Taken to the extreme, if there are n tools, this model may require up to $n(n-1)$ data filters. Furthermore, the application builder must determine an architecture and sometimes create software for some of the manual steps the analysts may have taken. A tightly integrated ANALYTICOL system could avoid this.

To experiment with tight integration, S, HEQS, FE, and D were combined using IFS as a menu/help interface and the utilities in S for making HEQS, FE, IFS, and D appear as S functions. The tools themselves were not modified. IFS and S became communicating coprocesses and the S data structure became the common data structure by which all tools could communicate (i.e., since the tools were

not modified, the model for n tools would require at most $2n$ filters). Obviously there was overhead, but the performance remained in the acceptable range for interactive studies. For only occasional S users, there was the added advantage of a menu interface to the more than 300 functions, plus the capability to learn and switch into the S command mode whenever desired. A small amount of FE functionality was lost since some form layouts cannot be described in S data structures.

Although some analysts like this integrated tool and have used it in studies and applications, it was not as popular as expected. The S data model is not a natural one for thinking about forms or equation modeling. Unless all the functions are required for a problem, many analysts prefer loose integration, using the tools independently and specializing their data filters to meet their singular needs. Therefore, the experiment has met only limited success. A new experiment in tight integration is under way to define and build the tools around a new common data structure that fits this set of the generic tasks in a more natural way.

Furthermore, while greater integration may be preferred for some analytical problems, experience shows that many users use only one or two tools for any specific problem. Generic filters that bridge tools commonly used together in the extended *UNIX* system environment have been developed by some users, and are beginning to be shared and added into the ANALYTICOL environment. This is at least a short-term solution to redundant filters and to further reduce the need for programming. It has the advantage of keeping the environment open to new tools developed by anyone, anywhere. It is therefore more natural to the *UNIX* system.

The architecture of a project can have a major effect on its ability to use tools as components. One architecture that works particularly well is that of independent processes whose executions are controlled by a common process. Communication passes dynamically between each process and the control process and through shared data files. The order of execution of the application's functions is controlled directly by the end user responding to either results of previous functions or external needs. This is the architecture of the example in Section V, and the one for which IFS was designed.

The ANALYTICOL environment will continue to evolve. In addition to exploring questions about integration and architecture, work is needed to standardize common elements among tools, such as error handling, style of error messages, common commands usage, and common editing techniques. Other future work will extend the environment to include new tools and emerging technology such as database machines, distributive computing, advanced work stations, bit-

map graphics, and user interfaces, through pointing devices such as mice, fingers, and light pens.

VII. EFFECT ON PRODUCTIVITY AND QUALITY

We have described a process of analytical computing, and an environment of software tools that were created to provide for more effective use of the computer and to increase the productivity of analysts involved in business and financial studies. But, how do you measure increases in productivity?

Some statistical measures exist that can be applied to software, such as lines of code per programmer months. But since the languages are very high level (more power per line) and applications are done in less time, this measure is not very useful. The measurement of errors per lines of code can show increased quality for an application when the tools' source code is included in the denominator, since the tools have been used extensively and introduce fewer errors in the numerator. But statistical measures do not seem to capture the benefits effectively. Since there have been no controlled studies, we can only offer the following observations.

For two analytical application systems, resources were estimated for the old way and compared to resources consumed in the new way. Details of the applications resources leading to these conclusions are provided in Table I. In both cases the productivity gain—about a factor of 5—led both projects to conclude the benefits were substantial. (The appearance of the value 5 in both applications should be considered only coincidence. The similarity of lines of code per staff-month for either approach within each application is indicative of the application complexity.)

In a recent survey of applications developers using FE, IFS, and D—these three tools offer more generic functions and have had

Table I—Experiences of two analytical applications

	Old Way (After Req. Known)	New Way (Evolving Req.)	Productivity Factor for Staff Months
Application 1. Supporting Rate Planning			
Linear months	7	1.5	
Staff months	21	4.5	
Lines of code*	50,000	10,000	>5
Lines/staff month	2,333	2,222	
Application 2. Financial Analysis of Business Plans			
Linear months	8	2	
Staff months	32	6	
Lines of code*	10,000	2,600	>5
Lines/staff month	313	433	

* These figures only include code written specifically for each application.

significant use outside analytical computing—85 percent showed major productivity gains and 40 percent considered these tools critical to the success of their projects.

Benefits not directly associated with the code development process are even more difficult to quantify. Fringe benefits, such as real-time validation of data entry that would not otherwise have been provided, have contributed to improved user interfaces and decreased user error. Such features were not a project requirement and did not enter into the benefit analysis leading to the selection of the tools. However, both types of benefits have improved the quality of the application and have led to increased user satisfaction.

A tariff for a new service was based on a study of five alternatives. To what extent did the study, which would not have been done without the tools, contribute to a better decision? This type of benefit is very difficult to quantify!

We feel that the environment of ANALYTICOL has been successful in increasing productivity of business analysts and application developers. We base this conclusion primarily on their acceptance and use of the tools. We invite you to read the next seven articles in this journal, which describe the generic tasks and features of each tool in more detail, and reflect on our conclusion.

VIII. ACKNOWLEDGMENTS

While we are grateful to many people who have contributed to ANALYTICOL and this paper, we especially want to thank J. P. Downs for taking the risks and providing the leadership and resources, and Lynn Grala for sharing her experiences in the rate studies example.

AUTHORS

Carolyn Childs, B.S. 1965, M.S. 1966, (Mathematics), University of Massachusetts; M.S. (Computer Science), 1974, University of Wisconsin-Madison; Instructor of Mathematics, 1966–1975; Assistant Professor, 1976; University of Wisconsin Center System, Waukesha; AT&T Bell Laboratories, 1976—. Ms. Childs is currently Supervisor, Common Software Tools Group. At AT&T Bell Laboratories she has contributed to methods and systems supporting analysis for project evaluation and economic impact of new technology. Her involvement in the creation and support of the analytical computer environment reflects her research interests in reusable software, tools and development environments. Member Phi Beta Kappa, Phi Kappa Phi, IEEE Computer Society.

C. Rebecca Meacham, B.S., (Mathematics), 1969, Millsaps College, Jackson, Mississippi; M.S. (Mathematics), 1973, The University of Mississippi; M.S. (Computer Science), 1979, The University of Tennessee; Mathematics Instructor 1969–1979; Member of Technical Staff at AT&T Bell Laboratories, 1979–

1983; Bell Communications Research, 1983—. Ms. Meacham is currently a Member of Technical Staff in the Analytical Computing Systems District at Bell Communications Research. Upon joining AT&T Bell Laboratories in 1979, she helped create a software development environment in which analysts can create and experiment with new modeling techniques, participated in the experiment that fully integrated the ANALYTICOL tools, and developed several prototype application systems using ANALYTICOL tools.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

FE—A Multi-Interface Form System

By R. M. PRICHARD, JR.*

(Manuscript received April 9, 1984)

The Form Editor system provides visual "form" and "menu" capabilities for applications based on the *UNIX*™ operating system. It offers many features usually not found together in other systems that perform similar functions. For example, it includes a program-level interface library, an executable component for data collection, and a multi-hardware/operating system environment. Because the software and language interfaces are simple and require minimal programming background for most applications, a significant reduction in system design and development time is possible. This paper discusses the capabilities and implementation of the Form Editor system.

I. INTRODUCTION

Electronic forms[1] are used in many programming applications to produce software systems with good end-user interfaces. A "form" is an image of a printed document with video attributes or characters representing the locations of required or requested information. Such forms generally fall into one of the following three classes: (1) control, (2) report, or (3) data collection.

Control forms (Fig. 1a), often referred to as control frames[1] or menus,[2] provide processing control by allowing a user to enter a letter or number that corresponds to the displayed list of allowed processing options. The program, on receiving the selected request, can respond in many different ways. For example, it can display a different control menu, report, data-entry form, or message, or prompt the user for additional input.

* AT&T Bell Laboratories.

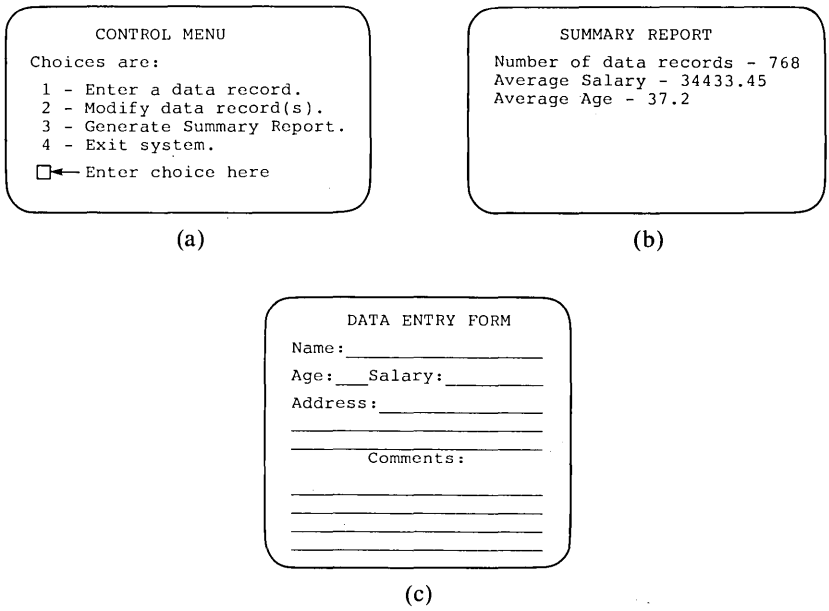

Fig. 1—(a) Sample control form. (b) Sample report form. (c) Sample data collection form.

Report forms (Fig. 1b) can contain descriptive text and data (processed or unprocessed) and are static displays. To change the data on the report, the user changes report input data elsewhere, and regenerates the report.

Data-collection forms (Fig. 1c) can require significantly more interaction by virtue of the amount and types of data that must be entered or changed. These forms can exceed the horizontal and vertical dimensions of the user's terminal and the form fields may need to be randomly accessed, validated, and/or computed. In addition, advanced text editing capabilities such as character insert, delete, or overstrike may be required.

What should be noted at this point is that the capabilities that must be provided by application software to support the possible user interactions with forms can vary significantly. Criteria have been defined for designing forms[3] and several forms-management languages and systems[4–6] have been described. In these reports, emphasis is placed on the generic operations on forms within office-automation systems and the interaction between the system users and the forms is barely addressed.

The FE system was developed as a programming tool for *UNIX* operating systems that need to support form-oriented user interfaces. Though it was initially developed to support data-collection applica-

tions, the system has been used to support all three form classes within an application.

The FE system is easy to integrate into new or existing application software and the form-definition language is simple and supports many capabilities required by form-oriented applications. It consists of an executable system component (Executable FE) and a library of C programming language[7] routines. The library provides terminal screen management and field manipulation routines. Support for CRT as well as non-CRT terminals is provided via the Termcap[8] or Terminfo[9] databases, and "hooks" to standard *UNIX* operating system utility- and application-specific programs exist to provide capabilities not directly supported by the FE system.

II. CURRENT STATUS

The FE system (version 3.1) is currently used in more than 100 applications and is available at most *UNIX* operating system-based computers in AT&T. The software currently runs on Digital Equipment Corporation's VAX* 11/750-780 and AT&T's 3B processor line under AT&T *UNIX* System V or *UNIX* 4.1/4.2 BSD operating systems. It has also recently been installed on the AT&T *UNIX* PC. The FE system (both the library and executable FE) is written in C with the executable component requiring about 120K bytes of memory and the library requiring 85K bytes.

The FE system has been used for a wide variety of applications in AT&T. Most applications (about 80 percent) are developed by individuals who are not software experts, but engineers, secretaries, and managers with some familiarity of the *UNIX* operating system. The complexity of these applications varies greatly. For example, several recruiting and professional society membership databases are applications that require only FE and a few shell program filters, whereas an automatic test program generator for integrated circuits integrates FE with many other system components. It has also been used as a report generator and to provide simple spread sheet-like capabilities for several applications.

The FE library has been used for those applications that require rigid control over system processing and/or data collection. Examples of typical applications that currently use the library include a field-oriented visual editor, front ends for several database management systems, and a budget management system. The library has been integrated into several commercial systems currently under development and is being considered as a standard software development tool in several AT&T Bell Laboratories computing centers.

* VAX is a trademark of Digital Equipment Corporation.

III. THE FE SYSTEM ARCHITECTURE

The executable `FE` system module operates as an editor in which the unit of reference is a form instead of a line or record in text editors such as `ed` or `vi`. Instead of inserting or deleting records in the data file and changing characters in a record, the user inserts or deletes forms and changes the fields contained on a form. Each form is described in a Physical Form Definition File (PFDF) and sets of PFDFs can be combined or concatenated by Logical Form Definitions (LFDs).

The executable module actually consists of two executable programs, as shown in Fig. 2. `FE` provides the end user with the capabilities provided by the `FE` library routines and the `FEio` module provides the underlying database management functions.

`FE` is usually invoked from within a shell program that passes the name of the data file, the path of the PFDF directory, the name of a file that contains the LFDs, and other optional arguments. These arguments can be used to custom tailor the `FE` system for an application. `FE` internally activates the `FEio` module and establishes co-process communication using pipes over which simple commands are sent to instruct `FEio` to perform such functions as data searching, retrieval, and update. `FEio` responds to these commands by returning a command status or one or more data records.

There are two advantages to this type of interface. An application can provide its own data interface program when the format of the `FEio` data file is inappropriate, and `FEio` can be used as a concurrent process by other application software components to provide a direct-access interface to executable `FE` data files.

The executable component of the `FE` system functions much like a filing clerk with the responsibility of maintaining the forms in one or more filing cabinet drawers (`FE` data files). For example, executable can be instructed to place a new form at a specific location in the file or search the data file for a form to be "pulled" for review or changes.

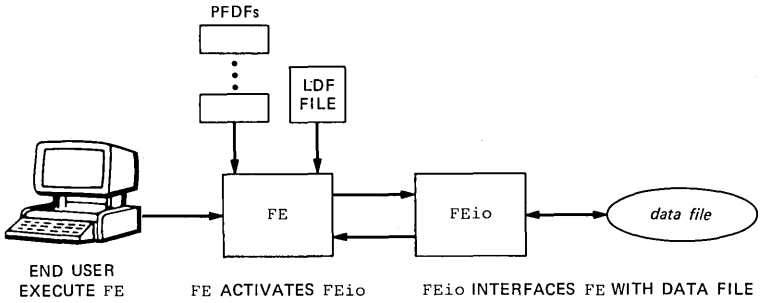

Fig. 2—End user, `FE`, and `FEio` interrelationship.

The FE data file can contain many different types of forms (logical forms), each of which may be referenced by its relative location from the front, or top, of the file or by its associated name. As forms are inserted into the data file, the size of the file increases; as they are deleted the size decreases.

IV. LIMITATIONS

FEio functions much like any text editor. It creates a temporary file to which modifications are applied and builds an internal index that points to the file locations of the forms each time it is invoked. This initialization time grows linearly with the size of the FE data file. However, once past the initialization step, the response time for most commands (including interform searching) is usually within three to six seconds. Editing data files that are close to the host system's file size limitation (usually 1 to 2 megabytes) can cause file overflow problems. When this happens, FE attempts to terminate processing as gracefully as possible.

V. THE FE SYSTEM FORM-DEFINITION LANGUAGE

Each form used in an application requires the creation of a PFDF that contains a description of the form's template and the attributes of its associated fields written in FE's high-level form definition language. These form-definition files can be concatenated or "pasted" together at run time to form "logical forms." This logical form capability provides the application developer with the ability to treat PFDFs as modular constructs that can be used as building blocks for form definition. For example, if two distinct application forms (Fig. 3) contained a common section, PFDF "A" could define the common section and two other PFDFs ("B" and "C") could define the unique sections. Two LFDs specifying how the FE system is to "bind" the form components together would then be defined.

5.1 Dynamic form construction

The internal representations of forms defined for an application are not static in the sense that they must be compiled along with the FE source code. Instead, each form is translated and loaded into memory at run time where it remains until a request to load a different form is received.

This loading strategy provides the following benefits:

1. Logical forms sharing a common PFDF will always be constructed correctly.
2. PFDFs may be modified during run time to provide "dynamic" forms.

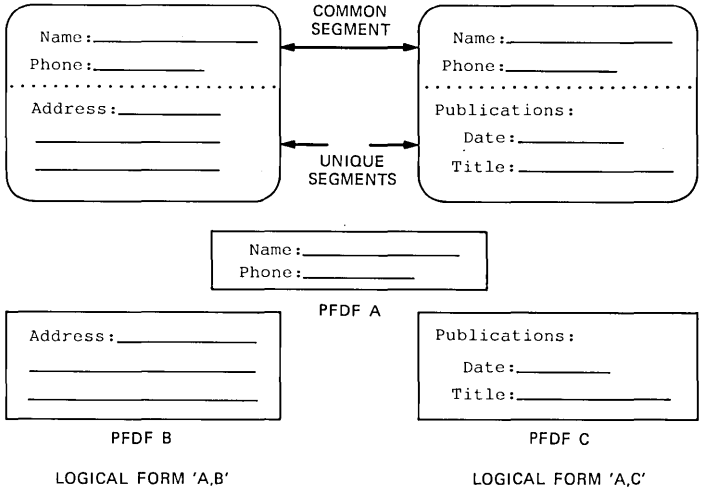

Fig. 3—Logical forms example.

3. Memory requirements are greatly reduced for multi-user, multi-form applications.

The major drawback associated with this strategy is that the time required to translate and load a form into memory is directly proportional to the size of the form. This translates to about two or three seconds for a form 24 lines or fewer and up to 20 seconds or more for forms larger than 100 lines. However, the forms in most applications seem to be in the 20- to 30-line range and two or three seconds seems to be acceptable. For applications that frequently switch between many large forms, the overhead may be unacceptable.

5.2 Physical form definition files

PFDFs are ASCII data files that can be created with any text editor, e.g., `ed`, `vi`, etc., and may define an entire form, or a form segment common to many forms. Because PFDFs are simple text files and are translated and loaded at run time, they can also be created during run time by application software. For example, an `FE`/Database Management System prototype exists that dynamically creates PFDFs from the database definition (transparent to the end user), and allows the user to input, modify, and query the database using the executable component of the `FE` system.

Each PFDF consists of three sections that describe the associated form's template, field attributes, and field help text. Only the template section is mandatory; an application could omit all other PFDF sections and default attributes would be assigned for all fields. Figure 4 shows the organization of these three sections on a typical PFDF.

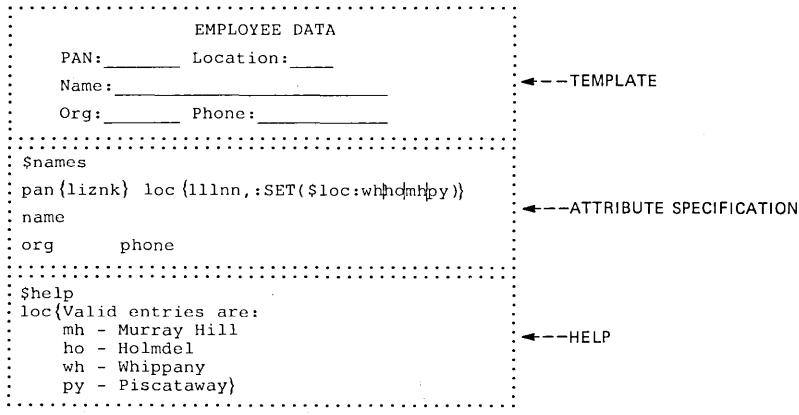

Fig. 4—PFDF format and organization.

The "Template" section describes the form image that is displayed on the terminal. Field locations (indicated by a series of contiguous underscore characters), lengths (the number of contiguous "_"s), and simple template text are elements of this section.

The "Attribute Specification" section of a PFDF is used to assign attributes to the fields that appear in the template section. Each field can be defined in terms of a name (*Fname*), data attributes (*Att_tuple*), a value assignment expression (*Asg_expr*) and a validation rule (*Valid_rule*) according to the following syntax:

Fname{*Att_tuple*,=*Asg_expr*,:*Valid_rule*}

Fname consists of a string of characters taken from the character set {A-z,0-9_-,.}. *Att_tuple* is a coded five tuple that defines the following field attributes:

1. Field enhancement attribute (underline, inverse video, red, blue, etc.).
2. Data type (integer, numeric, upper case, dollar, etc.).
3. Justification (right, left, centered, zero-fill).
4. Constant/Default/Locked/Shared/Protected value.
5. Key field (for interform searching).

For example, the first field of the form shown in Fig. 3 has *Fname* declared as "pan," its associated attribute tuple as "liznk." This attribute tuple decodes as follows:

'l'-Underlined on CRT,
'i'-Integer field,
'z'-Zero filled,
'n'-No protection,
'k'-An interform search key field.

The third field on the same PFDF declares the *Fname* attribute as "name" and allows the FE system to assume system defaults for all others.

Field values can be defined with an algebraic expression in *Asg_expr* that can include references to numeric constants, other fields, external programs, and shell-exported variables instead of end user input. For example,

```
total{1nrcn,=($field1 + ENV(value))/2}
date{1clcn,=EXT(date)}
```

The first example defines the value of field `total` as one half the value of the sum of field `field1` and the value of the shell environment variable `value` (as specified by the built-in function `ENV()`). In the second example, the value of field `date` is to be filled with the results of the external program `date` (as specified by the built-in function `EXT()`). The only requirement of programs executed by `EXT()` is that they return their result(s) to the standard output device. `EXT()` gives the application developer a direct "hook" to the *UNIX* system environment to obtain computation or validation capabilities not directly available in the form definition language.

A data-validation rule *Valid_rule* can be provided in each field-attribute specification. The syntax of the validation rules is algebraic, with arithmetic and logical operators taken from the C programming language. Validation rules specify under what conditions the field is logically correct or valid. Rules may contain references to other fields, external functions, constants, and environment variables. In addition, validation rules may also contain arithmetic operators. Below are two examples of validation specifications for character and numeric type fields:

```
(1) loc{1llnn,:SET($loc:mh|ho|wb|wh|py|cb)}
(2) sum{1nlnn,:$sum >= $val1 && RANGE($sum:0-$val2)}
```

Example 1 specifies that the value of the field `''loc''` must be a member of the set `mh`, `ho`, `wb`, `py`, or `cb` to be considered correct. `SET()` is the built-in function used for specifying a set of strings. In (2), the rule specifies that field `sum` must be greater than or equal to the value of field `val1` and also greater than or equal to zero and less than or equal to the value of field `val2`. Since `SET` and `RANGE` return Boolean results, the unary operator `!` (not) can be used to indicate "not a member of a set" (`!SET()`) or "outside the range" (`!RANGE()`).

A robust set of built-in functions exists for both value assignment and validation-rule specification that includes vector operations and regular-expression pattern matching.

The "help" section of a PFDF is used to define field-specific "help"

text for any or all fields defined in the Attribute Specification (`$names`) Section. The user may request the display of a field's "help" text by positioning the cursor on the selected field and entering the appropriate command.

In addition to explicitly defining the "help" text for a field in this section of the PFDF, the form designer can specify that the text source be retrieved from a data file or generated by an application program.

5.3 Logical form definitions

PFDFs may be combined, or concatenated by LFDs. LFDs can specify a simple or hierarchical concatenation of PFDFs. For example,

```
(1) payroll
(2) payroll,work_hist
(3) {dept{3*group{10*members}}}
```

Logical form (1) is defined as a single copy of PFDF `payroll`, while (2) defines a form that is constructed using a `payroll` PFDF concatenated with a `work_hist` PFDF. Example (3) is a hierarchical definition that translates as follows: concatenate one PFDF `dept` form with three PFDF `group` forms, each of which is concatenated with ten PFDF `members` forms.

VI. END-USER INTERFACE LANGUAGE

The end-user language is partitioned into "off-form" and "on-form" commands. "Off-form" commands are functionally and syntactically similar to the language of the text editor `ed` and operate on forms as a unit; for example, they delete or append a form. Other data file manipulation commands include multi-key sorting, searching, data extraction, and hard copy reproduction of forms, in addition to standard editing commands.

The on-form commands* are modeled on the visual editor `vi`, and apply only after an off-form command is entered to display either a blank form for input, or a filled-in form from the data file. These commands stay in effect until the user indicates the end of input or changes for that form. When the cursor is positioned on a form, the end user can be in either a "positioning" or a "data-change" mode.

In the "positioning" mode, single keystroke directional commands can move the cursor forward, back, up, or down. The cursor may also be positioned to a field via regular expression pattern matching and direct or relative line addressing. For example, move to form line *N*, or move up(−) or down(+) *N* form lines. When in the data-change mode, any characters entered by the user are placed on the field where

* Executable `FE` uses the data-collection library routine `FE_edit`.

the cursor is positioned. The user may enter or exit the positioning or data-change mode at any time during the editing session by typing the appropriate command.

Field editing capabilities include "insert character," "delete character," "erase field," "replace characters," and others; each of these can easily be "turned off" or redefined if considered inappropriate for the application.

VII. DATA FILE ORGANIZATION AND FORMAT

The organization of an `FE` data file can be seen in Fig. 5.

"Header" records delimit logical groups of associated "name/value" pair records. Each header record contains the following four components (separated by tab characters):

1. The logical name of the form,
2. The values of all fields defined with the "interform search" attribute,
3. The logical form's PFDF description, and
4. The number of errors contained in the associated data.

Name/value pair records are organized to match the layout of each logical form definition. For example, Fig. 5 represents the output file organization for the PFDF shown in Fig. 4. Changes made to a PFDF (adding, deleting, or changing the field order) do not require an `FE` data file to be changed. This feature is important for applications with volatile data input and validations requirements, since the field data will always "map" to the correct position on the constructed form.

Because `FE` data files are stored in ASCII format, simple `sed`, `awk`, or C programs can be written to generate reports or massage the data to a format acceptable to other application components. For applications that are already name/value pair oriented, the "data file"/"application system component" interface may be minimal.

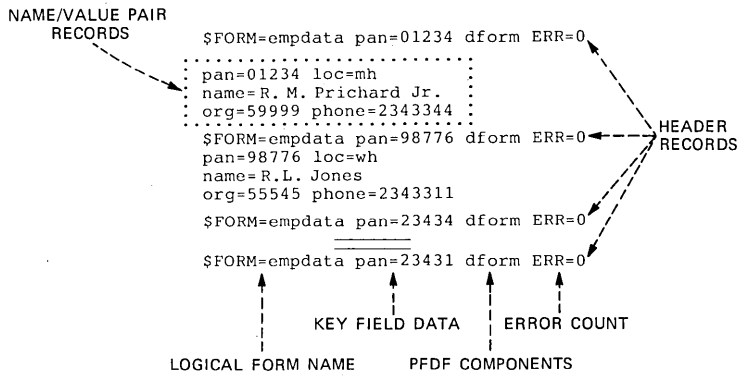

Fig. 5—`FE` data file organization and format.

VIII. EXT() BUILT-IN EXAMPLE

The power and versatility of `EXT()` can be seen in the following example:

A query form (Fig. 6) is displayed with only the entry of a single field "name" allowed. All other field values are supplied by a database retrieval program that uses the value of the "name" field as a search key for extracting the remaining field values from the associated databases.

This application can be implemented by defining a facsimile of the retrieval form in a PFDF with appropriate field names. One field's assignment attribute invokes the retrieval program as follows:

```
snn{Att_tuple,=EXT(ret_prog $name)}
```

To execute the retrieval program to fill in the rest of the form, the user would enter the appropriate data in the name field and then enter the **FE** system command "compute". `Ret_prog` would then extract the remaining form data from the database and write each field's name and associated value to the standard output device. For example,

```
pan=098999
ssn=212-22-2323
org=59999
  .
  .
  .
```

The **FE** system would then read the transmitted data and update, on the terminal, the field values as they are received.

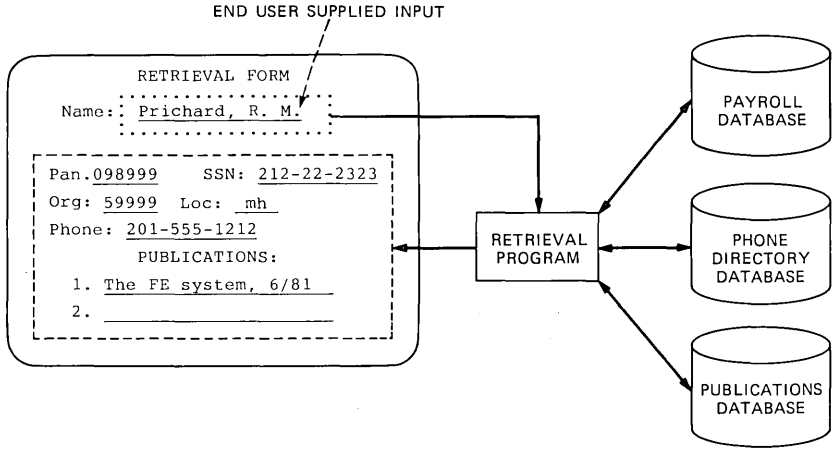

Fig. 6—Using the `EXT()` build-in.

IX. FE FORM LIBRARY

The form library currently consists of 80 subroutines. Of these subroutines, 25 are high-level routines intended for application interfaces and provide the following functions:

1. Virtual terminal and library initialization.
2. Physical form translation and loading.
3. Program placement of field values on forms.
4. Program retrieval of field values on forms.
5. End user input, modification, and validation of field values.
6. Termination or "wrap up" processing.

At the core of the FE library (Fig. 7) reside the virtual terminal interface and low-level form manipulation routines. The virtual terminal interface uses either the Termcap or Terminfo database and utility routines (based on the host operating system) to provide support for CRT as well as non-CRT devices.

The terminal interface and form manipulation routines are in turn used to form the application interface routines. The lower-level routines are intended for use by applications that require access to FE's internal structures or terminal control sequences.

The following example shows how the high-level library routines can be used to collect data via a form and "dump" the field names and data values to standard output in a name/value pair format.

> NOTE: Regular type represents the library routines while *italicized* type represents global library structures and variables.

```
#include <stdio.h>
#include "FE_structs.h"

#define REDRAW 1
#define FULLSCREEN 1

main()
{
```

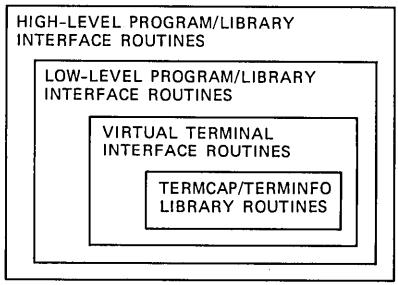

Fig. 7—FE library architecture.

```
(1)    extern int _dlinecnt, _fldcnt[];

(2)    extern struct symbol **_cp[];

       char buff[512];
       int i,j;
       struct symbol *cp;

(3)    FE_init(open("/dev/tty",2),FULLSCREEN);
       strcpy(buff,"dataform");

(4)    FE_bldform(buff,"/PFDF/form/directory");
(5)    FE_edit('c',REDRAW);

(6)    for(i=0;i<_dlinecnt;i++)
         for(j=0;j<_fldcnt[i];j++)
         {
           cp=_cp[i][j];
           printf("%s=%s\n",sp->name,sp->sval);
         }
     }
```

In this sample program, (1) and (2) declare two global FE library integer variables *_dlinecnt* (data line count) and *_fldcnt[]* (data line field count). These two variables define the dimensionality of the cursor positioning matrix *_cp[][]*. Each element of *_cp* is a pointer to the symbol table element associated with a single field on the current form in memory.

The first FE library routine called FE_init initializes the terminal interface routines (3). Two arguments are passed to it: a file descriptor for ''/dev/tty'' (opened for read/write) and FULLSCREEN to indicate that the full screen is to be used (not adjusted by the terminal's output baud rate). To build a form in memory the FE_bldform is called (4) with the logical form definition (stored in "buff," argument 1) and the directory where the logical form's PFDF components reside (argument 2).

To collect the data, FE_edit (5) is called with arguments 'c' and REDRAW. These arguments will cause FE_edit() first to redraw the template on the CRT and then to enter the "change mode," which allows the user to enter data immediately. This library routine provides the end user with a vi editor-like interface for entry and modification of the data. When the data entry session for the form is completed (the end user enters the appropriate "quit" command), control is returned to the program.

At (6), each element of the cursor positioning matrix *_cp* is "visited" and the associated field names and values are written to the standard output device via a printf.

Though this example uses the global internal structures of the library, the code in (6) can be replaced by the library's "get field value by name" routine calls. For example, if the *Fname* attribute of a field on the form built by the call to `FE_bldform` is `department`, the associated value could be retrieved by calling the `FE_get` routine as follows:

```
FE_get(''department'',value,changed,'r');
```

`Value` defines the string address where the retrieved value of the field `department` is stored, and `changed` is a string address where `changed[0]` indicates if the value of the field has (value = 1) or has not (value = 0) been changed. The fourth argument instructs `FE_get` to reset the associated field's internal "changed" flag to zero.

X. OBSERVATIONS

User community feedback has been favorable since the initial **FE** system prototype was released. A brief summary follows:

1. The end-user language is easily learned by users already familiar with the *UNIX* operating system because of the functional and syntactical similarity with the standard text editors **ed** and **vi**. For individuals not familiar with text editors, the time required to learn the **FE** end-user language is no more than that required to learn any other editor.
2. System documentation is generally acceptable.
3. Defining a form does not require much time because the form-definition language is simple.
4. Interfacing with most application components is not difficult.
5. Significant savings in design and development time have been achieved for all applications.

The last observation (5) can be best illustrated with an example.

An application system required about 12 to 15 weeks to develop a prompting program that only provided an initial data input facility. A total of eight different forms were supported and the software provided basic intra-form validation and a crash recovery capability. Data modification was implemented with the standard system text editor. When a new release of the system was built, **FE** was used to provide data collection and modification capabilities for more than 35 forms. Data validation was provided by application supplied software as **FE** validation was not yet available. The entire process to integrate **FE** into the application required less than two weeks to define the forms and build a simple software filter to interface the data file with other application components.

Because the form definitions were volatile in the initial development phases of the project, an even greater savings of development time

was realized since there were no programs to modify and recompile each time there was a form-definition change; all that was required was for the PFDFs to be changed using a text editor.

I believe the success of the FE system can be mainly attributed to the form-definition language (PFDs and LFDs) and the ability to support data entry as well as report and processing-control data forms. In addition, software developers have found it easy to "custom tailor" both the libraries and executable FE to meet the unique requirements of their applications.

In order for "screen" management software to exist in the future, generalized interfaces should be developed to support alternate input devices such as "mice," touch sensitive screens, etc. Though nothing will replace the standard keyboard for input of textual data, these alternate input devices will find their calling in processing control and cursor positioning.

XI. ACKNOWLEDGMENTS

I am indebted to D. G. Korn, K.-P. Vo, and other colleagues for their technical inputs and the FE system user community for providing constructive feedback.

REFERENCES

1. K.-P. Vo, "IFS—A Tool to Build Integrated, Interactive Application Software," AT&T Tech. J., this issue.
2. J. W. Brown, "Controlling the Complexity of Menu Networks," Commun. ACM, *25,* No. 7 (July 1982), pp. 412–18.
3. D. V. Morland, "Human Factors Guidelines for Terminal Interface Design," Commun. ACM, *26,* No. 7 (July 1983), pp. 484–94.
4. D. Tsichritzis, "Form Management," Commun. ACM, *25,* No. 7 (July 1982), pp. 45–78.
5. P. De Jong and R. Byrd, "Intelligent Forms Creation in the System for Business Automation(SBA)," IBM Research Report RC 8529, 1980.
6. D. W. Embley, "A Form Based Non-procedural Programming System," Technical Report, Department of Computer Science, University of Nebraska, 1980.
7. B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Englewood Cliffs, N. J.: Prentice Hall, 1978.
8. W. N. Joy and M. R. Horton, "UNIX User's Manual 4.1 BSD," TERMCAP(5) manual page.
9. M. R. Horton, "UNIX System V Release 2.0 Programmer Reference Manual," TERMINFO(4) manual page.

AUTHOR

Reuben M. Prichard, Jr., B.S. (Computer Science), 1976, and M.S. (Computer Science), 1978, Rutgers—The State University; AT&T Bell Laboratories, 1967—. Mr. Prichard has been a member of the Business Analysis Systems Center since 1976, where his work involves the design and development of decision support systems and software tools to support that work. His research interests include software design methodologies and human factors.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Data Extraction Tools

By D. G. BELANGER and C. M. R. KINTALA*

(Manuscript received April 9, 1984)

Data analysis includes the acquisition of data from a wide variety of sources, using different media and ranging in size from a few bytes to hundreds of millions of bytes. In the case of large data sets, the problem reduces to finding an efficient way for an analyst or other user to extract useful subsets from the source data with minimal programming knowledge and effort. Since different sources of data often have entirely different protocols, interfaces and procedures for access, the problem is also to reduce the complexity of data access by hiding this variety from the user. We describe a table-driven program generation system that provides a uniform interface to the analysts for requesting portions of data from any source. The system then generates a program that executes on the source system and extracts the requested data. The table-driven nature of the generator can be used to modify the style of the programs being generated. The tables are, in fact, target program abstractions specified in a high-level language. We now have tables that encode efficient C programs to extract data from IBM standard label tapes on *UNIX*™ systems and COBOL/DL-I/JCL programs to extract data from Information Management System databases.

I. OVERVIEW OF THE PROBLEM AND SOLUTION

The acquisition of data is a crucial part of the data analysis process. This data may be small enough in volume that it can be entered by hand by a single user (e.g., parameters to a program or a small data file); it may be data (e.g., survey data) entered manually by a variety of people; or it may be large volumes of data contained in the opera-

* Authors are employees of AT&T Bell Laboratories.

tional databases of the company (e.g., data on toll calls and charges from a specified area). A set of tools is described in this paper that addresses the problem of extracting, in a useful manner, data from high-volume sources (within or outside of the company). Specifically, there are tools for extraction of data from Information Management System (IMS) databases [Information Management System to *UNIX* System (IMX)] and from tapes generated on IBM computers (TTU). These tools use a single syntax and are resident on a *UNIX* system. The tools are built with a flexible code generation technique which allows for easy addition of new data sources. The key ideas in this process are the following:

1. The analyst typically does not have control over the source environment of the data. In particular, the computer system, database management system and mode of access (or distribution) of the data are likely specified by the administrator of the data.

2. The user should see a uniform and high-level language for data extraction. The details of various database systems and operating systems should be hidden from the user, fostering the feeling of working entirely within a single analytical environment.

3. The forms in which data are available are varied and changing. Any system of this sort must allow for new database management systems or other sources to be added easily.

4. The use of the system should be simple enough that the user is encouraged to extract only the amount of data that appears to be needed, knowing that more or different data can be easily obtained.

5. The volume of data available, and necessary, may be very large ($>10^8$ bytes per data set). Thus efficiency of extraction is critical.

Our approach to this problem was to design (1) a program generation system with a very high-level interface so that an analyst can easily phrase the required data request (i.e., no programmer help needed), and (2) a table-driven code generation system so that the system could be set up to generate code for several different modes of data extraction. In fact, we now have two tables for data extraction. One is for arbitrary IMS databases, for which the system writes programs using JCL, COBOL and DL/I languages; and one for extraction from Extended Binary Coded Decimal Interchange Code (EBCDIC) coded tapes (and translation to ASCII or binary code), for which C programs are written. In each of these cases the program generated is specifically written for the request made. The program is very fast, especially in the tape case (in fact considerably faster than one would expect from a typical programmer asked to do the same job).

Because of its usefulness to analysts, the program generation system and the two tables (for IMS and tape extraction programs) are often

used in conjunction with other analytical tools, some of which are described in this issue.

II. CURRENT STATUS

Versions of the program generation system for translation from IMS databases and from EBCDIC tapes are running on the AT&T *UNIX* Operating System V. The current versions are available to users within AT&T. These generators are currently being used routinely on data sets on the order of 150 megabytes. Much larger data sets are also being processed using these tools.

III. HISTORY AND MOTIVATION

3.1 The problem

During the process of data analysis, a wide variety of source data must be obtained and analyzed. This data may come from many different sources on different media (e.g., paper, magnetic tapes) and may range in size from a few bytes to hundreds of millions of bytes. The data itself may originate and be stored in a wide variety of computers using an even wider variety of database management (or file) systems. Often, these systems are part of a corporation's operations systems (e.g., accounting, sales operations) and are not accessible on a time-shared basis. When they are accessible, the mode of access is typically dictated by the database staff rather than by the analyst. The problem of data acquisition in these cases reduces to that of efficiently accessing a very large data set of a known type (e.g., an IMS database or a foreign tape) and extracting a defined subset of that data. In order to access such data sets, the analyst requires that a single, preferably high-level, language be available for making these requests.

The problem we are addressing then is how an analyst can routinely obtain data that is already in machine-readable form without learning new languages for the data retrieval and operating systems of foreign computers, and with the flexibility and efficiency needed to explore the data.

3.2 Existing tools

There are, of course, many tools which address parts of the data extraction problem. In this section we discuss the relationships between the IMS-to-*UNIX*/Tape-to-*UNIX* (IMX/TTU) system and other tools available to users of the *UNIX* operating system to manage data acquisition of large amounts of data.

If we consider the tools available on the *UNIX* operating system for the translation of foreign tapes, we see that a combination of `dd` and

awk, or in some cases dd and grep, can be used.[1] This approach will handle simple, character-oriented (i.e., not packed or binary) data. In order to do field selection, the user must write a script in awk. To do record selection on patterns, grep may be used. This method is not fast enough for very large data sets, even with a much faster than standard version of dd in use in our organization. In general TTU provides more functionality and is faster.* In addition, it takes advantage of efficiencies that can be gained by reference to the user's actual needs (e.g., translate only the requested fields) and by allowing users to request calls to their own processing routines. Finally, this method anticipates future expansion of needs and hiding their solutions within the current extraction language.

In the case of interfaces to database management systems (in this case IMS), the problem is not the absence of some good extraction systems (e.g., RAMIS II†). Instead, it is the lack of control over the environment in which the target data resides and the lack of uniformity in the extraction system's languages. Our approach has been to generate extraction programs in COBOL, initially, to provide a reasonably global coverage of installed IMS databases with IMX. This allows access to databases without access to a specific extractor. On the other hand, we have left open the option of adding program generation tables to use higher-level interfaces to IMS (or other database management systems). For example, provision of tables for RAMIS II and for ADABAS‡ is being considered. The point, then, is not that we need to raise the level of these languages but that the analyst needs a single interface (i.e., does not want to spend time relearning several different interfaces) and that our approach will provide that single interface for systems with low-level-interface languages (e.g., tapes translated in the C language) as well as those with higher-level languages.

IV. DESIGN APPROACH

In our system of data extraction tools, we take a table-driven program generation approach to retrieve data from large source data-

---

* For example, choosing 68 percent of 2 million 64-byte records using dd | egrep ran at about 136 seconds per megabyte. In addition, downstream processing of the data must read all 64 bytes per selected record. Using TTU, extraction times were about 8 seconds per megabyte when reading the entire record. In this example, only 24 of the 64 characters in each record were required. This reduced TTU time further to 4.5 seconds per megabyte. Of course, there was no change in dd | egrep. In this example, total system and user time decreased from 6.1 hours for dd | egrep | C program to 0.1 hour for TTU with processing option. Even when using the entire record, the difference is 8.4 hours to 0.2 hour. In general, improvement factors of 10 to 100 have been observed, depending on the data requested.[2]

† Trademark of Mathematica Products Group Inc.

‡ Trademark of Software AG of North America.

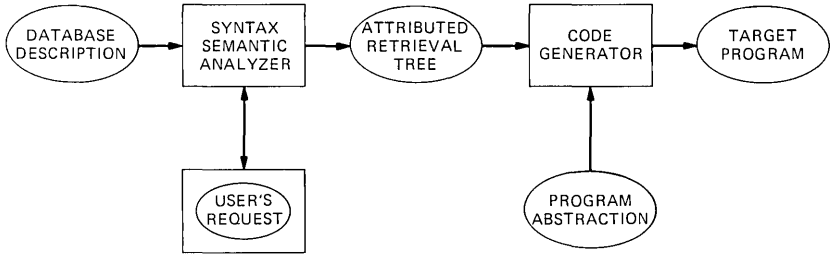

Fig. 1—System architecture.

bases. The architecture of this approach is depicted in Fig. 1. A dictionary file describing the source database is first created. This file provides the information for the system's display and part of the code-generator's decision process. The analyst requests the portion of the data to be extracted from this source in a high-level extraction language. The program generator analyzes this request for syntactic and semantic correctness and produces an "attributed retrieval tree" of this request. It then traverses this tree to translate the user's request into an equivalent program for the target database system using a table containing an abstraction of the extraction programs. This abstraction of the data extraction programs for the target system is written only once for each target database system in the language KS, a program abstraction language for the program generator.

This method of translating users' requests for data in a high-level language into complete programs for execution on the target database systems is similar to, albeit simpler than, the process of compiling. The attributed retrieval tree produced by the syntax and the semantic analyzer is similar to the parse tree produced by the front end of a compiler. Interpreting and making structural expansions of the constructs in the target program abstraction with the retrieval tree as the basis is analogous to a combination of the interpretative and the table-driven methods used in retargetable code generators for compilers.[3] A more formal treatment of the analogy between query language translators and compilers can be found in Ref. 4. By applying compiler technology to the database problems we hope to provide new insights into these problems and obtain new tools.

Aside from being similar to compilation, this table-driven program generation approach has several advantages in practice. Data extraction programs tend to be routine but tedious in character. Often, an application programmer working on a database system acquires an "existing" or a "sample" extraction program and changes it with the appropriate information about the new extraction request at the "appropriate" places in the original program to produce a new program

for the new request. This "knowledge," about how to write the new program, is encoded into the target program abstraction written in the KS language. The program generator can then "automatically" produce the target program given sufficient information about the request and the source database. Thus, this approach (1) provides high-level interfaces to the analysts for extracting data from databases, and (2) automates the process of programming to extract data.

By making the program generation table-driven, source database descriptions and target program structures can be changed easily. For example, in TTU, the tool that generates programs to read tapes, we have been able to add more functionality and speed to the system by simply incorporating some changes to the KS table without any recompilation of the tool itself. This approach separates the basic code generation algorithm from the details specific to the underlying database structure or the target program structure. This also allows us to easily change the tape record structures or the programming language, say from C to assembly.

V. SYSTEM DESCRIPTION

As explained in the previous section, we have a table-driven program generation system that can be easily targeted to generate programs, in a variety of programming languages, to extract data from a variety of source databases, such as tapes or IMS database systems. TTU, the tool we assembled from this system for generating C language programs to extract and translate data stored in EBCDIC format on tapes, will be described in this section to illustrate our system.

5.1 User interface

Any user wishing to extract data from tapes using TTU goes through the steps illustrated in Fig. 2. For any tape type,* the user or a data administrator must create a dictionary (i.e., a file describing the tape). A syntax-directed editor based on the "age" system[5] is provided for this purpose. It guarantees the syntactic correctness of the dictionary. Alternatively, any text editor may be used to create the dictionary by following the defined structure. Once the dictionary is created, users can specify the record conditions and fields to be selected. TTU uses this information to generate a program. The generated program performs the selection, extraction and translation of the data from tapes. In the case that special processing is required on the extracted data, the user can pipe the output of the generated program to a user process

* The tape type refers to the logical, not the physical, tape so that, for example, a tape issued monthly containing toll call records in the same format would be of the same type each month and require only one dictionary.

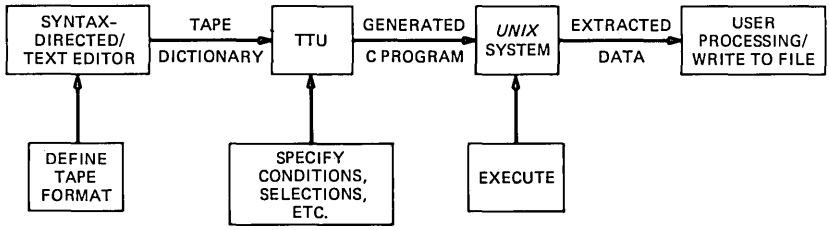

Fig. 2—User interaction with TTU.

or specify the name of the processing subprogram to TTU at the time of making the selections. The generated program will call the processing program at the appropriate places, eliminating expensive print and read operations. Finally, the generated program is compiled and executed in the usual manner.

5.1.1 Tape dictionary format

A tape dictionary is a file containing the information about the structure of the records in that tape. The first line in the dictionary contains the file name, followed by physical block length and Variable Length Record (VLR) tape indicator (`y` for VLR tape, `n` for fixed length record tape). This is followed by a block of lines for each record type in the tape. The first line in each block contains the record name followed by the record identifier string, the identifier indicator (`y` if the record has an identifier and `n` if it does not have an identifier) and the level number (`1` if there is only one record type or unrelated multiple record types on the tape; if the record types on the tape form a hierarchy, then it is the level number of the record type in that hierarchy). This first line in each block is followed by a sequence of lines, one for each field in the record. Each line for a field contains the field name followed by the field type indicator (`s` for alphanumeric string, `n` for numeric, `b` for binary and `p` for packed integer), key indicator (`y` if it is a key and `n` if not) and the field length.

5.1.2 Request format

Once a tape dictionary is created, the user invokes TTU for specifying the extraction request. TTU requests the tape name, searches its dictionaries for the requested tape dictionary and displays the record and field formats for that tape. Each field name is prefixed by a type indicator of `s` if the field is a string, `n` if it is a numeric string, or `p` if the field is a packed decimal, etc. The type of the field influences the format of requests for that data.

For each record, TTU requests a condition (i.e., a record selection criterion for that record). If the user wants all records of that type, a

carriage return should be entered. Otherwise, the condition should be entered in the following DNF (Disjunctive Normal Form) format:

field_cond & *field_cond* &...& *field_cond* | *field_cond* &...& *field_cond* |...,

where "&" means "and" and "|" means "or".

The *field_cond* is a field name (from the list provided by TTU) followed by a relational symbol (=, <, >, <=, > =) and a value or another field name. For example, to retrieve all state records where state is "NJ" or the revenue is greater than 15,000 and revenue is greater than expenses

```
state = "NJ" | revenue > 15000 & revenue > expenses.
```

Following the record selection process, the system prompts for field selection (i.e., record projection). A program will be generated to translate and return only those fields within a tape record which are requested. In this step the fields to be returned are identified. The default assumption is that all fields used in record conditions will be returned. In the previous example, state, revenue, and expenses will be returned. The response to the selection prompt may be a list of additional fields, separated by spaces, to be included in the retrieval; ALL, meaning retrieve all fields of this record type, or ALLBUT followed by a list of fields (separated by spaces) which are not wanted. The request for data is now complete. All that remains is to let the system know where the generated program is to be stored. For the target program generation, the system will default to a KS file, an abstraction of efficient C programs to read tapes, supplied with the system. This default can be overridden in order to use other program abstractions. The final step is to compile the generated program and execute it to read a tape.

5.2 Individual components

As illustrated in Fig. 1, TTU has two basic components, a syntax/semantic analyzer producing an attributed retrieval tree based on the user request and a code generator using this tree to generate the final program from a target program abstraction. The syntax analyzer is a YACC (Yet Another Compiler Compiler) generated parser[6] for the simple DNF condition language illustrated in the previous section. The semantic analyzer builds the attributed retrieval tree. The root of the tree contains all the attributes of the tape's global characteristics, for example the block length. Below the root, the internal node structure of the attributed tree corresponds to the record structure on the tape. Thus, there is a node under the root for every level-1 record on the tape, there are level-2 nodes for the level-2 records on the tape under the corresponding level-1 nodes, and so on. These nodes are

called record nodes. Every record node contains the corresponding record attributes, such as the user condition, the key string distinguishing it from other records, etc. Additionally, every record node has a set of leaf nodes for the corresponding fields in the record. The attributes of the field nodes are the items such as the field type and field length.

The code generator of TTU takes an attributed retrieval tree as input and interprets a target program abstraction given in a separate file to produce the target program equivalent to the user's request. The program abstractions are written in a special language called KS. Initially, the code generator starts at the root of the attributed tree. It reads the program abstraction one character at a time. Each character is printed to the output file unless it is one of the *meta* characters in the abstraction language. The *meta* characters are directives to the code generator for special processing to be done. The simplest of the meta characters asks for the substitution of a particular attribute value of the current node in the retrieval tree. This is similar to substitution in macro languages. There are meta characters which direct the code generator to repeatedly interpret a fragment of the program abstraction for every node in the retrieval tree whose attribute values satisfy a specified condition. In such iterative interpretations, the traversal of the retrieval tree can be limited to the subtree rooted at the current node, the path from the root to the current node or just the immediate successors of the current node. Iterative interpretations based on the attribute values of the current node are also possible.

The processing flow and the loop structures in data extraction programs seem to be isomorphic to the structure of the data in the source databases. This observation has allowed us to design and interpret the KS language for an attributed retrieval tree in such a *hand-in-glove* manner. The current version of the KS language is found to be rich enough to express the abstractions of programs in a variety of languages ranging from assembly to COBOL to C for data extraction from tapes and from IMS databases.

VI. WHAT WE HAVE LEARNED

In the process of developing the IMX/TTU system and changing it in response to user needs and suggestions, several things have become clear. The first is that, with proper tuning, fast processing of data can be done on *UNIX* systems. It is not unreasonable, for example, to think in terms of bringing large data sets to *UNIX* systems and processing them from tapes.

The program generation approach to this problem allows us to use efficiencies (e.g., loop unwinding) which would not, typically, be used by a programmer asked to do the same job. This implies, particularly

in the case of TTU, that the process of generating a TTU program in C is easier than requesting one from a programmer. In addition, the generated program will probably run faster. The resulting programs have very predictable structure but are relatively long and intricate.

The flexibility provided by the table-driven program generation approach has proven very valuable in allowing us to make expert programming knowledge available to end users. As an example, in a recent exercise where TTU performed slower than expected for a class of tapes, we were able, in a couple of hours, to add several improved algorithms by allowing an expert to modify the code generation table. The mode of transferring expertise by sitting down with an expert, adding suggestions to the code generation table, trying them out and altering them in real time was very productive.

The format of data in database management systems is relatively predictable (or at least hidden from the programmer). On the other hand, the formats used on tapes can be quite idiosyncratic. Consequently, while we expect IMX to handle nearly all IMS database structures, TTU translates only a subset of tape types. The objective has been to handle tapes produced by common COBOL, Fortran and PL/I programming techniques. This includes string, numeric, packed decimal, and fixed point binary. Adding other codes, e.g., Binary Coded Decimal (BCD) and floating point binary, is under consideration.

The addition to TTU (not yet to IMX) of the option for users to add their own processing routines has proven useful in terms of both flexibility and efficiency. Although its use requires programming ability, which the simple use of the system does not, it provides a growth path to more sophisticated use of data extraction.

VII. ACKNOWLEDGMENTS

We would like to thank Bob Kayel for motivating us to work on the design idea of separating the large production environments from the medium-sized analytical systems and for the encouragement in the initial stages of this work. We recognize the contributions from Bill Shugard and Griff Smith in preparing the target program abstractions for TTU and appreciate the suggestions and feedback from various users of this system.

REFERENCES

1. *UNIX System V User Reference Manual*, Release 2.0, AT&T Bell Laboratories, Inc., December 1983.
2. G. G. Smith, private communication.
3. M. Ganapati, C. Fischer, and J. Hennesey, "Retargetable Code Generators for Compilers," ACM Comput. Surveys, *14*, No. 2 (December 1982), pp. 573–92.
4. C. M. R. Kintala, "Attributed Grammars for Query Language Translations," Proc. Second ACM Symp. Principles of Database Systems, March 21–23, 1983, pp. 137–48.

5. B. A. Bottos and C. M. R. Kintala, "Generation of Syntax Directed Editors with Text Oriented Features," B.S.T.J., *62*, No. 10 (December 1983), pp. 3205–24.
6. S. C. Johnson and M. E. Lesk, "Language Development Tools," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 2155–76.

AUTHORS

David G. Belanger, B.S. (Mathematics), 1966, Union College; M.S., 1968 and Ph.D., 1971, (Mathematics), Case-Western Reserve University; Assistant, 1971–1974, Associate, 1974–1979, Professor of Mathematics and Computer Science, University of South Alabama; Computer Specialist, U. S. Army Corps of Engineers, 1973–1979; V. P., Gulf Coast Data Systems, Mobile, Alabama, 1977–1979; AT&T Bell Laboratories, 1979—. Mr. Belanger is currently Head, Advanced Software Department. His research interests include database management, automatic program generation and distributed computer workstations. Member, ACM, IEEE Computer Society.

Chandra M. R. Kintala, B.Sc. (Electrical Engineering), 1970, Regional Engineering College, Rourkela, India; M.Tech. (Electrical Engineering), 1973, Indian Institute of Technology, Kanpur, India; Ph.D. (Computer Science), 1977, Pennsylvania State University; Assistant Professor of Computer Science, University of Southern California, 1977–1980; AT&T Bell Laboratories, 1980—. Mr. Kintala is presently Supervisor, Advanced Programming Environments Group. His current research interests include programming environments and compiler techniques for application languages. He is also an Adjunct Professor of Computer Science at Stevens Institute of Technology. Member, ACM, IEEE Computer Society, Sigma Xi, Phi Kappa Phi and Who's Who in the East.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Datastream—A Language for Large Files

By D. SWARTWOUT*

(Manuscript received April 9, 1984)

Datastream is a simple language designed for writing programs that perform computations on large data files in the *UNIX*™ operating system. Datastream handles larger files and a wider variety of computations than most *UNIX* database management systems, but it provides no explicit support for updates. Its features and performance characteristics are particularly good for working with infrequently updated, mostly statistical data. This paper describes the Datastream language, outlines its evolution, and summarizes users' experience with a prototype implementation.

I. INTRODUCTION AND CURRENT STATUS

Datastream is a simple language designed for writing programs that perform computations on large data files in the *UNIX* operating system. It features simple control structures, built-in input/output, and arithmetic expressions in the style of the C programming language.[1] Datastream users can concentrate on defining computations without spending much time on other aspects of programming. The system has been implemented on the VAX 11/780† computer system under AT&T *UNIX* System V and University of California, Berkeley *UNIX* 4.1 and 4.2 Berkeley System Distribution (BSD), and on the AT&T 3B20 Simplex computer. It is written in C and has roughly 11,000 lines of source code.

Datastream processes larger files than most *UNIX* database management systems that run on comparable hardware. The largest data

* AT&T Information Systems.
† Trademark of Digital Equipment Corporation.

file that Datastream has handled contained roughly 70M bytes of data. Several applications use files of 40 to 50M bytes. Sets of data files can be interconnected and used as a database; the largest database totals 140M bytes. Datastream is a tool for working with databases, but it is not a database *management* system: it provides no explicit support for updates. Datastream programs can build new data files from old ones, however. These characteristics make the system particularly suitable for analysts and others whose files are large and infrequently updated.

Section II of this paper gives a brief history of Datastream's evolution, and Section III summarizes the resulting architecture. Section IV describes the main features of the language, Section V discusses experience with it, and Section VI draws some conclusions.

II. HISTORY AND MOTIVES

Datastream has passed through three distinct stages since work began on it in late 1980. It was originally intended to manage distributed analytical databases. Several interesting, large-scale, single-site applications appeared before any distributed ones, so our attention shifted to large nondistributed databases. At that time "large-scale" meant roughly 2 to 20M bytes, but initial successes with databases in this range encouraged users to try still larger files. Changes made to accommodate these "very-large-scale" files removed the last similarities to ordinary database management systems; Datastream had evolved into a special-purpose programming language. The rest of this section discusses these changes in more detail.

2.1 Distributed database system

We set out originally to develop a system that would maintain distributed analytical databases efficiently. The *UNIX* operating system was expected to be used at each site, no database-oriented changes would be made to the operating system kernel or file system, and no special features were expected from the network connecting the sites. At the database designer's discretion, a given data file could be present at two or more sites, but no site was expected to have all the data. Queries were to be decomposable so that several sites could process parts of a query in parallel whenever possible. Distributing partial queries to multiple processors can be a complex process, and we did not wish to spend much time developing software to solve it. Datastream's query language was designed to permit easy decomposition of queries.

Concentrating on analytical databases had several important consequences. First, analytical databases change infrequently. New data may arrive quarterly or annually or even never, and updates are usually appended to the database; they almost never destroy existing data.

Second, analytical databases are large. It took several years of experience to fully understand this point, but it was clear from the beginning that we would need hardware of at least the VAX computer class. Finally, not all analysts should be considered "naive users." Many are experienced, if not expert, programmers. Some have written tens of thousands of lines of Fortran, and virtually all of them are comfortable with such things as operator precedence, assignment, and the difference between integer and floating-point arithmetic.

2.2 Statistical database system

An earlier paper discusses adjustments that were made to accommodate large single-site databases.[2] Those adjustments are summarized here. The general goal was to make Datastream a tool that would extract interesting subsets of databases for further analysis by a system such as S.[3,4] The original prototype of Datastream performed rudimentary computations such as conversion between English and metric units, but it could not do aggregate computations such as sums and averages. Datastream's computational facilities needed to be much stronger to make it useful as an analytical tool. We added three main features to strengthen the query language: conditional expressions, `collect` statements, and `compress` statements, all described in Section IV. We did not add specific aggregate functions such as sum, count, or average. Analysts can write a wide variety of aggregate computations with `collect` and `compress` statements, including, but by no means limited to, the standard aggregates.

2.3 Special-purpose programming language

An epitaph for phase two in Datastream's evolution might read, "Give them a megabyte, and they take a hundred." Initial success with databases up to about 20M bytes encouraged users to try still larger data sets. The largest in use at this writing has about 70M bytes. Datastream could not handle databases of this scale in phase two because building new databases required too much computing and file handling. "Raw" data had to be preprocessed to make it intelligible to the query-processing software. The results were stored in specially formatted database files. These "cooked" files could be processed efficiently, but they had several disadvantages. They were expensive to create. The preprocessing program sorted large amounts of data; its running time was proportional to $n \log n$, where n is the number of records in the database. It also created large temporary files. The configurations and load patterns of the machines running Datastream were such that these costs became excessive as the size of the data files approached 20M bytes. Cooked files could be used only by Datastream. Numeric fields were stored in binary (not ASCII) form,

character strings were terminated by null (that is, zero) bytes, and records had Datastream-specific headers with assorted binary fields. Ordinary commands such as `grep` and `sort` were useless for handling cooked files, and users who tried to display them risked putting their terminals into a hopelessly confused state.

We solved these problems by replacing the lowest-level file-handling routines in the query-processing software with similar routines that could handle ordinary files. By "ordinary" we mean that records correspond to lines, data fields are delimited by some fixed character such as the tab, and numeric fields are in a printable form. "Ordinary" does not mean "any." It is hard to get Datastream to produce useful results from a file that contains more than one kind of record, for example.

Datastream originally constructed relationships among database objects in the cooking process, so we had to find a suitable way to set up connections between uncooked files. This was the most difficult part of the conversion. We chose a descendant of the original relationship mechanism that sacrifices some generality but works with ordinary files. This approach to connections is described in Section IV.

As a result of the switch to ordinary files, Datastream ceased to be a database management system. We regard it as a special-purpose language tailored to writing programs that perform computations on large, analytical data files. Datastream also can be used to build new analytical files from old ones. For historical reasons, Datastream programs are usually called "queries."

III. ARCHITECTURE

3.1 Software

Datastream is implemented as three major modules: the query compiler, the execution supervisor, and the statement processor. Each is discussed in a subsection below. The command

`stream` *qname*

starts query processing. *Qname* is a file that contains the text of the query. Usually a query file is created by the user with a text editor, but queries have been written by C programs, `awk`, `m4`, and even other queries. `Stream` writes its results on the standard output, so they can be displayed on the user's terminal, saved in a file, or piped to another program for further processing. Several variations on this theme can be requested by additional command-line arguments.

3.1.1 Query compiler

The Datastream query compiler translates queries into an inter-

mediate code. This code is kept in a file and used again without recompilation the next time *qname* is run, unless the source text has changed. The intermediate code consists of sections that correspond closely with statements in the source text. They are called "compiled statements" from here on. Each compiled statement describes input and output formats, temporary variables, constants, and the processing required to transform input to output. Most of the latter consists of reverse-Polish expressions that define computations and assign results to temporary variables or fields in output records. The intermediate code also contains a small amount of information about control flow.

The code for the query compiler includes a scanner generated by `lex`,[5] a parser generated by `yacc`,[5] and assorted support routines. It has been rewritten extensively twice, with numerous minor revisions. It is a one-pass compiler with respect to statements (that is, the compilation of a statement is independent of statements that follow it), but it makes two passes over each statement. The first pass parses the statement and generates code for expressions. The second pass formats output records, identifies temporary variables, and writes the remainder of the intermediate code. Most of these characteristics resulted from evolutionary pressure, not explicit design. The query compiler is the largest and most complex of the modules. At this writing it requires about 100K bytes of main memory at run time. In some unusual cases the statement processor can grow larger than this via dynamic storage allocation.

3.1.2 Execution supervisor

When the execution supervisor starts working, it searches for an up-to-date compiled query file and runs the query compiler to make one, if necessary. Then the execution supervisor starts one copy of the statement processor for each compiled statement in the query. It breaks the compiled query into statements, passes each compiled statement to the corresponding process, creates pipes through which the statement processors communicate, and waits for them to terminate. It checks for abnormal terminations and writes an error message when something goes wrong.

Figure 1 illustrates this process. Boxes represent *UNIX* system processes and ovals represent data files. Arrows trace the movements of data and queries through the system. The source text of a query is represented in the upper left-hand corner. Each statement begins with one of the key words `get`, `collect`, or `compress`, which represent Datastream's three control structures. In simplified form, they work as follows. The `get` statement tells its statement processor to get selected lines from a data file and transmit them through a pipe to the next statement processor. The `compress` statement reads lines from

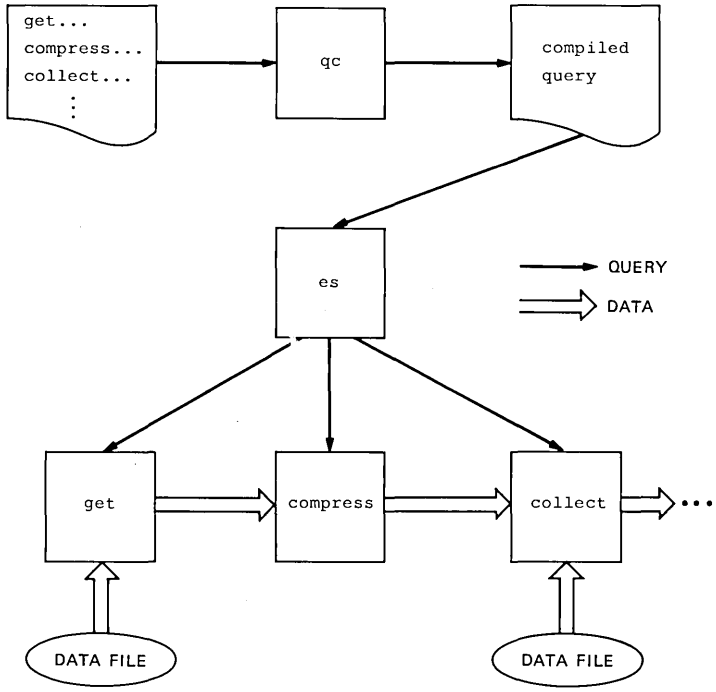

Fig. 1—Datastream architecture.

its inbound pipe and compresses them into aggregate values (for example, averages) that are written into its outbound pipe. `Compress` statements do not read from the database. Like the `get` statement, the `collect` statement gets lines from a data file, but instead of writing them immediately into its outbound pipe, it computes aggregate values from them, attaches these values to the inbound lines, and writes the augmented lines into its outbound pipe. Known collectively as the *mainstream*, these pipes are the data paths that connect the `get`, `compress`, and `collect` processes in Fig. 1.

Most *UNIX* systems impose a limit on the number of concurrent processes a user may have. When a query has too many statements to process at once, the execution supervisor divides the query into blocks of statements and then processes the query a block at a time. The Datastream user can control the number of statements per block, but in practice the default value of ten statements per block is rarely overridden.

Although it is not discussed further in this paper, it is possible to specify postprocessing of query output by including a *UNIX* system shell script[6] in the source text of a query. When this feature is used, the execution supervisor has to start a shell, give it the script, and

connect it with a pipe to the appropriate statement processor. The execution supervisor's duties are "just" routine management of detail, but its code totals six hundred lines of C.

3.1.3 Statement processor

A statement processor reads from two sources of input and writes two kinds of output. Its inputs are an inbound mainstream pipe and data file. The first statement of a query has no inbound mainstream to read, of course. No statement processor reads more than one data file, and `compress` statements read only the mainstream. Each statement in a query except the last one writes into an outbound mainstream pipe that becomes the inbound mainstream for the next statement processor. Some statement processors write on the standard output, specifically, the last statement in each query. Sometimes intermediate statements print with standard output directed into files. Anything that can be transmitted through the mainstream can be printed on standard output, and vice versa.

3.2 Files

A Datastream data file is a sequence of lines. Each line is subdivided into *fields*, which can be one of four *types*: integer, real, character string, or pointer. Fields of all types appear as variable-length ASCII strings in the data file, and they are separated by a single-character *delimiter*. The delimiter is usually a tab, but it can change from file to file. When a numeric field is used in a query statement, the ASCII text from the data file is converted to a suitable binary form. On the hardware that currently supports Datastream, integers and pointers are 32 bits long, and reals are 64 bits.

Each line in a data file represents an *entity*, and all the lines in a given data file represent entities of some entity type.[7] *Entity types* usually correspond to interesting persons, places, or things in the external world, for example, telephone calls or telephone customers. The user assigns a name to every entity type. In this paper, names for entity types will be ordinary nouns with the first letter capitalized to emphasize status as a component of the database: Call and Customer, for example. Datastream knows nothing about an entity except the values of the fields in the line that represents it. Lines in a given data file all have the same fields in the same order. If a line has too few fields, the missing fields are assumed to have appropriate null values; extra fields, if any, are ignored.

Data files have to be described in a configuration file before Datastream can use them. The *configuration file* contains a description of each entity type in the database, including the name of the entity type, the name of its data file, and a list of its field names and types.

The query compiler reads the configuration file to get information about names and types. The execution supervisor extracts data file names and passes them to statement processors that need to know which data file to open.

For example, a data file might be described as follows:

```
Call          :   /usr/don/phone/Callfile
    telno     :   string
    minutes   :   real
    miles     :   integer
```

Lines of `/usr/don/phone/Callfile` represent telephone calls, and each contains three fields: a character string named `telno`, a real named `minutes`, and an integer names `miles`, in that order. `Call` is the entity type name. One can think of a data file as a table with a fixed number of named, typed columns and an unspecified number of unnamed rows. Here are some Call data displayed in that form:

telno	*minutes*	*miles*
111-2233	1.65	55
111-2233	25.1	561
444-5566	1.88333	34
444-5566	19.45	455
444-5566	31.6333	736
777-8899	44.9167	1040

IV. FEATURES

A Datastream query consists of one or more *statements.* The simplest kind of query has just one statement that prints the value of some field from each line of some data file.

```
get each Call
    print .miles, .telno ;
```

results:

```
55          111-2233
561         111-2233
34          444-5566
455         444-5566
736         444-5566
1040        777-8899
```

The phrase `get each` *entity-type* is an iterator; it causes the rest of the statement to be executed once for each line in the entity type's

data file. Lines are processed in the order in which they appear in the data file; reordering the lines in `/usr/don/phone/Callfile` would cause a corresponding change in the output shown above. The key word `print` introduces a list of values to be printed. The values can be simple fields as above, or more complicated *expressions*. The dots in `.miles` and `.telno` mean that `miles` and `telno` are fields in lines from Call's data file. They are redundant information in this query, but as we will see later, such dots play an important role in more complex ones. This convention is meant to suggest `Call.miles`, which is a familiar notation to users of C and Pascal.[8] The query compiler ignores most white space (that is, blanks, tabs, and newlines). Every statement ends with a semicolon.

4.1 Expressions and assignments

Typical Datastream queries are full of expressions. The arithmetic operations available are addition, subtraction, multiplication, division, and modulus. All are infix binary operators, and they are represented by the symbols used in C: `+ - * / %`. The precedence rules from C are used, and arbitrary groupings can be specified with parentheses. Datastream and C part company on the issue of string concatenation. In Datastream the symbol ~ (tilde) is an infix binary operator that concatenates string arguments. C does not have string concatenation, and in C the tilde is a unary operator that computes the one's complement of its argument.

Conditional expressions are an important part of the language. A conditional expression has the form

`if` (*condition*) *expression1* `else` *expression2*

If the condition is true, then *expression1* is evaluated and its value is the value of the whole expression. If the condition is false, then *expression2* is evaluated and its value becomes the value of the whole expression. This is exactly the `?:` construction from C, set in a syntax that Datastream's users find more comfortable.

A condition can be any Boolean combination of comparisons of expressions. The Boolean operations are `or`, `and`, and `not`, in order of increasing precedence. Comparison operators are mostly taken from C: `== != > >= < <=`. In addition, a comparison can be a regular expression match:

string-expression `matches` *"regular-expression"*

A simple regular expression is used in the following example.

```
get each Call
    print if ( .telno matches "444-" )
        "city" else "suburb", .telno ;
```

results:

```
suburb      111-2233
suburb      111-2233
city        444-5566
city        444-5566
city        444-5566
suburb      777-8899
```

In general, strings can be matched against regular expressions of the same form as those recognized by the editor `ed`.

Early in Datastream's evolution, users who were not familiar with C complained that the conditional expression

```
if (a = b) 1 else 2
```

contained a syntax error (the symbol for equality comparison was supposed to be ==, not =). The two symbols are now treated as synonyms.

Properly installed functions written in C can also be used in expressions:

```
get each Call
    print .miles, log( .miles );
```

results:

```
55          4.00733
561         6.32972
34          3.52636
455         6.1203
736         6.60123
1040        6.94968
```

The `print` clause uses simple formats that are adequate for most purposes. Sometimes a query writer needs more precise control over the format of the output. The `printf` clause is available for this purpose. The query writer simply makes a `printf` call as it would appear in a C program:

```
get each Call
    printf( "phone number: %s time: %12.2e\n",
        .telno, .minutes );
```

results:

```
phone number:    111-2233    time:    1.65e+00
phone number:    111-2233    time:    2.51e+01
phone number:    444-5566    time:    1.88e+00
phone number:    444-5566    time:    1.94e+01
phone number:    444-5566    time:    3.16e+01
phone number:    777-8899    time:    4.49e+01
```

4.2 Control structure, part 1

The `get` statements we have described can be adjusted by inserting a `such that` clause:

```
get each Call
     such that .miles > 500
        print .telno, .miles ;
```

results:

```
111-2233      561
444-5566      736
777-8899      1040
```

The `such that` clause allows the rest of the statement to be executed only on lines of the data file for which the condition is true. There is no mechanism for optimizing data file access in the presence of a `such that` clause; every line of the data file is read and tested.

The `collect` statement is Datastream's second control structure.

```
initialize count = 0, totalmin = 0.;

collect each Call
     count = count + 1,
     totalmin = totalmin + .minutes
then
     print count, totalmin ;
```

results:

```
6    124.633
```

Like `get`, `collect` reads every line in a data file. The assignments between `collect` and `then` (known as *inner assignments*) are executed once for each line (in the order in which they appear in the query). When processing reaches the end of the data file, the `print` clause is

executed. The `initialize` statement is executed once before anything else is done.

The query above would be shorter if one could write something like

```
print count( Call ), sum( Call.minutes ) ;
```

The main problem with functions such as `count` and `sum` is that they are never enough; analysts are adept at inventing questions that require endless subtle variations on functions they have used before. Rather than try to stay ahead of users' creativity, we designed the `collect` statement, with which one can write computations including `sum`, `count`, `average`, `maximum`, and so on, all in a similar way. It also allows more exotic things, such as the following stratified sum:

```
initialize shortmin = 0., medmin = 0., longmin = 0.;

collect each Call
     shortmin = if ( .miles < 100 )
                shortmin + .minutes else shortmin,
     medmin = if ( .miles >= 100 and .miles < 500 )
                medmin + .minutes else medmin,
     longmin = if ( .miles >= 500 )
                longmin + .minutes else longmin
then
     print shortmin, medmin, longmin ;
```

results:

```
3.53333    19.45    101.65
```

The analysts who use Datastream are almost always comfortable writing this kind of computation. Initializations, counters, and partial sums are familiar to those with some programming experience (most of them), and conditional expressions are easy to learn. No one seems to miss programming the things that Datastream does automatically in a query like this: opening the file, reading and parsing the lines, recognizing the end of file, closing the file, and so on. Query writers are free to spend their time describing solutions to problems rather than programming input/output operations.

4.3 Mainstream variables

Some queries require more than a single statement. The mainstream provides a way to pass data values from one statement to the next for further processing. Wherever a `print` clause appears, a `keep` clause can be used instead.

`print` *expression1, expression2, . . .*

`keep` *expression1* `as` *name1, expression2* `as` *name2, . . .*

The `print` clause writes a stream of lines on the standard output, while the `keep` clause writes a stream of mainstream records into a pipe. A *mainstream record* is a set of values for a set of *variables*. The values can be any of the four Datastream types: integer, real, string, or pointer, but they are represented in an internal form that is not accessible to the user. To maintain this contrast between data in files and data in mainstream pipes, we will observe the following convention: data files are filled with lines, and lines consist of fields; mainstream pipes are filled with records, and records consist of variables.

Replacing `print` with `keep` in the long-haul query above produces

```
get each Call
    such that .miles > 500
          keep .telno as T, .miles as M ;
    .
    .
    .
```

This statement writes mainstream records as follows:

T	*M*
111-2233	561
444-5566	736
777-8899	1040

4.4 Control structure, part 2

At the receiving end of a mainstream pipe, one can write a `compress` statement.

```
get each Call
    such that .miles > 500
        keep .telno as T, .miles as M ;

initialize longest_call = 0, longest_telno = "";

compress
    longest_telno = if ( M > longest_call ) T else
    longest_telno,
    longest_call = if ( M > longest_call ) M else
    longest_call
then
    print longest_telno, longest_call ;
```

results:

```
777-8899      1040
```

One can think of the `compress` statement as a `collect` statement that operates on records from the mainstream rather than lines from a data file. The initialization is done once, after which the inner assignments are executed on each inbound mainstream record. The `print` clause executes when the inbound mainstream ends. The last example was contrived to make a simple illustration of the `compress` statement. It is possible (exercise for the reader) to rewrite it with a single `collect` statement. The following query uses a `compress on` clause to find the shortest-distance call for each Customer.

```
get each Call
    keep .telno as T, .miles as M ;
initialize shortest_call = 1000000;

compress on T
    shortest_call = if ( M < shortest_call ) M else
    shortest_call
then
    print T, shortest_call ;
```

results:

```
111-2233      55
444-5566      34
777-8899      1040
```

`Compress on` is the most elaborate of Datastream's control structures. The stream of inbound mainstream records is divided into *compression sequences*. Each compression sequence is a maximal sequence of successive inbound mainstream records, all with the same values for the variables in the `compress on` clause (in this example, just `T`). The mainstream records written by the first statement are shown below, separated into compression sequences by blank lines.

```
T             M
111-2233      55
111-2233      561

444-5566      34
444-5566      455
444-5566      736

777-8899      1040
```

The initialization executes once at the beginning of the compression sequence. Then the inner assignment executes for each member of the sequence. Finally, the `print` clause executes once at the end.

Ordering is crucial here, because the statement processor does *not* look ahead in the inbound mainstream. If the Call file were rearranged as follows

telno	*minutes*	*miles*
111-2233	1.65	55
111-2233	25.1	561
444-5566	1.88333	34
777-8899	44.9167	1040
444-5566	19.45	455
444-5566	31.6333	736

then the compression sequences would be

T	*M*
111-2233	55
111-2233	561
444-5566	34
777-8899	1040
444-5566	455
444-5566	736

and the query's output would be

111-2233	55
444-5566	34
777-8899	1040
444-5566	455

The `compress on` clause has been valuable in constructing new data files as well. This use of it will be discussed further in Section 4.6.

4.5 Connected files

The original Datastream included a program called `build` which constructed "cooked" databases from "raw" data. It was capable of building arbitrary binary relationships between entity types. When `build` was scrapped for performance reasons, some means for connecting ordinary *UNIX* system files had to be found. The approach we have taken is not as general as the original, but it provides high

performance and does not make specially formatted copies of large data files.

Datastream's current connection mechanism is based on pointers. A *pointer* is an offset from the beginning of a data file to the beginning of some data line. Pointers can be stored as fields in one file and used in queries to provide fast access to interesting lines in some other file. In particular, pointers are used to implement entity lists. An *entity list* associates one entity with an ordered list of entities of some other type. For example, assuming that the `telno` field in our Call file is the telephone number of the customer who dialed the call, consider the following data about Customers:

Configuration:

```
Customer              :   /usr/don/phone/Customerfile
    telno             :   string
    Call_count        :   integer
    total_minutes     :   real
    total_miles       :   integer
    Call_pointer      :   Call control by telno
```

Data:

telno	*Call_ count*	*total_ minutes*	*total_ miles*	*Call_ pointer*
111-2233	2	26.75	616	0
444-5566	3	52.9666	1225	35
777-8899	1	44.9167	1040	95

`Call_count` is the number of calls the Customer made, `total_minutes` and `total_miles` are self-explanatory, and `Call_pointer` is the number of bytes (including new lines) of `/usr/don/phone/Callfile` that have to be skipped to reach the Customer's first Call.

The following example shows how a pointer behaves in a query.

```
get each Customer
    such that .telno = "444-5566"
        keep .Call_pointer as it ;
get each Call for it
    print .telno, .minutes ;
```

results:

```
444-5566      1.88333
444-5566      19.45
444-5566      31.6333
```

The `keep` clause defines `it` to be a pointer-valued variable. The second statement behaves just as it would without the `for it`, except that the first `Call` retrieved from the data file is the one `it` points to, and `Calls` are processed so long as they have the same value for `telno`. Readers familiar with relational database systems will recognize `Call_pointer` as a way of storing the results of joining `Call` and `Customer` on `telno`. The configuration file declares that `telno` is the *control field* for `Call_pointer`; that is, a change in its value signals the end of the entity list. The above syntax for declaring control fields is new, unsettled, and likely to change.

The first statement in the previous example writes only one outbound mainstream record. If it wrote more than one, the second statement would repeat its action for each inbound mainstream record.

```
get each Customer
     such that .Call_count >= 2
          keep .Call_pointer as it ;

get each Call for it
     print .telno, .minutes ;
```

results:

```
111-2233      1.65
111-2233      25.1
444-5566      1.88333
444-5566      19.45
444-5566      31.6333
```

Datastream supports a similar `collect each . . . for` clause, as the following query shows.

```
get each Customer
     such that .Call_count >= 2
          keep .telno as telno, .Call_pointer as it ;

initialize longest = 0.;

collect each Call for it
     longest = if ( .minutes > longest ) .minutes else
     longest
then
     print telno, longest ;
```

results:

```
111-2233      25.1
444-5566      31.6333
```

The initialization is done once for each mainstream record. Then the inner assignment is done for each Call starting with the one `it` points to and continuing until `telno` changes. When the end of file or a Call with a different `telno` is encountered, the `print` clause executes and processing resumes with the initialization for the next inbound mainstream record.

Pointers can be printed, and they can be used in some expressions, notably conditional expressions. They cannot be used in arithmetic. Pointers can be compared for equality but not magnitude. The general rule is that algebraic manipulation of pointers should be discouraged because a pointer that does not point to the beginning of a data line can lead to seriously mangled results. Of course, pointer arithmetic and comparison are legal in many languages, and they may find their way into Datastream if a pressing need arises.

4.6 Constructing pointers

Users can make data files containing pointers any way they wish. It has been done with special-purpose C programs, but it is more common to use queries. As the statement processor traverses a data file, it keeps track of where it is. In any statement of the form

```
get each X...
```

or

```
collect each X...
```

the phrase `the` *X* is a pointer-valued expression whose value is the offset to the beginning of the current data line. This feature can be used as follows (note the use of `the Call` in the `keep` clause).

```
get each Call
     keep the Call as Call__inbound, .telno as telno ;

initialze Call__ptr = null(Call);
compress on telno
     Call__ptr = if ( Call__ptr = null(Call) )
          Call__inbound else Call__ptr

then
     print telno, Call__ptr ;
```

results:

```
111-2233      0
444-5566      35
777-8899      95
```

The mainstream records written by the first statement are shown below, separated into compression sequences with respect to `telno`.

Call_inbound	*telno*
0	111-2233
17	111-2233
35	444-5566
55	444-5566
74	444-5566
95	777-8899

An expression of the form `null(`*entity-type*`)` is always unequal to ordinary *entity-type* pointers in comparisons. This means the comparison in the conditional expression above is true only for the first member of each compression sequence, when `Call__ptr` has its initial value of `null(Call)`. The conditional expression evaluates to the ordinary Call pointer `Call__inbound`, which is assigned to `Call__ptr`. For all succeeding members of the compression sequence, `Call__ptr` is unequal to `null(Call)`, so its value does not change. Thus whenever `Call__ptr` is printed, its value is just a pointer to the first Call for some Customer.

The query above was written by a program called `canon`. `Canon` takes a configuration file and the name of a pointer field (`Call_pointer` in this example) as input. It writes a "canonical" pointer-construction query as output. Such a query always has two statements, a `get` and a `compress`. The `get` statement gets each *X*, where *X* is the entity type pointed to. The pointer's control fields and `the` *X* are kept. The `compress` statement compresses on the control fields, computes `X__ptr` as above, and finally prints the control fields and `X__ptr`. The user can run `canon`'s output as is, or augment it to compute summary information. The double underscores are a simple-minded way to make names generated by `canon` look different from names chosen by the user.

To compute the Customer data used in this paper, the canonical query shown above was modified by hand to compute `Call_count`, `total_minutes`, and `total_miles`.

```
get each Call
     keep .minutes as time, .miles as distance,
          the Call as Call__inbound, .telno as telno ;
initialize Call__ptr = null(Call),
     tdistance = 0, ttime = 0., count = 0;
compress on telno
     count = count + 1,
     ttime = ttime + time,
     tdistance = tdistance + distance,
     Call__ptr = if ( Call__ptr = null(Call) )
          Call__inbound else Call__ptr
then
     print telno, count, ttime, tdistance, Call__ptr ;
```

Pointer construction queries such as the above are sometimes used with the *UNIX* system command `join` to construct data files. For example, one might join the Customer data above to the results of a survey to make a more elaborate Customer file, or the output of two pointer construction queries could be joined to produce `Customer` data with, say, a list of Calls made and a list of Equipment installed.

V. EXPERIENCE

5.1 Positive

5.1.1 Size

At the time of this writing, the largest database accessible with Datastream contains about 70M bytes. A little more than half of that is original data; the rest was constructed by Datastream queries. Some databases are smaller than the original data. In the most dramatic case a file of 50M bytes turned out to be 60-percent blanks and insignificant zeroes in numeric fields. A query that simply printed every field reduced the original file to an equivalent one with only 20M bytes. This is common with data that analysts acquire from systems that use fixed-length fields.

5.1.2 Speed

Query processing performance is adequate, although users would be happier if every query finished in five seconds. Timing experiments found that Datastream's basic overhead on a VAX 11/780 computer running University of California, Berkeley *UNIX* 4.1 BSD was about nine microseconds of CPU time per byte of data read. Basic overhead was measured by timing a query that does nothing but count the records in a file. This query ran about 40 percent faster than the `cat` command on the same file with standard output ignored.

In another performance test, an expert programmer wrote C programs that answered three simple queries. These programs required 1 percent, 40 percent, and 250 percent more CPU time to process a 20M-byte file than was needed for equivalent Datastream queries. The programmer took about 40 minutes to write his programs, compared to ten minutes for the queries. The queries were chosen to show Datastream in a good light, of course, and they were written by the author of this paper, an expert query writer. When an analyst was asked to write such queries, his performed slightly *better* than the author's.

5.1.3 Use of pipes

Datastream's use of separate statement-handling processes communicating through pipes made implementing the query-processing code simpler than it might have been otherwise, but it carried a risk of bad performance. Specifically, we were concerned about the possibility of thrashing among statement processors competing for the same CPU, and swapping of pipes competing for the same system buffer space. The first has not been a serious problem because of the large (10K-byte) buffers used in the statement processors. In general, a statement processor has room to read or write 10 to 100 mainstream records or data file lines at a time. This means it can do considerable processing without giving up the CPU. Second, we have never found swapped pipes in our performance experiments. Pipes do not fill beyond a system-specific limit (usually 10K bytes), and processing a query requires at most nine of them at any one time. Most queries need four pipes or fewer. Our experience has been on machines with three or four megabytes of main memory, and these systems have had no trouble accommodating the pipes. Of course, the same large memories support large buffers in the statement processors as well.

5.1.4 Use of an intermediate code

The decision to separate the query compiler from the statement processor with an intermediate code turned out to be a good one. It has simplified development, debugging, maintenance, and tuning: to date no bug has required changes to both modules, and several times major changes have been made in one without affecting the other. A human expert can read the intermediate code without much difficulty, and that has proved helpful in debugging.

5.1.5 Query language

Users find that it takes some work to learn the control structures, especially `compress on`. Once grasped, however, the language seems easy to use. It allows analysts to concentrate on specifying computa-

tions without spending much time on the details of I/O or control structure.

5.2 Negative

5.2.1 Updates

The switch to ordinary files has made it much easier to correct errors or append new data than it used to be. However, derived data (summaries, pointers) sometimes account for a large fraction of an analytical database (as much as 50 percent), and it has to be rederived when the underlying data changes. This can be a real nuisance; some applications have decided not to use Datastream because they needed to do too many updates. Others have considered the possibility of using Datastream for queries and other utilities or ad hoc code to handle updates.

5.2.2 Connections

From time to time an application would like a more powerful mechanism for connecting files than the sort-and-point technique available now. For example, data files are often sorted on the control fields for some pointer, and secondary indices into such files would be helpful for some applications. Some extensions in this direction seem reasonable and may be implemented in the future.

5.2.3 Functions

Some systems (notably S[3,4]) put procedures supplied by users in processes separate from the basic code. This simplifies installation and protects the basic code from name conflicts and bugs in the code supplied by users. Unfortunately, it is difficult to make such a scheme work efficiently in Datastream because overhead for context switching and interprocess communication can be large compared to calling a function. At present we maintain two versions of the statement processor. One is the basic statement processor with no functions at all, while the other is the basic statement processor loaded with a set of functions that are shared by all the Datastream users on a given system. The shared library had 17 functions at last count. It can probably grow to several times that size before it becomes unmanageable. Datastream also provides a utility that allows analysts to make statement processors loaded with personal libraries of functions that can be called from queries.

Another problem with functions results from evolution: `printf` clauses and expressions of the form `null`(*entity-type*) look like function calls, but they do not behave exactly like functions. Fortunately, this inconsistency does not seem to bother the users.

VI. SUMMARY AND POSSIBLE IMPROVEMENTS

Experience with Datastream has shown that a language designed to simplify access to large analytical databases can be useful in a *UNIX* system environment. Datastream meets that need except for some defects and omissions. Its users routinely use it to get information from files that are five to ten times larger than we thought possible when work on Datastream began. In spite of some initial reservations about the architecture of the query-processing software, we have achieved an efficient implementation. The conversion to ordinary files has been well worth the effort; it eliminated much redundant use of disk space and allowed Datastream to work with, rather than against, other software tools.

Datastream might benefit from a facility for efficient random sampling from data files and a way to define (not just call) functions in queries. New data structures and connection mechanisms could be supported by constructing a set of statement processors, each with a different routine for handling data files. This is feasible because the query-processing software is almost independent of the low-level file handler. The new structures could be fully updateable, at least in principle, and the new file handlers might even be those of some other database system.

VII. ACKNOWLEDGMENTS

I would like to thank the original Datastream team of Ed Fisher, Steve North, and Ray Yanofchick; others who provided suggestions and encouragement, including John Chambers, Dave Belanger, Rick Becker, John Walden, Bob Kayel, Bill Keese, and Jim Downs; and the courageous users of the prototype: without them none of this would make any difference.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. D. Swartwout, "How Far Should a Database System Go? (to Support a Statistical One)," Proc. Second Int. Workshop on Statistical Database Management, Los Altos, Calif., September 27–29, 1983.
3. R. A. Becker and J. M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Belmont, Calif.: Wadsworth, 1984.
4. R. A. Becker and J. M. Chambers, "Design of the S System for Data Analysis," Commun. ACM, *27*, No. 5 (May 1984), pp. 486–95.
5. S. C. Johnson and M. E. Lesk, "Language Development Tools," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 2155–75.
6. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 1971–90.
7. P. P.-S. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," ACM Trans. Database Syst., *1*, No. 1 (1976), pp. 9–36.
8. N. Wirth, "The Programming Language PASCAL," Acta Informatica, *1*, No. 1 (1971), pp. 35–63.

AUTHOR

Don Swartwout, B.A. (Mathematics) 1974, Kalamazoo College; Ph.D. (Mathematics) 1979, University of Michigan; AT&T Bell Laboratories, 1979–1985; AT&T Information Systems, 1985—. Mr. Swartwout's dissertation dealt with the mathematical foundations of database systems. At AT&T Bell Laboratories, he has worked on concurrency control and query languages for database systems, programming environments, source-to-source programming language translation, and compiler development. He continues working on compiler development at AT&T Information Systems.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

HEQS—A Hierarchical Equation Solver

By E. DERMAN* and E. G. SHEPPARD†

(Manuscript received April 9, 1984)

HEQS is a set of tools for numerically solving sets of algebraic equations from their description in a text file. It allows users to compactly define (or alter an already defined) set of equations (a *model*), and then analyze and solve it with minimal intervention. HEQS automatically checks the algebraic and logical consistency of the equations, reports errors, and allows users to correct the errors by simply changing the original textual description of the relevant equations. HEQS can deal with unsubscripted or multiply subscripted (array) variables. Because it relieves users of the need to perform repetitive algebraic substitution and evaluation, it is most useful for sets of equations involving from tens to thousands of variables. HEQS commands can be used (1) interactively, to define and solve models, or (2) as a set of *UNIX*™ operating system high-level algebraic tools for building applications that require model solving. This paper describes the motivation and design of HEQS, illustrates its use, and highlights its underlying algorithms.

I. INTRODUCTION

1.1 HEQS: A modeling environment

HEQS (Hierarchical Equation Solver) is a package of C and *UNIX* system shell programs that allows users to interactively define, debug, modify, and obtain the numerical solution to models described by sets of algebraic equations in a text file. The equations it handles can involve unsubscripted or multiply subscripted (array) variables. Used interactively, HEQS programs provide a *modeling environment* for end users to build and solve models. Used from within shell scripts or menus, they form a set of high-level algebraic tools for builders of applications that require model-solving capabilities.

* AT&T Bell Laboratories; now with Goldman Sachs & Co. † AT&T Bell Laboratories; now with Asymetrix Corp.

HEQS frees users from tedious and repetitive checking of the logical consistency of equations, from algebraic substitution, and from numerical solution. For this reason, it is most useful for repetitively solving a set of equations while changing some data values or equations, or for solving large equation sets involving from tens to thousands of variables. Even for just a few equations, however, HEQS makes it easy to enter them as text and directly obtain the solution.

1.2 Nonprocedural model solving

The input or primitive of HEQS is a *model*—an easily edited text description of a set of equations (and associated comments) to be solved. Models are most naturally kept in *UNIX* system files. Their equations can be written in any order the user finds conceptually useful. HEQS itself finds an order, or *hierarchy*, for the numerical solution of the equations, hence its mnemonic name.

To analyze, find the errors in, or solve models, users invoke simple one-word HEQS commands. Despite the logical and algebraic operations HEQS performs on the model, all subsequent HEQS output about model errors or solutions refers to the original user description; this localizes for users the source of the error, and so suggests the necessary correction or modification. This feature, that users need deal only with their model description entered in any order, without specifying a method for analysis or solution, makes the equation solving system *nonprocedural.*

HEQS is therefore useful for mathematically naive users, who cannot themselves solve small (but perhaps complicated) sets of equations, as well as sophisticated users, whom it frees from repeated algebraic and numerical analysis of large models that are frequently changed.

1.3 HEQS under the UNIX *operating system*

HEQS programs are tailored to the *UNIX* system and its shell.* The programs can be used in the two modes:

1. As simple high-level commands to solve models, or
2. Combined with the control structures of the shell, to provide a model-solving language for use by applications system builders that require equation-solving capabilities for their users.

1.4 Main features

HEQS provides

1. A shorthand *language* for compactly describing large sets of algebraic equations involving multiply subscripted variables,

* The commands are all implemented as separate programs that communicate with each other through intermediate files, analogous to the *UNIX* Source Code Control System package.

2. An *interpretive nonlinear solver* with predefined and user-definable functions,

3. Extensive checking for common logical and algebraic errors in the description of models, and for numerical problems that occur during their solution,

4. *What-if analysis* (determining the numerical effect of a change in some model equations),

5. *Goal-seeking* (determining what data values guarantee particular output values),

6. *Sensitivity analysis* (determining the variation in variables of interest when the data changes), and

7. *A model compiler*—that is, an automatic code generator that produces a C program for rapidly solving a particular HEQS model with user-specified data values. This allows applications developers to provide numerical solution of particular models more efficiently than the interpretive HEQS solver does for general models, and furthermore allows the applications program to execute on machines that do not run HEQS.

Graphics, report generation, etc., may be obtained by linking HEQS output to other *UNIX* system tools.

HEQS is more flexible and powerful than standard commercially available "spreadsheet" programs, which usually handle only fixed-length time series of limited algebraic complexity. HEQS accepts large sets of equations for subscripted or unsubscripted real variables involved in both simultaneous and nonlinear relationships, and has more error-detection facilities.

This paper is arranged as follows. Section II contains some varied illustrations of the use of HEQS. Section III describes the current status of HEQS. Section IV briefly describes the motivation for building HEQS, and the design. Section V gives an overview of the implementation, and Section VI discusses error reporting, an important feature for users. Finally, Section VII describes our conclusions. An Appendix on the algorithms and data structures used is included.

II. USING HEQS: EXAMPLES

This section contains three fairly lengthy transcripts of HEQS sessions run under the *UNIX* system. They illustrate, in successively more complex examples, some of the features of HEQS.

In order to annotate the transcripts with explanations, we adopt the following convention. Boldface text below denotes HEQS commands being invoked by the user. A $ sign denotes the computer prompt. Italicized text before or after a HEQS command contains explanatory remarks about the command and the reason for its invocation; these remarks are not part of the transcript; they explain why particular

commands are being invoked. Finally, typewriter text denotes the computer's response.

2.1 Defining, correcting, and solving a simple model

This example illustrates the solution of a few equations kept in the file *badmod*. The original model contains some errors, which are successively found and corrected with the HEQS system's help.

```
$ cat badmod        Read the model in the file badmod.

# This file badmod contains a simple model with errors
# written in HEQS equation notation
# This is a comment because the line starts with a # sign
        /* This is also a comment because it started with a
           /* and ends with the following */
# Here are the equations
# Note that some equations are simultaneous and that they
#   are
# written in an arbitary order which is not the order of
#   solution.
B = 2*C - D + 1
D = 4
A = (B+C)//D + sqrt(E)
  # note: typo - two divide signs
C = 3*B - 2*D
  # note: no equation for E

$ cheq<badmod       This command reads the model into HEQS and
                    checks equations for errors.
error in line 3:
A = (B+C)//

CANNOT PROCEED!
CORRECT MODEL  AND REENTER.
The model has been corrected—look at it in file bettermod1.
$ cat bettermod1

B = 2*C - D + 1
D = 4
A = (B+C)/D + sqrt(E)
  # note: typo fixed
C = 3*B - 2*D

  # note: no equation for E
$ cheq<bettermod1       Read correct model into HEQS.
$ canislv       Check consistency of the model—can it be solved?
```

```
CANNOT SOLVE FOR:
        A
BECAUSE NO DATA OR EQUATIONS AVAILABLE FOR:
        E
$ cat bettermod2        Look at edited and recorrected model.
B = 2*C - D + 1
D = 4
A = (B+C)/D + sqrt(E)
        # typo fixed
C = 3*B - 2*D
E = -1
       # added missing equation
       # but sqrt(negative) is illegal

$ cheq <bettermod2      Read in next corrected version of model.
$ slv       Solve the model.
D=4
C=3.4
B=3.8
E=-1

An argument to a mathematical function is outside the legal
range in equation
       A=(B+C)/D+sqrt(E)

Probable origin in model is
       line 6: equation: A = (B+C)/D + sqrt(E)

$ cat bestmod       Look at final corrected version.

B = 2*C - D +1
D = 4
A = (B+C)/D + sqrt(E)
C = 3*B - 2*D
E = +1
       # made E positive

$ cheq <bestmod     Read final version into HEQS.
$ slv       Solve final corrected version.
D=4
C=3.4
B=3.8
E=1
A=2.8
```

```
$ whatif          Now tinker with model using HEQS command 'whatif'.
          D=5
D=5
C=4.4
B=4.8
E=1
A=2.84

$ sens A D=5      Ask how sensitive A is to D at D=5 using 'sens' com-
                  mand.
A +1.0% change in variable "D" causes a
          +0.1% change in variable "A"
```

Use 'repslv' command to see how A varies with D by repetitively solving the model for several D values.

```
$ repslv A D=1,2,3
D[1]=1    A[1]=2.2
D[2]=2    A[2]=2.6
D[3]=3    A[3]=2.733333333
$
```

2.2 Solving a projectile problem

This example illustrates the kinematics of a projectile fired vertically upwards from the earth's surface. The model's equations are defined in the file *rocket_eqs*. HEQS is then used interactively to explore what the model predicts. The **goalsk** command, which solves the equations implicitly, is used to find the time at which the projectile reaches the apex of its trajectory.

```
$ cat rocket_eqs
/* variables    */

# s - distance above earth's surface in feet
# v - final velocity in ft per sec

/* parameters  */

# u - initial velocity in ft per sec
# t - time of flight in seconds
# a - acceleration (due to gravity) in ft per sec sq

/* equations  */

s = u*t + 0.5*a*(t**2)
v = u + a*t
```

```
/*  initial parameter values  */
t = 1
a = -32
u = 88
```

```
$ cheq <rocket_eqs        Read model into HEQS.
$ slv        Solve it for initial parameter values.
a=-32
t=1
u=88
s=72
v=56
```

Find value at t=2.

```
$ whatif
     t=2

a=-32
t=2
u=88
s=122
v=24
```

Find value of t at which velocity v is zero—i.e. projectile is at apex.

```
$ goalsk
     t:v=0

a=-32
v=0
u=88
t=2.75
s=121
```

Use 'repslv' command to see how v varies with t
by repetitively solving the model for several t values.

```
$ repslv v t=0:3:13

  t[1]=0          v[1]=88
  t[2]=0.25       v[2]=80
  t[3]=0.5        v[3]=72
  t[4]=0.75       v[4]=64
  t[5]=1          v[5]=56
  t[6]=1.25       v[6]=48
  t[7]=1.5        v[7]=40
  t[8]=1.75       v[8]=32
```

```
t[9]=2          v[9]=24
t[10]=2.25      v[10]=16
t[11]=2.5       v[11]=8
t[12]=2.75      v[12]=0
t[13]=3         v[13]=-8
$
```

2.3 A complicated financial model

Table I illustrates the use of HEQS on a toy financial model kept in the file *bearmod*; it includes many comments, and should be self-explanatory. This model uses features of the HEQS language for describing multidimensional arrays, for writing conditional (IF-ELSE) equations that depend upon Boolean values, special built-in functions like SUMOF that sum elements of arrays, and macro definitions that make the model easily maintainable.

Note that the model is truly nonprocedural: equations are written to describe the financial relationships in an order that makes them easy to understand, but not necessarily easy to solve. HEQS provides the intelligence to determine an order and method for solution.

III. PRESENT CAPABILITIES

HEQS' main virtue is to allow the easy definition, alteration, and solution of models like the ones given above. The characteristic feature of such models is that although they may involve hundreds or thousands of variables, they decompose into interdependent irreducible subsets of simultaneous equations, each subset involving only a few (generally less than ten) variables.*

Analyzing and solving the 112-variable model of Section 2.3 on a Western Electric 3B20 Simplex or Digital Equipment Company VAX 11/780 minicomputer running the *UNIX* System V operating system requires less than ten seconds of user plus system CPU time, as reported by the *UNIX* system command `time`. About six seconds of this time is devoted to reading the model into the system (via the HEQS command `cheq` described in more detail later) and analyzing it to find an order for solution. The numerical solution itself, invoked by the `slv` as illustrated in the examples of Section II, requires the remaining four seconds of user plus system CPU time.

Whenever any equations are altered or added to the model, subsequent analysis, error-checking, and solution still require at most this amount of time. Changing the model means simply reediting the

* Although HEQS imposes no actual limit on the number of variables in a simultaneous subset, it was not designed to efficiently solve irreducible subsets of simultaneous equations involving hundreds of variables. The financial and market models for which it has been typically used so far have all satisfied this criterion.

original model file. Prior to the existence of HEQS, a model of this type would have been solved by writing an ad hoc procedural Fortran or C program of approximately a thousand lines, or by writing special preprocessors to commercially available spreadsheet programs with less power. This would have required at least several hours of work, plus a similar amount of maintenance time for any significant alteration to the model. Altering or expanding a HEQS model requires no new programming; the savings in time and labor gained by using HEQS is therefore obvious.

For larger models with thousands of variables, HEQS' running time increases roughly in proportion to the number of equations in the model, as well as the equations' complexity. For example, a one-thousand-variable model requires 37 seconds of user plus system time to pass through `cheq`, and 51 seconds to be solved via `slv`. Changes to equations or data values in the model can be made interactively and the model again solved in similar amounts of time. The advantage of using HEQS therefore becomes even more apparent.

Although 51 seconds is not a long time to wait for a solution when one is developing or refining a model, it may be too long when that refined model becomes part of an application program. For such cases HEQS provides a command `compmodel`, described in Section V; `compmodel` compiles a completed model into an efficient C program that solves the model for different data values much more quickly, as described in Section 1.4. Compiling the thousand-variable model mentioned above results in a reduction of solution time from 51 seconds to 3 seconds, an appreciable decrease that ensures that the response time of the compiled solving program is more than adequate for applications end users.

Future HEQS enhancements may include integrating these equation-solving capabilities with reports and graphics functions, which users of HEQS must currently obtain by passing their solutions to other *UNIX* system tools.

IV. MOTIVATION AND DESIGN

4.1 History

HEQS arose out of the desire to automate the often tedious development and maintenance of Fortran-based financial modeling programs being written or planned in our organization in 1979/1980. We realized[1] that the programs being written were really solving an algebraic representation (a *model*) of some corporate financial structure under varying end-user assumptions, and we aimed our efforts at automating this process.

The implementation of these programs was the result of many error-susceptible interactions between analyst end users and programmers.

Table I—Three bear model

```
$cat bearmod
############################################################
#                    THREE BEAR MODEL
############################################################

        /*
        * This is a financial model for the bear family.
        * Once upon a time there were three bears: poppa, momma and baby bear.
        * Every winter they hibernated, every fall they were unemployed.
        * To survive, they needed 400 pounds of porridge a month.
        * Each month they could use both their savings and earnings
        * to buy porridge.
        * This model lets them determine how their survival each month is
        * affected by prices, salaries, etc.
        */

        #       A definition of the family, months and seasons follows.
        #       The operator ≪ used below, as in t≪1, denotes a shift of
        #       the time series t to the left by one unit; thus, if t is
        #       the series 1 through 12, t≪1 is the series 0 through 11.
        #       Note how the DEFINE statements below allow easy maintenance.
        #       Adding an extra bear to the whole model, or altering the definition
        #       of winter simply requires changing a DEFINE statement to propagate
        #       change through the whole model.
```

```
############################################################
DEFINE BEARS          poppa momma baby
DEFINE MONTH          1:12
DEFINE PREV(t)        t<<1    /*  this defines the macro function PREV()  */
DEFINE SUMMER         MONTH > 5 && MONTH < 9
DEFINE FALL           MONTH == 9 | | MONTH == 10
DEFINE WINTER         MONTH > 10 | | MONTH < 3
############################################################

        /*  Note that the equations below have been entered in     */
        /*  the order suitable for thinking about the model,       */
        /*  NOT the order for solving it.                        */

        /*  Note also that although income[poppa,MONTH] and        */
        /*  income[momma,MONTH] for any MONTH in the SUMMER        */
        /*  are involved in simultaneous sets of equations,        */
        /*  the user need not worry or even be aware of this.      */
        /*  HEQS will figure this out and solve it.              */
        /*  Finally, note that when the left-hand side of an       */
        /*  equation involves a subscripted variable like          */
        /*  income[poppa,MONTH], many equations are being          */
        /*  simultaneously defined. This is much easier and        */
        /*  less procedural than writing something like          */
        /*     FOR i in 1 to 12                                  */
        /*     DO                                                */
        /*       income[poppa,i]=. . . .                           */
        /* DONE                                                   */
```

Table I—continued

```
#-------------------------------------------------------------------------------------------
#                    INCOME STATEMENT
#-------------------------------------------------------------------------------------------

monthly_income[MONTH]      =    SUMOF income[BEARS,MONTH]

income[poppa,MONTH]        =    IF      (WINTER || FALL)
                                THEN    0
                                ELSE    100 - penalty_2inc * income[momma,MONTH]
                                /*******************************************************
                                poppa's income diminishes when momma earns and
                                he must babysit; penalty_2inc is the coupling
                                factor between the two incomes.
                                *******************************************************/

income[momma,MONTH]        =    IF      (SUMMER)
                                THEN    0.25 * income[poppa,MONTH]
                                ELSE    0
                                /*******************************************************
                                momma works only a quarter of the time poppa
                                does, and only in summer
                                *******************************************************/
```

```
income[baby,MONTH]          =   0

bank_balance[MONTH]         =   bank_balance[PREV(MONTH)]
                                + monthly_income[MONTH] - money_spent[MONTH]

money_spent[MONTH]          =   porridge_needed[MONTH] * price_of_porridge

# ------------------------------------------------------------------
#            FOOD AND SURVIVAL CONSTRAINTS
# ------------------------------------------------------------------

survival[MONTH]             =   IF ( survival[PREV(MONTH)] == 1 &&
                                porridge_avail[MONTH] >= porridge_needed[MONTH] )
                                THEN    1
                                ELSE    0
                                / ******************************************************
                                a value of 1 means they live ;
                                0 means death by starvation
                                ******************************************************/

porridge_needed[MONTH]      =   IF (WINTER)
                                THEN  0          /*  hibernation  */
                                ELSE  400

porridge_avail[MONTH]       =   (bank_balance[PREV(MONTH)] + monthly_income[MONTH]) /
                                price_of_porridge
```

Table I—continued

```
# ------------------------------------------------------------------
#                    DATA
# ------------------------------------------------------------------

price_of_porridge          =    0.25
penalty_2inc               =    0.1     /*  estimated penalty factor for 2 incomes  */
bank_balance[0]            =    0       /*  initial bank balance  */
survival[0]                =    1       /*  they start out alive  */

$ cheq <bearmod      Initial HEQS command to read the model from its file.
$  canislv      HEQS command to ensure all unknowns can be solved for.
$  slv      HEQS command to numerically solve the model.
   . . . . .
survival[0]=1      Only the answers of interest, not the whole solution, are shown here.
survival[1]=1
survival[2]=1
survival[3]=1
survival[4]=1
survival[5]=1
survival[6]=1
survival[7]=1
survival[8]=1
survival[9]=0
survival[10]=0
survival[11]=0
survival[12]=0
   . . . . .
```

```
$ whatif        HEQS command to speculatively change and solve model.
        price_of_porridge = 0.26
   . . . . .
survival[0]=1
survival[1]=1
survival[2]=1
survival[3]=1
survival[4]=0
survival[5]=0
survival[6]=0
survival[7]=0
survival[8]=0
survival[9]=0
survival[10]=0
survival[11]=0
survival[12]=0
   . . . . .
$
```

System requirements and model equations were elicited from analysts. Maintenance was driven by the frequency of the analysts' needs to change the model to reflect actual and permanent regulatory, legal, accounting, and corporate structure changes. Finally, analysts often wanted to make speculative and temporary changes to models to investigate their assumptions in forecasting.

We decided to create tools that would let analysts do their work without having to rely on programmers. Since their underlying need was to solve an algebraic model, we started to design a system that would solve models from a high-level description of its structure and equations. Such a system would let analysts define, solve, and alter their models as their needs dictated; it would also let programmers who still built applications for analysts do their work more easily and rapidly.

The first system built was EXEC, a 1980/1981 prototype automatic code generator. It produced hard-coded Fortran subroutines to calculate the values of variables in a model by accessing a database of corporate and financial model equations previously set up by an administrator. EXEC was implemented in the *UNIX* system shell language and could solve only nonsimultaneous equations.

It soon became apparent that it was preferable for users to build and solve models interactively, since model refinement is an iterative process requiring frequent debugging. In contrast to EXEC's compilation of models into Fortran subroutines, HEQS was therefore designed as an interpreter that directly solved models. Its specifications and some design features are described below.

4.2 Requirements

To allow nonprogrammers to develop and solve models, we required that HEQS' only primitive should be a text file of equations. However complex the algebraic analysis and solution of the model, HEQS users should only have to deal with a straightforward text description of the model. Users should be free to write equations and comments in any conceptually useful order, in a notation as close to common algebra as possible.

We further required that HEQS provide interactive model definition and solving, with extensive error checking in all phases of modeling. Errors that might prevent the complete solution of a model had to be reported to the user in terms of the responsible equation or problem in the original model. Error correction should require only that the user correct by reediting the original high-level description of his model. We also decided that HEQS should make *no* attempt to remedy user errors in their models by guessing at their intention. Responsibility for the meaning of a model should lie with the modeler only. HEQS should provide power, but not at the expense of safety.

These requirements for interactiveness led us to implement large parts of HEQS as an interpreter. This has the obvious advantage of making it easy to provide good debugging, and the disadvantage of slower execution (i.e., numerical solution) of debugged models. We judged this worthwhile, but in fact later added a `compmodel` command, described in Section V, that allowed the production of efficient C programs to solve particular models.

The objects of interest in financial and other types of modeling are often multidimensional structures, for example, the revenues of a company for each division and each year. Such structures are well represented by subscripted variables (arrays) and so HEQS was required to accept equations for both subscripted variables (sometimes called *tensors* in the discussion below) and unsubscripted (*scalar*) variables, and provide a variety of operators for manipulating them. Since not all functions that users might need could be foreseen, HEQS had to allow users to define their own functions. It needed numerically to solve sets of nonlinear simultaneous equations for these variables. Finally, to cater to the investigatory nature of modeling, it needed to solve models in a whatif, goal-seeking, and sensitivity mode, as defined in Sections 1.4 and V.

Since we expected that HEQS would be used as a high-level applications-building language, we also required it to be compatible with other *UNIX* tools. This led us to implement the whole system as a collection of programs, each performing a simple analytic step in model solving, with each program returning appropriate error codes to indicate successful or unsuccessful task completion. Applications developers could then build larger modeling systems quickly by embedding HEQS commands in shell scripts under the control of the shell.

4.3 Design

4.3.1 Solving sets of equations

Our aim was to provide a modeling environment where users maintain and solve their own models without programmer invervention. We therefore tried to make HEQS mimic human algebraic reasoning, analyzing, and solving systems of equations in a manner similar to the way people would. In this way, when an error occurs, preventing the complete solution of a model, it can be localized for users at a point close to where they would have discovered it themselves.

How would one solve large sets of algebraic equations* of the form

lhs_variable = expression?

* From now on, "equation" always means an equation of the form *left-hand-side variable = expression*, as distinct from the more general algebraic equation *expression = expression*. The "*left-hand-side variable*" above may, however, be an array or tensor implicitly involving several scalar variables.

As an illustration consider the set

$$a = 2b - c, \tag{1a}$$

$$b = 3a + c^d, \tag{1b}$$

$$c = 4d - e, \tag{1c}$$

$$d = 7c + 3, \text{ and} \tag{1d}$$

$$e = 4. \tag{1e}$$

One would carry out the following steps.

1. Check the algebraic syntax of the equations.
2. Analyze the variable dependencies. In this example, we note that

a depends on b and c;
b depends on a, c, and d;
c depends on d and e;
d depends on c;
e is known.

3. Determine those irreducibly simultaneous subsets of variables whose equations must be solved simultaneously. Here,

a and b are irreducibly simultaneous,
c and d are irreducibly simultaneous,
e is (trivially) irreducibly simultaneous.

4. Find an order for successive numerical solution of the simultaneous subsets which guarantees that, as each subset is solved, all unknowns on its right-hand side have already been solved. Here,

e is known,
thus d and c can be evaluated,
finally a and b can be evaluated.

5. Solve the subsets numerically in this order.

These are the steps HEQS should carry out.

HEQS uses a graph-theoretic approach[2] to model analysis and error checking, and a variable substitution algorithm for the numerical solution of equations.[3] For nonlinear equations, the variable substitution is supplemented by an iterative approximation.[4] These together provide a good paradigm for simple human analysis, error detection, and solution.

To provide a formalism for model analysis and solution, models in HEQS are internally represented as directed graphs[5] and are described in greater detail in the Appendix. The illustrative set of equations

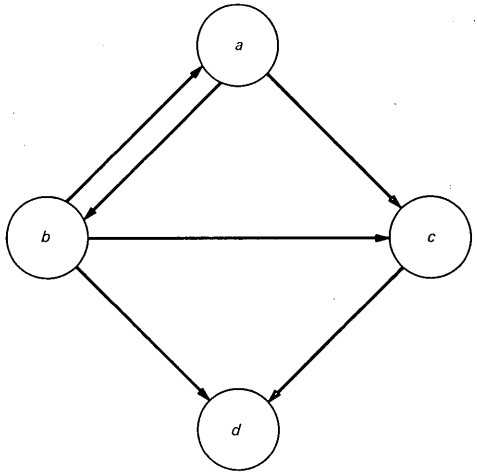

Fig. 1—Directed graph displaying the dependency structure of the model in eq. (2).

$$a = 2b - c, \tag{2a}$$

$$c = d + 2, \tag{2b}$$

$$d = 3, \text{ and} \tag{2c}$$

$$b = 3a + c^d \tag{2d}$$

is represented by the directed graph in Fig. 1. Here, graph vertices correspond to variables, and directed edges or arrows correspond to the equal sign that shows the dependency of left-hand-side variables upon the right-hand-side variables in their defining equations. Strong components (sets of vertices such that any two are connected to each other by paths in either direction) correspond to sets of irreducibly simultaneous variables whose equations can be solved only with simultaneous solution techniques. Given this correspondence between equations and graphs, graph theory provides a rich source of algorithms for analyzing and solving models. Some of these are listed below.

1. Strong component algorithms are useful for isolating simultaneous subsets of variables.

2. Depth-first search from graph roots is useful for determining the order in which these subsets can be solved.

3. Depth-first search from a vertex corresponds to tracing the effect of changing the value of a variable (i.e., the vertex) upon other variables in the model, that is, doing "what-if" analysis.

4. Algorithms for finding paths between vertices in the graph can be used to help solve the implicit equations that occur in goal-seeking. These algorithms are described in more detail in the Appendix. In

brief, graphs provide a formal way of representing human thinking about models.

The graphical representation and strong component algorithms also allow HEQS to subdivide the large model into a sequence of smaller ones, each more easily inspected and solved. This lets HEQS give human-oriented error messages. Helping users debug large models precludes solving a model by inverting a matrix for all the equations, or using any other technique that treats the whole model as one irreducible set of equations. A matrix inversion solver is inadequate in any case because models of interest are generally nonlinear and sparse.

4.3.2 Modular architecture

The basic architecture of the system was chosen to model corresponding steps in the equation-solving process. It is displayed in Fig. 2. Details of the system are contained in Section V. Its front end was a macro preprocessor that parsed and then *scalarized* compactly written scalar or tensor equations into their scalar text equivalents; it then abstracted and stored in a graph (as described above) the dependent (left-hand-side) variable and its independent (right-hand-side) variables for each scalar equation. These scalar equations and their graph were then passed to a module that decomposed them (using strong component algorithms) into the smallest possible sets of simultaneous equations, and determined an order (the *hierarchy* of the mnemonic "HEQS") for their solution. A numerical solver then found the solution to the scalar equations passed to it in the appropriate order. The advantages of this modular approach are listed below.

1. It allows the crucial separation of the "scalarizer," which defines the language for writing equations, from the solver, which finds the solution. This makes possible the independent enhancement or replacement of either, and has already proved extremely useful.
2. Each module can be independently developed, with one module assigned to a programmer.
3. Each module can be independently debugged owing to the weak coupling of all modules, which communicate only by intermediate files. These files are also useful in finding programming errors.
4. Modeling can be conveniently halted when errors occur, because each module performs a limited analytic task.

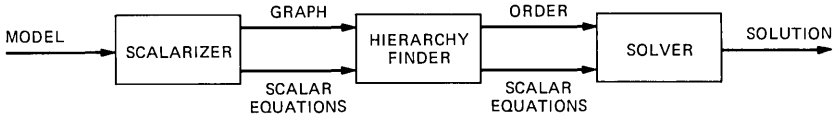

Fig. 2—HEQS functional architecture.

The main disadvantage of this approach is suboptimal time and space efficiency owing to information transfer between modules, but this can easily be dealt with by further development and integration if necessary.

V. HEQS ARCHITECTURE

We now present a brief description of the architecture of HEQS, and the commands available. The commands in HEQS are implemented as separate programs, described below. Their interactions via intermediate files are displayed schematically in Fig. 3. In this figure, rectangles denote HEQS commands, ellipses denote intermediate files, and arrows show the information flow.

`Cheq` (CHeck EQuations) is the primary gateway into HEQS' modeling environment. It reads equations and comments from a file or standard input, parses them, reports syntax errors, scalarizes as necessary, and builds a dependency graph for future model analysis. `Cheq` has full macro capabilities and understands tensor notation, allowing

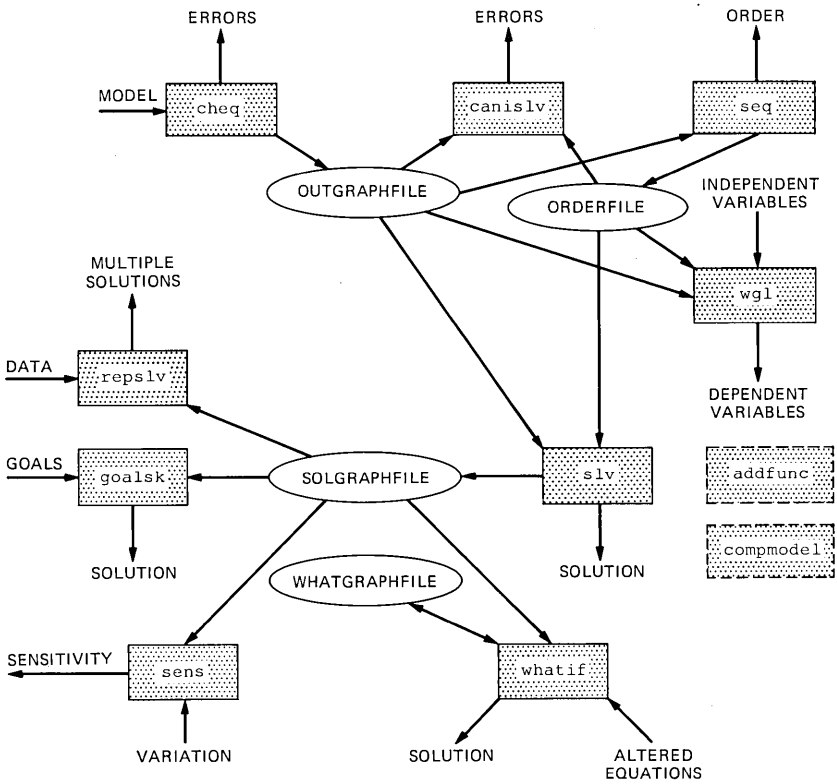

Fig. 3—HEQS architecture.

powerful and compact representation of large equation sets. The equation language it accepts was illustrated in Section II. The dependency graph that contains all model information is stored in an intermediate file (the *outgraphfile*) for use by other programs, whose first action is almost always to reconstruct all relevant information about the model by opening this file.

`Seq` (Sequence EQuations) decomposes the model entered via `cheq` into linked simultaneous subsets of equations and then determines an order for solution that allows them to be solved subset by subset. The order is stored in another intermediate file (*orderfile*) for use by `slv` below.

`Canislv` (CAN I SoLVe) uses the order for solution in *orderfile* to test whether the model is underdetermined. It reports all variables that do not appear on the left-hand side of some equation in the model (and thus cannot be solved for), as well as variables whose values cannot be found because they depend upon the value of such variables. It is intended to provide model builders with interactive help in completing their model.

`Slv` (SoLVe) applies a numerical solver to the simultaneous subsets of the model in the order determined by `seq`, and reports the solution. The graph corresponding to the solution is left in an intermediate file (*solgraphfile*). If the solution fails for some reason (unknown functions, unspecified variable values, overflows, illegal function arguments, nonconverging iterative solution to a nonlinear equation, etc.), `slv` reports the problem and the simultaneous subset in which it occurs. In this way, it pinpoints the nature and location of the model error, and so suggests (one hopes) how the user can correct it.

`Slv` has several additional features. It contains a library of common mathematical functions it can evaluate when they occur in a model's equations. It solves nonlinear equations by iteration, and requires that users provide an estimated initial value for any variable whose value is found in this way. Finally, HEQS has facilities for customizing `slv` by linking user-defined C subroutines to its library (see `addfunc` below).

`Heqs` (Hierarchical EQuation Solver) is a short shell program utilizing `cheq`, `canislv`, and `slv` to parse, check for consistency, and solve equations in a file. It provides a simple one-word command for new users to solve and debug models, and illustrates how *HEQS* commands can be used with the shell to write model-solving scripts.

`Whatif` (WHAT-IF) determines the numerical effects of tinkering with a model by changing its equations. It accepts from the standard input a small set of new equations to replace speculatively some equations in the model, determines what variables of the model need be recalculated, and finds the value of these variables again. It leaves

the altered model and its solution in a new intermediate file (`whatgraphfile`). This is used for doing successive `whatif`'s or `goalsk`'s (described below) on models already altered as well as on the original model.

`Goalsk` (GOAL-SeeKing) allows the determination of the data values in the model that guarantee particular output values for the solution. Models normally contain all unknowns of interest on the left-hand side of defining equations, with the understanding that these unknowns are to be determined. `Goalsk` allows users to specifiy (1) a target value for a set of left-hand-side unknowns, and (2) a set of right-hand-side variables for which they would like to know values that guarantee the targets. HEQS then tunes the right-hand-side variables to the appropriate values, and reports them. This is equivalent to finding the numerical solution of implicit equations in the model.

`Sens` (SENSitivity) calculates for users the percentage change in the values of each of a set of model variables that result from a specified percentage change in another specified variable. Its effect is similar to that of several `whatif` commands in succession.

`Repslv` (REPetitive Solution) repetitively solves a model for a range of different input values for data variables. It is analogous to a Monte Carlo solution of a model.

`Wgl` (WigGLe) does variable impact analysis on a model. Given one or more variable names as arguments, it reports the names of variables whose values are affected by changes ("wiggles") in the values of the arguments, owing to the implicit dependence induced by the model's equations. It is intended to aid users interested in analyzing the effects of tinkering with a model.

`Addfunc` (ADD a user-defined FUNCtion) allows users to add their own C subroutines defining their own functions to the numerical solver used by `slv`, `heqs`, `goalsk`, `whatif`, `repslv`, and `sens`. It produces an extended version of the `slv` program for personal use by the user. Since it is a "meta" command that modifies the HEQS `slv` command rather than solving a user's model, it is displayed as a dashed box next to the `slv` command in Fig. 3.

`Compmodel` (COMPile a MODEL) produces a compilable C program that solves a particular model more rapidly than the standard HEQS `slv` command interpretively solves a general model. It too is a meta command.

VI. ERROR REPORTING IN A MODEL ENVIRONMENT

Programs written in compiled procedural languages may contain either compile-time or run-time errors. Models in HEQS may analogously contain compile-time errors of logic or syntax detectable during

analysis, or run-time errors corresponding to numerical problems found by the solver as it tries to proceed through the simultaneous subsets of equations. In either case, our main design criterion has been to ensure that HEQS reports solubility problems in terms of the original user equations that seem to cause the error, despite any analysis and transformation the model may have undergone in the system's graph-theory algorithms. HEQS makes no attempt to fix models by using its own "intelligence" in place of the modeler's. We believe that detecting and reporting a large class of errors to users in the language of their original model will adequately enable them to fix things themselves. With this in mind, we list below some of the model errors found by the system and explain where they are reported.

Syntax errors in the model's equations are detected by `cheq` while parsing and scalarizing the equations. Since HEQS scalarizes tensor equations and has full macro capabilities, `cheq` has an option that lets users see the equivalent scalar equations that constitute their model after scalarization and macro expansion. This feature is useful for advanced users making liberal use of `cheq`'s compact notational power. `Cheq` reports all syntax errors in a model; it only builds a dependency graph for error-free models.

Semantic errors (errors that make a model "meaningless") are detected in `cheq`, `canislv`, and `slv`. Over-determined models—models where a variable is used as the left-hand side of more than one equation—are caught and reported in `cheq`. Under-determined models, in which some equations or data values necessary for complete solution of the model are missing, are detected in `canislv` and `slv`. `Canislv` has been found to be especially useful for detecting typographical errors, since mistyped variable names often lead to under-determined models.

Unknown functions are detected in `slv`. `Cheq` assumes all functions in a model are available in the function library in `slv`, so that such errors are caught during numerical solution after model analysis. Functions called with the wrong number of arguments are detected here too.

Mathematical errors involving underflow, overflow, functions called with illegitimate argument ranges, etc. are detected in `slv`. Its error messages report the offending scalar equation and function or operation, the original model equation from which it stems by scalarization, the data values used in the model equation, and the equations in the simultaneous subset to which they belong.

Nonlinear equations are detected by `slv`, which first eliminates any nonlinearities that can be removed by linear solution and substitution. To avoid the problem of finding the "wrong," that is, unwanted solution to a nonlinear equation with multiple solutions, users are

then asked to provide an estimate of the answer to be used as a starting point for iterative solution. Problems involving nonconvergence or the absence of a real solution are reported here too.

Typographical errors are found only when they lead to some insolubility like those listed above. Here, HEQS' detailed output in terms of the original user equation is usually adequate for localizing the error.

VII. WHAT WE LEARNED

HEQS has shown that it is feasible to give users an easy way of writing, maintaining, and solving large sets of algebraic equations. Its language, with its subscripted variables and macro facilities, allows powerful, user-definable, easily maintainable and yet *compact* description of algebraic relationships. Its error-checking facilities help even naive users correct their models. Its solver provides nonprocedural solutions to many algebraic problems of interest.

The decision to implement HEQS as an interpreter rather than a compiler seems successful, since our users work in a field where models change quickly, and easy model building and consequent error correction is crucial. The interactive debugging provided by an interpreter is particularly important; it is hard to imagine users with a nonprogramming background having the patience to debug models that are solved by first translating them into a procedural language, then compiling them, and finally running the resultant program.

The graph-theoretic internal representation of models has provided a natural paradigm for equation solving by humans. It offers a standard mathematical way of representing all the analytic steps of modeling. We have found that even mathematically naive users seem to have an intuitive grasp of dependency trees. This suggests that a two-dimensional display of the dependency trees would provide a useful interface to the modeling commands.

The separation of the parser/scalarizer from the solver in the implementation has allowed flexible development. The parser and the solver were modified or rewritten independently at several times during the upgrading of HEQS. In each case, integration into the system as a whole was simple. The possibility of building special-purpose front ends for particular areas of modeling without affecting the overall system is attractive.

The disadvantage of this separation is subtle, and was slow to emerge. Since the solver sees only scalarized equations, it has no understanding of tensors, which are parsed only by the scalarizer. Thus, user-defined precompiled subroutines linked to the solver can only accept scalar arguments. It is not possible for users to define their own subroutines that take tensor arguments, since this would

require dynamically modifying the precompiled parser, an unrealistic goal. In practice, however, the addition of full macro capabilities to the HEQS parser allowed users to define macros that take tensor arguments, which eliminated much of the problem.

A final important lesson in implementation was the benefit of HEQS' modular design. HEQS programs were small, each performing one analytic task. They communicated via intermediate text files that held the relevant data structures. The weak coupling implicit in this design allowed different programmers to work on different modules with little conflict and much independence, and inspection of the intermediate text files aided debugging. Now that the system has stabilized, it may be desirable to join the separate programs into one large one, dispose of the need for intermediate files, and thus obtain a more efficient and tested integrated version.

VIII. ACKNOWLEDGMENTS

C. J. Van Wyk furnished us with algorithms and programs for the solution of algebraic equations by substitution. T.-W. Pao devised a way to allow users to link their own subroutines to HEQS' solver. Both of them provided useful suggestions and conversion. C. Childs and R. Rodriguez gave helpful support and advice.

REFERENCES

1. E. Derman and Z. M. Ma, unpublished work.
2. E. Derman and C. J. Van Wyk, unpublished work.
3. Christopher J. Van Wyk, *A Language for Typesetting Graphics*, Ph.D. dissertation, Stanford University, 1980.
4. D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," J. Soc. Ind. Appl. Math., *11*, No. 2 (June 1963), pp. 431–41.
5. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, New York: Addison-Wesley, 1974.

APPENDIX

HEQS Models As Graphs

This Appendix describes the internal graphical representation of a user model, and explains how it is used to obtain the order of computations for model solution, to perform impact analyses, to eliminate extraneous computations from a what-if analysis, and to reorder model computations to perform goal-seeking. In particular, the method used to reorder the computations for goal-seeking is described in some detail. We first give an intuitive description of the HEQS graphical representation of models, and we later present a rigorous definition.

A HEQS model is a sequence of equations, each of whose left-hand sides is a variable to be computed. Furthermore, a model is considered complete only if every model variable is constrained to appear once

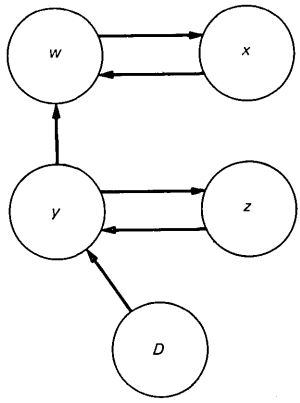

Fig. 4—Graphical representation of the model in eq. (3).

and only once on the left-hand side of a model equation. Thus, the list of equations

$$w = 2*x + y - 10, \tag{3a}$$

$$x = w**0.5, \tag{3b}$$

$$y = z*D - 3, \tag{3c}$$

$$z = y - 5, \text{ and} \tag{3d}$$

$$D = 20 \tag{3e}$$

(where ** denotes exponentiation) forms a legitimate HEQS model. Internally, HEQS represents each model as a directed graph whose nodes correspond to model variables and whose arcs indicate computational dependencies. The graphical representation of the model given above is shown in Fig. 4.

It is important to understand that each node in the graph "points" to the model equation that is used to derive a value for the variable represented by the node. Given the constraints on a HEQS model, that is, each variable occurs exactly once on the left-hand side of an equation, this node to equation matching is implicit within the model equations themselves. Later, in conjunction with the discussion of goal-seeking, we describe a method for performing this matching process when no constraints are placed on the form of equations, so that both sides of an equation may be arbitrary expressions.

The value of a variable corresponding to a node can only be determined when all the right-hand-side variables in its equation are known. For the graph, this implies that the computational ordering of a model (the order in which it can be solved) must satisfy the constraint

The value of a model variable is determined only after all variables connected to it by in-arcs have been evaluated.

This statement is, of course, not quite complete, as can be seen in the model of eq. (3). Here, the variable w cannot be evaluated until the variable x has been solved and vice versa. In other words, the variables w and x form a set that must be computed simultaneously, that is, produce a simultaneous set of equations. In fact, the nodes corresponding to w and x form a strong component of the graph representing the model (a strong component of a directed graph is a set of nodes such that there is a path from any node in the set to any other). It should be clear that any simultaneous set of equations in a model must correspond to a strong component in the graph representing that model. Thus, to obtain the computational ordering of a model, we first collapse the graph of a model into strong components. For our example model, we obtain the directed graph of strong components depicted in Fig. 5. The computational ordering of a model can then be fully described by the statement

The value of the variables in an individual strong component (simultaneous equation set) cannot be determined until all strong components connected to it by in-arcs have been evaluated.

Thus, we must select the set of strong components in an order satisfying this statement and solve them in that order. (A depth-first search of the collapsed graph provides one suitable order.)

Using this graphical representation to determine the computational

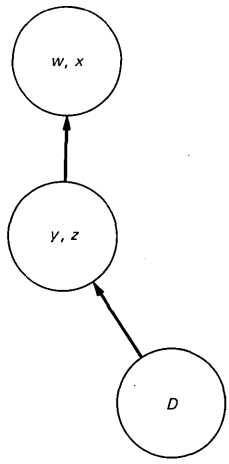

Fig. 5—Collapsed graph corresponding to the model in eq. (3).

ordering for obtaining a model solution has several advantages over other methods.

1. Some seemingly nonlinear computations can be easily solved through substitution (i.e., without using nonlinear solution methods). In the example, the equations $y = z*D - 3$ and $z = y - 5$ form a linear (not a nonlinear) simultaneous set since the variable D is evaluated prior to the evaluation of y and z (i.e., the strong component containing D is solved prior to the strong component containing y and z).

2. This method isolates computational errors (division by zero, overflows, underflows, etc.) to the equations in which they occur. For example, the equation $x = y/(a + b)$ could result in an error if $a = -b$, and not otherwise. Detecting these types of errors helps localize logical errors within a model.

3. The time required to solve a linear simultaneous set of equations with a linear substitution method (the method used in HEQS) varies in proportion to the number of atomic arithmetic operations (addition, subtraction, etc.) in the set of equations. This is a clear improvement over treating a model as a large matrix in which the number of operations increases with the square of the number of model variables. (Of course, matrix methods have many other disadvantages. For example, they are implicitly linear.)

4. When a truly nonlinear simultaneous set of equations does occur (as with w and x in the above example), the smaller the set, the faster the convergence of any iterative solution method. Furthermore, solving two nonlinear simultaneous sets independently will, in general, be faster than solving the two sets as a single, larger simultaneous set. (We use Marquardt's iterative method of solution for nonlinear simultaneous sets[5]).

5. The extraction and ordering of strong components is fast since it is linear in the number of nodes in the graph.

The HEQS graphical paradigm of a model is useful in several other respects. For example, a common question to ask about a model is, "If I change the value of this variable, what other variables in the model are changed (impacted)?" This type of question is referred to as impact analysis. Our graphical representation provides a straightforward and fast method for performing such an analysis. The inverse of an impact analysis (what variables produce changes in a particular variable) is also quite easy and fast. Furthermore, both these capabilities allow the elimination of extraneous computations in a so-called what-if analysis. An example of the type of question posed in such an analysis would be

> *What is the effect on corporate profits if the price of product A rises by 10 percent?*

The variables which must be recomputed to solve this what-if question are exactly those that lie on a path from the "price" variable to the "profit" variable. All other computations may be excluded.

Another common capability required in a modeling system such as HEQS is goal-seeking, which is, in an intuitive sense, the inverse of the what-if capability. For example, a typical question posed would be

> *What must the price of product A be to increase corporate profits by 10 percent?*

Given our model in eq. (3), this is similar to asking, "What value must the variable D be given so that the constraint $w = 15$ is satisified?" That is to say, we wish to find a solution to the set of equations

$$w = 15, \tag{4a}$$

$$w = 2*x + y - 10, \tag{4b}$$

$$x = w**0.5, \tag{4c}$$

$$y = z*D - 3, \text{ and} \tag{4d}$$

$$z = y - 5. \tag{4e}$$

(Note that the equation $D = 20$ has been removed from the model, and the equation $w = 15$ has been added. These model perturbations are characteristic of the goal-seeking problem.) In this goal-seeking analysis, we refer to the w as the goal-seeking variable, and to D as the target variable. Our problem, then, is to determine an order of computation for the new goal-seeking model. Ideally, we would like to obtain this computational ordering by a sequence of simple transformations applied to the graph representing the original model. We present such a method whose most complex step is discovering a path from one node to another.

The previous method for ordering computations (forming a graphical representation using the implicit matching of nodes to equations) in a HEQS model fails for this goal-seeking problem for an obvious reason: the new model violates the constraint that each model variable occur exactly once on the left-hand side of an equation (w is on the left-hand side of two equations while D is not on the left of any equation). What we require is an algorithm for obtaining the matching between the model variables represented by nodes in the graph, and the new set of model equations. Before giving a precise description of our goal-seeking algorithm as applied to the general case, we first illustrate the steps involved by reordering the computations in the goal-seeking problem given here. Later, we give a rigorous description of this process.

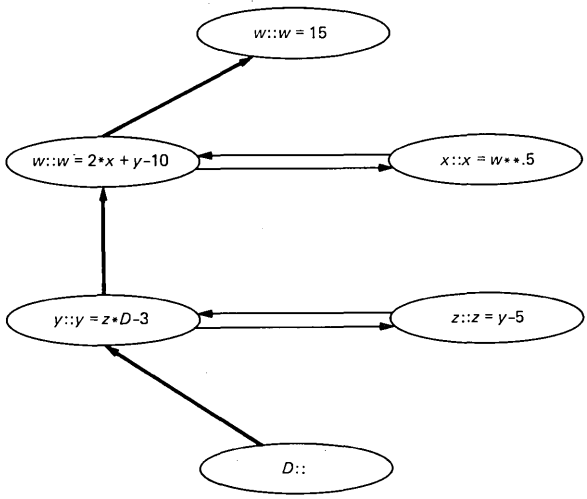

Fig. 6—Graph for goal-seeking in model of eq. (4).

To determine a computational order for goal-seeking using graphs, the first step is to remove the equation corresponding to the target variable D in the graph of Fig. 3, and then add a new node that describes the new goal-seeking constraint. The resulting graph is depicted in Fig. 6. Note that in this figure, the equations associated with each node have been placed in the graph nodes, separated from the variable corresponding to the node by two colons. Also note that only the equation $D = 20$ has been removed from the graph, not the node containing the variable D.

We now proceed to find a path from the node corresponding to the target variable D to the newly added node that contains the goal-seeking constraint equation. The arcs in one such path are shown in bold in Fig. 6. Having found such a path, we then transform the graph by moving the equation in each node in the path from that node to its predecessor node in the path (the result of this operation is depicted in Fig. 7). This places in each variable node an equation that can be used to derive the value of that variable in the goal-seeking problem. We also retain a one to one correspondence between variables and equations. After having moved the equations "backwards" through the path, we redirect each arc that points to a node in the path so that it is directed to the predecessor of that node in the path instead (the graph arcs affected are marked in bold in Fig. 7). This makes the dependencies between the nodes and their new associated equations consistent. The result of this arc redirection process is shown in Fig. 8.

Finally, we remove the new graph node and all its connected arcs. The resulting graph has all the properties we require for ordering

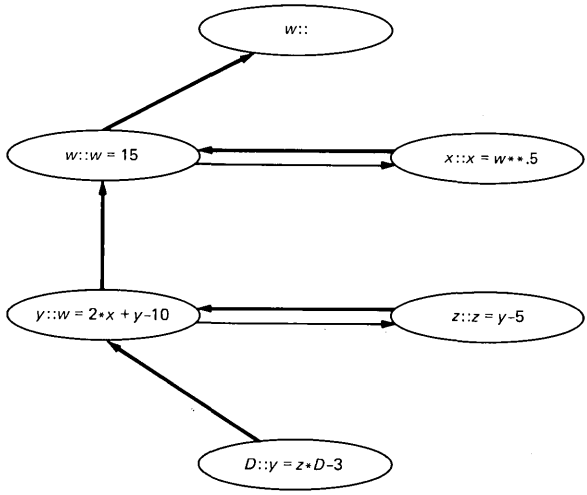

Fig. 7—Transformed graph for goal-seeking in model of eq. (4).

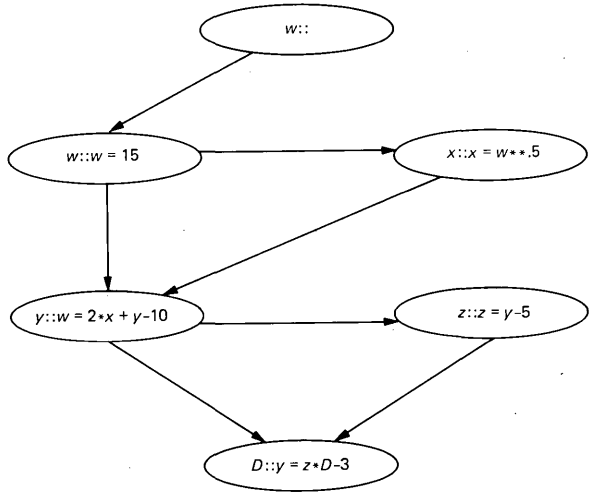

Fig. 8—Final graph suitable for determining the computational ordering of model in eq. (4).

computations in the goal-seeking problem; its strong component collapse and depth-first search yields an appropriate computational order.

We now give a more rigorous definition of model graphs and of the goal-seeking algorithm.

We can conceive of a model **M** as being an ordered triple (**V**, **E**, ∇), where **V** is the set of model variables, **E** is the set of model equations, and ∇:**E** $\rightarrow 2^{\mathbf{V}}$ is a function (from the set of equations to the power set of **V**) defined so that for all $e \in \mathbf{E}$, $\nabla(e)$ is the set of variables

mentioned in the equation e. Given a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, we can develop a graph representing $\mathbf{M}$ as $\mathbf{G} = (\mathbf{N}, \mathbf{A})$, an ordered pair of a node set and an arc set. The individual nodes in $\mathbf{N}$ are ordered pairs (v, e) taken from the product $\mathbf{V} \times \mathbf{E}$ that satisfy the requirement $v \in \nabla(e)$. Given two nodes $n, m \in \mathbf{N}$, where $n = (v_n, e_n)$ and $m = (v_m, e_m)$, then $(n, m) \in \mathbf{A}$ if $v_m \in \nabla(e_n)$. (We understand $(n, m) \in \mathbf{A}$ to mean that the node n "depends" on the node m. Such an arc is drawn from the node m to the node n.) Finally, we say that a graph $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ is a *solvable* representation of a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$ if each variable in $\mathbf{V}$ and each equation in $\mathbf{E}$ occur exactly once in a node in $\mathbf{N}$. Such solvable graphical representations are amenable to HEQS algorithms.

We now give a detailed description of the goal-seeking algorithm used in HEQS. Assuming that we have a solvable graphical representation $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ of a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, the goal-seeking problem is given as

> *Vary the model variable* **TV** *so that the variable* **GV** *satisfies the constraint* **GE**,

where **GE** is an equation in the normal HEQS form and the variable **GV** is the left-hand side of this equation. Clearly, the variable **TV** is the target variable, the variable **GV** is the goal-seeking variable, and the equation **GE** is the goal-seeking constraint equation. If we let **TE** be the model equation that is paired with **TV** in the set of graph nodes **N**, what we require to solve this problem is a solvable graphical representation of a new model $\mathbf{M}' = (\mathbf{V}, \mathbf{E}', \nabla')$, where $\mathbf{E}' \equiv (\mathbf{E} - \{\mathbf{TE}\}) \cup \{\mathbf{GE}\}$, and ∇' is extended from ∇ in the natural manner to include **GE** in its domain. The goal-seeking algorithm is then a sequence of transformations applied to the graph **G**.

1. Form a new graph $\mathbf{H} = (\mathbf{M}, \mathbf{B})$ so that $\mathbf{M} \equiv \mathbf{N} \cup \{(\mathbf{GV}, \mathbf{GE})\}$, and $\mathbf{B} \equiv \mathbf{A} \cup \Delta\mathbf{A}$, where we define $\Delta\mathbf{A}$ as a set of arcs so that $((\mathbf{GV}, \mathbf{GE}),(v, e)) \in \Delta\mathbf{A}$ for all $v \in \nabla'(\mathbf{GE})$. In other words, add to the graph **G** the node (**GV**, **GE**) and all necessary in-arcs pointing to this new node. Note that **H** is not a solvable model representation since **GV** occurs in two nodes in **B**.

2. Find a path $\mathbf{P} = (p_1, p_2, \ldots, p_{|\mathbf{P}|})$ in **H** so that $p_1 = (\mathbf{GV}, \mathbf{GE})$, $p_{|\mathbf{P}|} = (\mathbf{TV}, \mathbf{TE})$, and $(p_i, p_{i+1}) \in \mathbf{B}$ for all $0 < i < |\mathbf{P}|$. Assume $p_i \equiv (v_i, e_i)$.

3. Form a graph $\mathbf{I} = (\mathbf{L}, \mathbf{B})$ from **H**, where

$$\mathbf{L} \equiv (\mathbf{M} - \bigcup_{p \in \mathbf{P}} \{p\}) \cup \{(\mathbf{GV}, \mathbf{GE}), (v_2, e_1), \ldots, (v_{|\mathbf{P}|}, e_{|\mathbf{P}|-1})\}.$$

Note that we have shifted the appropriate equations through the path nodes.

4. Form another graph $\mathbf{J} = (\mathbf{L}, \mathbf{C})$ from **I**, where $\mathbf{C} \equiv (\mathbf{B} - \Delta\mathbf{B}) \cup \Delta\mathbf{B}'$, given that $\Delta\mathbf{B} \equiv \{(n, m) \in \mathbf{B}: n \in \mathbf{P}\}$, and $\Delta\mathbf{B}' \equiv \{(n, m): n =$

$p_i \in P$ and $(p_{i-1}, m) \in \Delta B\}$. Here we have redirected all arcs pointing to nodes in the path **P**. We refer to steps 3 and 4 collectively as *path inversion.*

5. Finally, we remove the excess node (**GV**, **GE**) and all out-arcs pointing away from this node from the graph **J** to obtain the necessary goal-seeking graph $\mathbf{G}' = (\mathbf{N}', \mathbf{A}')$. Note that all in-arcs to the node (**GV**, **GE**) in the graph **J** were redirected in step 3. It should be clear that the graph $\mathbf{G}'$ resulting from this sequence of transformations is a solvable representation of the goal-seeking model $\mathbf{M}'$.

This algorithm will only fail if there is no path as described in step 2. This is in accord with our expectations. Namely, this indicates that the goal-seeking variable is independent of the target variable in the goal-seeking model.

Having developed an algorithm for ordering the computations in a goal-seeking problem, we now note some additional properties of this algorithm. First, we can use the same algorithm to solve simultaneous goal-seeking problems that have the form

> *Find values for the set of model variables* $\mathbf{TV} = \{\mathbf{TV}_1, \mathbf{TV}_2, \ldots, \mathbf{TV}_n\}$ *so that the model variables* $\mathbf{GV} = \{\mathbf{GV}_1, \mathbf{GV}_2, \ldots, \mathbf{GV}_n\}$ *satisfy the constraint equations* $\mathbf{GE} = \{\mathbf{GE}_1, \mathbf{GE}_2, \ldots, \mathbf{GE}_n\}$.

We do this by applying our algorithm repeatedly. In other words, add new nodes for all the variables in **GE**. Then, find a path from $\mathbf{TV}_1$ to one of these new nodes. Invert this path and remove the goal-seeking node on the end of this path. Then, find a path from $\mathbf{TV}_2$ to one of the remaining new nodes, invert it, and remove this goal-seeking node. Repeat this process until both sets of variables have been exhausted. Clearly, this process will fail in several circumstances: if one or more of the goal-seeking variables are independent of all of the target variables, for example. Furthermore, the algorithm will fail for models such as

$$y = 2*x + 5, \tag{5a}$$

$$z = 3*x - 2, \tag{5b}$$

$$x = a - b, \tag{5c}$$

$$a = 2, \text{ and} \tag{5d}$$

$$b = 3, \tag{5e}$$

and the multiple goal-seeking problem is stated as

> *Vary the variables a and b so that the constraints* $y = 7$ *and* $z = 10$ *are satisfied.*

The algorithm fails since there are no two vertex disjoint paths from variables a and b to variables y and z. In other words, y and z cannot be given values independently only by varying a and b.

Another interesting point about this algorithm is that it can be used to create a solvable graphical representation of any model (assuming that the model has a solution), even when the equations of the model are not in the standard HEQS form (every variable is the left-hand side of exactly one equation). For example, the model

$$w + 10 = 2*x + y, \tag{6a}$$

$$0 = x - w**0.5, \tag{6b}$$

$$z*D = y + 3, \tag{6c}$$

$$5 = y - z, \text{ and} \tag{6d}$$

$$D = 20 \tag{6e}$$

is closely related to our first example model. In fact, it is exactly the same model, with the same solution, but the equations do not conform to the standard HEQS form. To get a solvable graph for this model, $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, we use a variation of the goal-seeking algorithm. In the variation, we will admit nodes in the graph that possess a model variable, but do not have an associated equation. We call such nodes *improper* (*proper* nodes are those which have both a variable and an equation). All improper nodes will be written as (v, ω), where we use the character ω to indicate the presence of a null equation (or null pointer in computer science terms).

The algorithm proceeds in steps. First, select an equation $e \in \mathbf{E}$, and pick an arbitrary element $v \in \nabla(e)$ to be the temporary left-hand side of this equation. Generate the graph node (v, e). For all other variables in $\nabla(e)$, we create improper nodes for these variables, and make arcs pointing from these nodes to the single proper node (v, e). At each succeeding step, we select an unexamined equation from the model. For each such equation e, one of three cases will apply:

1. None of the variables in $\nabla(e)$ occur in the graph in any node (neither proper nor improper).
2. At least one of the variables in $\nabla(e)$ occurs in an improper node in the graph. Assume that (v, ω) is such a node.
3. All of the variables in $\nabla(e)$ occur in proper nodes.

In the first case, we proceed as in the first step of this process. Select one of the variables in $\nabla(e)$ as the temporary left-hand side of the equation e, generate a proper node for this variable, make improper nodes for the rest of the variables in $\nabla(e)$, and add the appropriate in-arcs to the proper node.

In the second case, place e in the improper node (v, ω) to form a

proper node (use e to replace the null equation). Then attach the necessary in-arcs to this node (adding improper nodes for those variables in $\nabla(e)$ that have not yet been placed in the graph).

In the last case, we temporarily add a node (v, e) for some arbitrary $v \in \nabla(e)$ and any necessary in-arcs to this node. We then find a path from some improper node to this temporary node, invert this path, and finally remove the temporary node and any associated out-arcs.

At each step in the process described above, each variable that has been added to the graph occurs in exactly one node. When all equations have been examined, the process ends, and we are left with a solvable graphical representation of the model.

Of course, if the model equations are underdetermined, some improper nodes will exist in the final graph, and the model cannot be solved. If, at any point in the algorithm, case 3, occurs, and no path can be developed, the equations are over-determined, and the model again cannot be solved.

At some future point, the form restrictions placed on HEQS equations will be relaxed so that arbitrary expressions may occur on both the left- and right-hand side of any equation, and this algorithm will be used to derive a solvable representation of the model.

AUTHORS

Emanuel Derman, B.Sc. (Applied Mathematics), University of Cape Town, 1965; M.S. (Physics), Columbia University, 1968; Ph.D. (Theoretical Particle Physics), Columbia University, 1973; 1973–1979: Postdoctoral research in structure of elementary particles at University of Pennsylvania, Oxford University (England) and The Rockefeller University, 1973–1979; Assistant Professor, Dept. of Physics, University of Colorado, 1979–1980; AT&T Bell Laboratories, 1980–1985; Goldman Sachs & Co. Mr. Derman's primary fields of interest are languages and artificial intelligence.

Edward G. Sheppard, B.S. (Mathematics), Emory University, 1980; M.S. (Comp. Sci.), Emory University, 1980. 1980–1983: AT&T Bell Laboratories, 1980–1983; Bell Communications Research, Inc., 1984–1985; Asymetrix Corp., 1985—. Mr. Sheppard's primary field of interest at AT&T Bell Laboratories was application software design and implementation.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

IFS—A Tool to Build Integrated, Interactive Application Software

By K.-P. VO*

(Manuscript received April 9, 1984)

The Interpretive Frame System (IFS) is a tool for creating application software with sophisticated interactive interfaces. IFS is based on the notion of a frame network. A frame network consists of many interconnected modules called frames, each of which represents a logical activity in the system. Frames are written in a high-level language. Besides the usual computational constructs such as conditionals, loops, or arithmetics and Boolean expressions, the IFS language also includes facilities for building program/program interactions, such as subprocess invocations or coprocess communications, and constructs for building user/program interactions such as menus or forms. IFS is a suitable tool to integrate existing programs by providing a uniform and easy-to-use user interface. It can also be used to build a new system in a top-down manner by first defining the network of frames and their interactions and user interface, then programming problem-specific parts. Therefore, it provides a general framework supporting any combination of top-down and bottom-up software development methodologies. This paper gives an overview of the frame network concept, the user interface of frame network systems, the frame programming language, and the IFS system implementation.

I. INTRODUCTION

Much effort in current application software development is directed toward building systems to be used by users with little expertise in computing. These application systems typically combine the functions of many generic computing systems, such as data management and

*AT&T Bell Laboratories.

statistical data analysis. However, the users of such a system often do not care about how the system was implemented, but only how useful it is in their work and how easy it is to use and learn. Therefore, the user interfaces of these systems are usually interactive, and provide users with guidance to the various system resources and their proper use.

There are difficulties in building application software. First, there is a lack of a convenient way to put together collections of programs and control the information flow among them. The problem is further aggravated, since generic computing functions are often available in separate programs but with very dissimilar interfaces. Second, there is a lack of a good tool to create and control sophisticated, interactive user interfaces. Given the intended community of users, it is easily arguable that the major (and hard) part in building a system lies in creating the right interface.

The Interpretive Frame System (IFS) provides a solution to the above problems. An application software system is conceptually viewed as a network of modules called *frames*. Each frame represents a logical activity in the system. The logical activity includes all necessary computing functions, their interactions, and user interactions. Depending on the usage, the network transitions between frames serve two different purposes: to ensure correct execution of cohesive task sequences and to provide guidance in proper use of loosely coupled tasks.

IFS consists of a language to write frames and a system to interpret frame actions. The IFS programming language provides the following:

1. Constructs for user/program interactions, such as menus or forms,
2. Constructs for program/program interactions such as subprocess invocations and coprocess communications,
3. Constructs for transition and passing information among frames, and
4. The usual computational constructs such as conditionals, loops, arithmetics, string operations and expressions to control the interactions of (1), (2), and (3).

The IFS interpreter carries out the actions encoded in frames. It further provides facilities to be used directly by end users to customize the user interface to local requirements and to arbitrarily access frames where appropriate.

IFS is a tool usable to support a large spectrum of software development methodologies.[1–4] At one end, a software product can be developed from bottom up by putting together existing programs and providing a high-level, uniform interface based on frames. At the other

end, a product can be developed from top down by first defining the frames, their interactions, and user interactions before implementing the low-level computing functions. In practice, we have observed compromises in which a prototype is built based on some initial and perhaps incomplete specifications, but with a realistic user interface, and partially supported by existing programs. The prototype is then continuously refined to fill in gaps in requirements and to tune efficiency until the final product results.

The remainder of the paper is organized as follows. Section II describes an architecture for application systems and its relation to the frame network model. Section III first gives an example of a frame network system and its user interface, then gives an overview of frame programming and other IFS system features. Section IV describes the current IFS implementation. Finally, in the conclusion, we discuss some aspects of current IFS usage, problems, and possible future enhancements.

II. APPLICATION ARCHITECTURE AND FRAME NETWORKS

2.1 The architecture of an application

The architecture of an application software system can be roughly divided into four layers: a data component, a set of low-level functions, a structure organizing the functions into logical tasks, and a user interface.

At the lowest layer is the data component describing the data format and its physical storage. Depending on the application, the data architecture can be simple flat files or sophisticated databases or combinations thereof. The next layer is a supporting set of generic functions to act on the data. These functions range from single programs for file listing or sorting to systems for data analysis and graphics or report generation. The elements comprising the first two layers are usually present as services provided by the computing environment. Even if some of these functions are not available, from a software reuse standpoint it is probably worth it to implement them as independent programs because of their applicability. The third layer comprises the computing solutions to the problems of the application. It organizes the appropriate set of generic functions into logical tasks with the right granularity, and controls the information flow among them. Finally, the top layer is the user interface that controls how users will use the system.

In a typical application, the top two layers are where developers should spend the most effort. They form the facade from which users perceive the system and, as such, determine its success or failure. Because of a lack of proper tools, however, the two top layers are often done in an ad hoc manner, resulting in systems that are hard to use

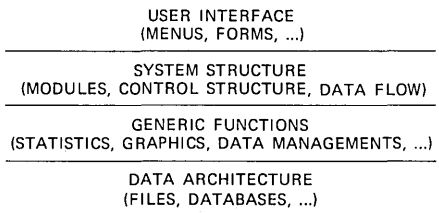

Fig. 1—The architecture of an application system.

and to maintain. IFS solves the problem by providing developers with a robust framework to modularly structure the activities in their systems. Therefore, local changes in the application structure do not necessitate or propagate changes in the entire system. IFS also provides a large and consistent repertoire of tools to customize the proper interfaces among programs, as well as between programs and users.

2.2 Frame networks

A frame network provides an abstraction for the top two layers (see Fig. 1) of an application system. A frame represents the totality of a logical task and all aspects of its internal as well as external interactions. At the program level, it contains all necessary computing functions to carry out the task. At the system level, it controls the information flow among subprograms and to and from other frames. At the user level, it provides users with interactions and guidance to ensure proper inputs and correct execution of the task. A frame can call other frames to perform subtasks or simply transfer control to another frame. The call and control transfer structure forms the frame network connectivity. Thus far, the frame network structure is described similarly to a subroutine network structure in regular procedural programs. This is not always the case. Indeed, for many applications, the sequences of task execution are highly cohesive and in such cases frames do behave as subroutines. However, in many other applications, a frame only represents a high-level abstraction of a task in a collection of loosely coupled tasks. In such a case, the order in which frames are executed is immaterial. The frame network transitions are used mainly to guide users to various resources available in the system. The IFS interpreter in fact allows users to arbitrarily access frames in such cases.

III. SYSTEM DESCRIPTION

3.1 A reminder service: example of a frame network system

In this section, we present an example of how a frame network system is put together on top of a set of programs constituting a reminder service. Then, we show a scenario of using the final system.

The reminder service maintains a database of reminder items. It provides four functions: adding new items, deleting old items, peeking to see items in some time range, and sending out reminders at appropriate times. For simplicity, we shall assume that these functions are implemented as independent programs: `add`, `delete`, `peek`, and `remind`. Their actual implementation details are irrelevant, since we are interested only in their use. For example, to add into the database a new reminder item to be activated on a particular date and at a particular time, `add` is called as

```
add "ContentOfReminder" "date" "time"
```

At the respective date and time, the program `remind` will generate a reminder to be sent via the appropriate media. Note that `remind` is an automatic service of the system that users never have to invoke directly.

As a system of programs, the reminder service can be used directly but not in a transparent manner. For example, the syntax of calling `add` requires the arguments to the program to be in a rigid sequence of positions. Therefore, it is desirable that a more integrated and flexible interface be put on top of the programs. We do this by putting together a frame network on top of the programs `add`, `delete`, and `peek`. The frame network consists of four frames: Reminder, Adder, Deleter, and Peeker connected as a rooted tree. Reminder is the root of the tree and consists of a menu offering the services of the other three frames. The sole action associated with each menu choice in this case is to activate the appropriate frame. Adder, Deleter, and Peeker are frames directly interfacing the programs `add`, `delete`, and `peek`. Their jobs are to collect the necessary information to invoke the underlying programs. For example, Adder is implemented as a form to collect the content of a reminder and its date and time, then to invoke `add` with the collected information. Figure 2 summarizes the architecture of the final reminder system. The reader should compare the layers of Fig. 2 to those of Fig. 1.

The rest of this section shows a scenario of interacting with the reminder service frame system. A few words should be said about the screen organization employed by IFS. The top line of the screen shows the title of the current frame, a piece of information summarizing the function of the frame. Each frame is uniquely known in its network by its Identification (`ID`) string. The second line of the screen shows the stack of IDs of frames traversed to get to the current frame. The IDs are separated by either the colon `:` or the vertical bar `|`. As frames are called, the ID stack grows from right to left so that the current frame ID is the rightmost string (see Figs. 3, 4, and 5). Users of a frame system can observe the frame ID stack and learn the functions

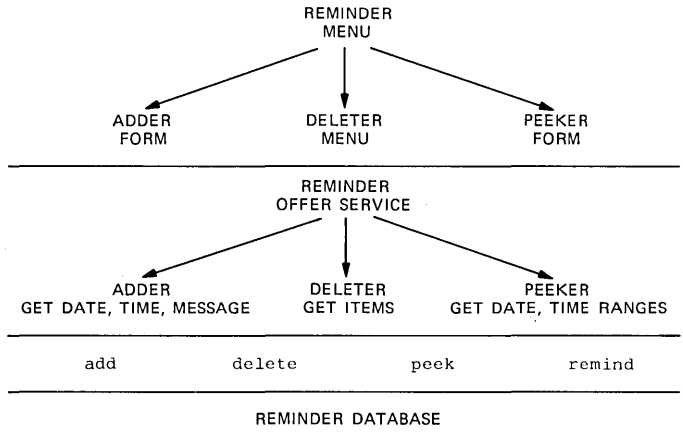

Fig. 2—The reminder service frame network system.

```
Reminder Service                                          Reminder

**********************************
*                                *
*     Please pick a service      *
*                                *
*  a  Add a new reminder         *
*  d  Delete old reminders       *
*  l  List reminders already set *
*  e  Exit.                      *
*                                *
*     <?, ?Choice, Choice>  a    *
*                                *
**********************************


Info Exit Call Goto Return Desc Log Source Unix Wscreen Dflt Mask
```

Fig. 3—Interacting with the reminder service—the Reminder frame.

of different frames and how they are structured in the network. The bottom line of the screen shows global commands that are provided by the IFS interpreter independent of the frame programming. These commands can be accessed by users by typing the escape key (the key labeled with ESC) to make direct use of certain frames in the application system or to customize their interaction environment such as changing the display or setting defaults to repetitive questions. Some of the more interesting commands will be discussed in Section 3.3.

Figure 3 shows the root frame, Reminder, of the Reminder Service. The second line of the screen shows `Reminder`, the ID of the Reminder frame. The middle of the screen shows a menu of available services. Each menu item has two parts: a selector, which is an underlined string, and a brief description summarizing the functions of the item. The parts of the menu prompt <`?`, `?Choice`, `Choice`> indicate that an item is chosen by typing its selector, and further information on a

particular item, if available, can be obtained by typing `?` followed by its selector. The `?` when typed by itself always signifies a request for further information on the current interaction item, in this case, the menu prompt itself. Here, the user had typed `d` to choose the option of adding a new item into the reminder database.

Figure 4 shows the input form of the frame Adder appearing in a window overlaying the window of the menu. A window is a display device used to retain as much of the previous context of interaction as possible. The second line of the screen shows the string `Adder` separated from `Reminder` by a colon, `:`, indicating that Adder was called from Reminder as programmed in the network. Before actually adding a new item, the user decided to check the database for any possible conflict. The escape key was typed to gain access to the global functions. The global command prompt `<?Command, Command, RETURN>` was displayed. Now the user could have used the command `return` to return to the Reminder frame, then pick option `l` to bring up the Peeker frame. However, since the user had used the system before and knew (by observing the ID stack) that Peeker was the right frame to use, the command `call Peeker` was used to directly call up the Peeker frame. The semantics of the global command `call` is to suspend the execution of the current frame and start the execution of the called frame. Thus, Adder was suspended while Peeker executed. This gives an example in which an experienced user of a frame network system can directly access any part of the system, bypassing the programming of the network.

In Fig. 5, upon receiving the user's request for executing Peeker, the execution of the frame Adder was suspended and the frame Peeker was brought up as an input form in another window. The second line of the screen now showed the ID of Peeker separated from the other

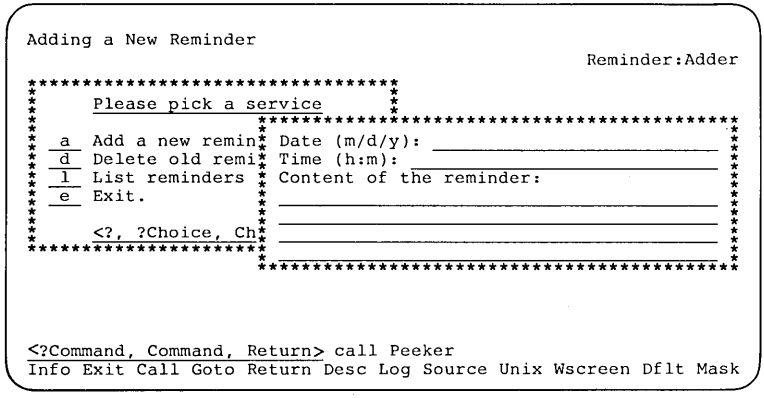

Fig. 4—Interacting with the reminder service—the Adder frame.

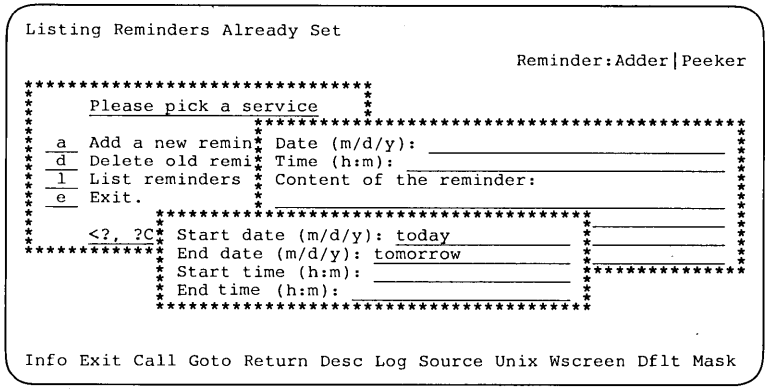

Fig. 5—Interacting with the reminder service—the Peeker frame.

IDs by a vertical bar indicating that Peeker was called directly by a user's request. The user filled in the first two fields of the form, indicating that all reminder items between the present day and the next day were to be shown.

In the remainder of the session, Peeker invoked the program `peek` to extract the relevant set of reminder items from the database and showed them to the user. Then, the Peeker window was popped from the screen and the execution of Adder continued.

3.2 Programming frames

Frames and their interactions are written in a frame programming language. The language provides the following:

1. Constructs for interactions between the program and user such as menus or question-answer form of dialogues,
2. Constructs for interactions among the programs such as subprocess invocations or coprocess communications,
3. Facilities for transition and passing information among frames, and
4. The usual computational statements such as conditionals, loops, and expressions for control over those interactions.

Syntactically, each frame consists of nested labeled blocks where each block represents an "interaction unit." For example, the menu in Fig. 3 of the Reminder Service can be specified by a menu block, called an `{m` block, as in Fig. 6.

In this example, the menu block, enclosed between the pair of strings `{m` and `}m` has a title block (enclosed between `{t` and `}t` and four option blocks (each one enclosed between `{o` and `}o`). Each option block in turn has a title block. The semantics of this menu block is that a menu of four items as shown in Fig. 3 is to be displayed and

```
{m : (0 != 0)
      {t
      Please pick a service
      }t
      {o a
            {t
            Add a new reminder
            }t
            ~call Adder
      }o
      {o d
            {t
             Delete old reminders
            }t
            ~call Deleter
      }o
      {o l
            {t
            List reminders already set
            }t
            ~call Peeker
      }o
      {o e
            {t
            Exit.
            }t
            ~exit
      }o
}m
```

Fig. 6—A Menu block.

when the user picks an option the actions, if any, specified within that option, are to be performed. The sole action for the first option in the above example is `~call Adder`, which will call the Adder frame. The condition `:(0 != 0)` is called an exit condition. When a block (or any other syntactic unit) is executed, it is iterated until the exit condition becomes true. In this case, since `0 != 0` is always false, the menu block has been specified to loop forever. The system will exit altogether, however, when the user picks option `e`.

Each frame description in general is a frame block enclosed between a pair of strings `{f` and `}f`, and has within it several other types of blocks such as menu blocks, action blocks, context blocks, etc. Each block is optionally associated with an entry condition and an exit condition, and has within it an optional title block, a description block and a sequence of other blocks or actions. A partial Backus Normal Form (BNF) specification of the frame language syntax is in Appendix A. To illustrate the simplicity of the language, the complete descriptions of the Reminder frame and the Adder frame of the Reminder Service example are provided in Appendix B. We shall discuss here the semantic issues involved in interpreting the different types of blocks and transitioning among frames.

3.2.1 Frame identification

A frame block is the outermost block of a frame and has the following form:

```
{f frame_ID frame_args
. . .
}f
```

It defines the frame identification and any arguments that the frame may require when it is called. A frame is uniquely identified in its network by the `frame_ID`. Frames can be called directly by the users by their IDs. Thus, frame IDs serve as high-level abstractions of functions in a frame network.

3.2.2 Conditions, loops, and variables

Each block in the language can be optionally associated with two conditions as follows:

```
{. (entry_cond) : (exit_cond)
    . . .
}.
```

Either or both of the conditions can be absent with the default value `true`. If the entry condition is true, the block is executed. After that, it is iterated until the exit condition becomes true. The conditions are logical expressions involving built-in numerical string constants, variables, regular expression matching, and numerical comparison operators. Variables in IFS have the form `variable_name` and are of string type. However, when a variable is involved in numerical computations or comparisons, it is cast into a real number. At that time, if the string value does not represent a real number, it is assumed to be zero. By default, the scope of a variable is the frame in which it appears. It is also possible to dynamically change the scope of a frame variable to be shared with other frames.

3.2.3 Dynamic menus

The format of an option block in a menu block is

```
{o (entry_cond) : (exit_cond) selector
    {t
    Brief description
    }t
    {d
    Online help text for the option
    }d
    Actions
}o
```

The entry condition of an option, if true, signifies that the option is available to be offered. Otherwise, it will be suppressed. Systems such as ZOG[5] provide only static menus, i.e., the list of items in a menu is

fixed by the specification. In IFS, a menu is a dynamic entity that can be customized to the context of user interaction at that instance. An option is selected by users by typing in its selector. Upon the users' selection, the actions associated with the option are executed and iterated until the exit condition becomes true. The actions are composed from other general constructs of the frame language including new menus, forms, transitions to new frames, or process communications. The description block `{d` provides context-sensitive help for the option. The help text can be parametrized by embedding frame variables. Users can obtain help on an option by typing the question mark and the option selector.

3.2.4 Gathering user values

Question blocks provide a mechanism for gathering user values that can be used in subsequent actions. The format of a question block is

```
{q (entry_cond) : (exit_cond) ~input_var
The Question
      {d
      Online help text for the question
      }d
{q
```

The string entered by the user in response to this question is assigned to the variable `~input_var`. The entry condition in a question block, if true at that instance, indicates that the input variable must be set interactively and the question is asked. If the entry condition is false, the question is irrelevant in the current context and is skipped. Thus, if the questions were displayed in a form, their entry conditions define dynamic field protection. The exit condition of a question represents input validation. After the user input is received, the exit condition is evaluated and if it is false, the input is rejected and a new input is requested. To aid input validation, the frame language provides a rich set of comparison operators that includes regular expression matching as well as numerical comparisons. The Adder frame example in Appendix B shows how the values for three variables, `~date`, `~time`, and `~content`, can be obtained from the user before the reminder database can be added with the new data. The only validation check in that case is that the input for `~date` must be non-null.

3.2.5 Process invocation and communication

Problem-specific processing in a frame can be done by invoking in an action block a subprocess which is usually a *UNIX*™ operating system shell program.[6] The simplest form of an action block is

```
{a (entry_cond) : (exit_cond) > ~ret_val1 ~ret_val12 . . .
    shell_program_name argument_1 argument_2 . . .
    . . .
}a
```

The interpretation of the entry and exit conditions is as in other blocks. A new shell is invoked every time the {a block is executed. The lines inside the {a block define a shell script to be executed. There are also mechanisms in the IFS language to allow a subprocess to return values back to IFS. The returned values, if any, are assigned to the return variables ~ret_val's.

Provided some concurrency conditions are met by the invoked processes, the shell can be run as a coprocess as follows:

```
{p (entry_cond) : (exit_cond) "/bin/sh" "-i" >  ~ret_val1
  . . .
~!EndOfInput,EndOfOutput
  shell_program_name argument_1 argument_2 . . .
  . . .
}p
```

In this method, the shell is invoked once to run interactively and stays in the background with its standard input and output channels connected to IFS. The communication protocol between IFS and a coprocess is defined by the control line consisting of `EndOfInput` and `EndOfOutput` strings. The `EndOfInput` string defines a pattern that indicates the end of a message from the coprocess to IFS. A minus sign for that string indicates that no input is expected from the shell coprocess. Similarly, `EndOfOutput` defines a delimiting string to be sent by IFS after the message has been sent. A minus sign again signifies that IFS does not have to send any message delimiter to the shell coprocess after the message is sent. For example, the action block in the Adder frame of Appendix B can be replaced by the following {p block shown below:

```
{p (~date != ~null && ~content != ~null) "/bin/sh" "-i"
~!-,-
add "~content" "~date" "~time"
}p
```

Here, the entry condition shows that the {p block is only executed if either ~date or ~content are not empty. The first time the {p block is executed, an interactive shell program is invoked and put in the background with its standard input and output channels connected to IFS. Each time the {p block is executed, the body of the message contained in the block is sent to the shell coprocess. Subsequently,

the shell executes the program `add` to update the database and then waits for other messages from the frame system. Meanwhile, the frame system resumes its normal activities.

3.2.6 Compound actions

A sequence of related actions can be grouped together in a context block, the `{c` block. The actions of a context block can be other contexts, menus, process communications, dialogues, etc. In particular, a group of questions to be displayed and executed as a form must be together in a context block. Besides the interactive nature of the actions inside a context block, the context block is similar to a compound statement in a procedural language. In the Adder frame in Appendix B, three question blocks and an action block are grouped together in a context block.

3.2.7 Transitions among frames

Transitions among frames are accomplished via the operators `~call`, `~goto`, and `~return`. The operators `~call` and `~return` behave like calling and returning from subroutines in a regular programming language with the additional feature that frames can return multiple values. The action of the first menu option in the Reminder frame is to call the frame Adder. A sequence of `~call` creates a stack of frames. The operator `~goto` restarts such a stack with a new frame.

3.2.8 Other remarks on programming frames

The reader should note that there is no display information involved in the definitions of any of the frames. In IFS, display programming is separated from logical programming. This aspect of IFS contrasts with other screen definition languages in the literature (for example, see Ref. 7) where the system builder has the burden of laying out the complete design of screen displays. As far as a frame is concerned, the parameters that it requires are obtained by asking a series of questions, user choices are obtained by selections from menus, and processing of user data is done by running subprocesses or coprocesses. Navigation in the frame network is performed by IFS actions. Display organization is either done in a default mode by the interpreter or customized by using a display editor or a display language. The combination of all these facilities in a system makes IFS unique in its ability to facilitate the building of integrated and interactive software systems that can be used in a variety of environments.

3.3 IFS system features

An example of interacting with a reminder service frame system was shown earlier. The user of a frame system faces a terminal screen

showing various types of information. The action expected from the user is clearly defined from the interaction context. If a menu is showing, a choice is expected. If a form is showing, some response is expected to fill in the field of the form where the terminal cursor is situated. The user, however, can also initiate other actions by typing either the question mark or the escape key (the key labeled with ESC). The question mark indicates that the user needs more information on the current interaction item and such information, if extant, is displayed by the system. For example, if the current interaction item is a field in a form, the information explaining this field is displayed. The escape key, on the other hand, indicates that the user wishes to make use of the global commands shown on the bottom line of the screen. The set of global commands that can be augmented or partially suppressed by system developers provides functions usable independent of the frame programming. The global commands provided by IFS itself range from customizing the interaction environment of the system to randomly accessing different parts of the frame network. A full description of these commands is beyond the scope of this paper. We present some of the more interesting commands below.

3.3.1 Display editing: `Mask`

The display environment of a frame system can be customized using the global command `Mask`. `Mask` lets users interactively define new layouts for forms and menus as well as windows containing these constructs. For example, to lay out a form, it is necessary to know where on the screen to display field labels, what video attributes to display them in, and whether the fields are left, center, or right justified. To ease the definition of such information, the display editor `Mask` lets users move the cursor freely on the screen to indicate where a label should be drawn. It also presents users with lists of available colors to display different parts of the form.

The ability for redefining the display environment at will is important because different users have different requirements dependent on their local environments; therefore, it is unreasonable to force conformity a priori. Further, for the system builders, having `Mask` makes it easy to experiment with different styles of display.

3.3.2 Default answer setting: `Dflt`

The global command `Dflt` lets users set up default answers to questions or fields in a form. During the execution of such a question or field, the user can choose one of the default answers with a single keystroke or overwrite them by typing a new answer.

`Dflt` is menulike in the following sense. A question or form field is a parameter collector whose domain of values is large in general but

may be restricted for individual users. For example, a question for names has an infinite domain in general but for an individual user, its domain probably has size one. The system builder who solves a general problem must program such a parameter collector for the general case. The command `Dflt` lets individual users tailor such a collector into a personalized menu for transparency and efficiency.

3.3.3 Frame random accessing: `call`

The global command `call` can be used to randomly call any frame in the system, provided that the ID of the called frame is known. The execution of the frame previous to the call is temporarily suspended and is resumed after the completion of the called frame.

The command `call` is one of a family of network movement commands which includes `goto`, `return`, `break`, and `exit`. These commands let advanced users of a system directly access parts of the system independent of the network programming.

IV. CURRENT IMPLEMENTATION

IFS is written in the C language[8] and based on the *UNIX* operating system. It has been used on many flavors of the *UNIX* system, including versions of AT&T System V and University of California at Berkeley 4.1 BSD and 4.2 BSD.

In the current implementation there are four main programs: a frame language compiler (`ifc`), a display language compiler (`ifv`) and a decompiler (`vfi`), and an interpreter (`ifm`). The steps in using these programs to create and run a frame are roughly as follows:

1. Write/edit and compile (`ifc`) a frame language script.
2. Write/edit and compile (`ifv`) a display language script if desired.
3. Interpret (`ifm`) the frame.
4. Modify the display using the global command `Mask` if desired.
5. Decompile (`vfi`) display information into a readable ASCII form.
6. Stop or go back to step 1.

These IFS programs are presented below.

4.1 Frame language compiler: `ifc`

The frame language compiler, `ifc`, compiles frame scripts into intermediate data structures which are interpreted by the interpreter. In the compiling process, the frame scripts are checked for valid syntax and any static text processing is done. Because IFS is interpretive in nature, it is possible to fold the functions of the compiler into the interpreter itself and reduce the complexity in using the system. However, we chose this division to speed up run-time execution. The trade-off is attractive because most frame systems are built once but

would be used many times and certain costs in frame processing such as text processing are significant.

4.2 Display language compiler and decompiler: `ifv, vfi`

Display information of a frame is kept in a separate data structure whose structure closely reflects that of the frame structure. The display structure can be created in two different ways, by using the global command `Mask` (Section 3.3.1) or by writing a display language script and applying the display compiler `ifv`. In an application, it may be desirable to standardize the display of certain collections of information such as on-line help texts. The display language eases the display standardization of many frames since the display script of one frame can be easily modified and replicated for other frames using standard text editors in the operating system. On the other hand, creating the first display description of a series of frames is easier using the `Mask` command than by laying it out on paper and writing a display script. Therefore, `Mask` is usually used to make the first prototype of a display structure. Afterward, the display decompiler, `vfi`, is used to convert the structure into the display script form which can be further processed for other frames using text editors.

4.3 Frame interpreter: `ifm`

The interpreter, `ifm`, executes actions encoded in frames (Section 3.2), controls the display, and processes users' inputs (Section 3.3). It also provides global commands that users can use independently from the frame network programming to customize their local interaction environment (Section 3.3). Internally, the interpreter is divided into two parts: a control part that interprets frame actions, and an interactive part that defines appropriate displays and interactions for different interactive constructs.

The control part of the interpreter consists of routines to get frame and display structures from the file system and routines to interpret the programming language constructs such as menu, form, or network movements. Each construct of the language is interpreted by one routine, and the nesting of constructs is implemented by recursion. Since frame and display structures are kept as files in the file system, there is a cost to read them into memory when a frame is executed. To increase system efficiency, the action code of direct or indirect recursive frames is shared. Further, a marking technique was implemented to retain popular frames in memory for some period of time after their executions are completed.

The interactive part of the interpreter is similar to the control part in structure. It consists of routines to take care of the display and

interaction with different interactive constructs of the language. The routines assumes a minimal capability from the terminals, the ability to move the cursor to any point on the screen.

V. FINAL REMARKS

We have presented an overview of IFS, a software tool to build interactive and integrated software. The main contributions of IFS are the concept of a frame network as a model for structuring application systems and a high-level programming language embodying this model. The programming language includes a large set of interactive facilities such as menus or forms for building user interfaces and constructs such as interprocess communications designed to aid the building of interfaces around existing programs. At this writing, IFS has been in use for a few years. Many frame systems have been built, some with hundreds of frames. These applications span a wide range from analytical systems to office automation systems and systems integrating Computer-Aided Design/Computer-Aided Manufacturing (CAD/CAM) tools. The experience gained in building these applications shows that IFS has been effective in reducing the work between conception and realization of an application system. To conclude the paper, we make a few observations about the current use of IFS, some perceived problems and possible future improvements.

In IFS, interaction programming and analytical programming are separated. The main work of frames is to coordinate the flow of information among the modules and the user. Problem-specific computations are often carried out by other programs. This architecture makes it easy to reuse software in building new systems. Further, it encourages experimentation with high-level modules and user interactions before actual work on the analytical part begins. In fact, we have observed a successful software construction method in which prototypes are built and continuously refined until the final products result. The benefit of constructing software this way is that at any time, the user interface of a prototype is fully functional so that selected users can make trial use and help developers debugging it.

In building interactive systems, a balance needs to be maintained between ease of learning and ease of access. For example, a straightforward menu hierarchical system may be easy to learn at first, but can also quickly become restrictive when users get familiar with the system functions. Within the IFS framework, there is enough flexibility for system builders to make poor judgment and build systems that are awkward to learn or to use. Nonetheless, the applications built using IFS have been generally satisfactory. This is helped in part because the frame network concept and the programming language

are easy to learn. Therefore, many of the applications were built by people who are not expert programmers, but who intimately know the use of the applications and sometimes are users themselves. Further, IFS provides integrally many different mechanisms to build and dynamically control interactions. With the right judgment, the user interface of an application can be made to strike the balance between ease of learning and ease of access. By and large, this has been observed in practice.

As a programming environment, IFS lacks some desirable features. For example, in the programming language, there are currently no explicit ways for system builders to raise or handle exceptions such as asynchronous events like terminal hangup or other operating system signals. The lack of such facilities have made the building of interfaces to certain database operations rather cumbersome. There is also a less frequent need for a debugging tool, especially to examine the state of a frame system upon unexpected computational results. This need has not been acute because frame systems are interactive by nature and their anomalies tend to manifest quickly during trials.

To accommodate a wide range of applications, the current implementation of IFS assumes a minimal requirement on user hardware, a character terminal with cursor addressing. Terminals with bitmap graphics and pointer devices such as mice or light pens are becoming cheaper and accessible to a wider class of computer users. On such terminals, it is desirable to have graphical interactions as well as character-oriented interactions. For example, in many applications, menu items can be better represented with pictures than with texts, and their selections can be done faster with a pointer device than with keyboard typing. The two-part design of the interpreter, separating control from interaction, makes possible improvements in the interaction part with a minimal amount of change in the system entire. Perhaps some future version of IFS will be enhanced to make more use of the new hardware.

VI. ACKNOWLEDGMENTS

Since the conception of IFS, I have been fortunate to benefit from ideas and constructive criticisms from many colleagues and courageous, friendly users of various versions of IFS. I thank in particular: J. M. Chambers, E. R. Fisher, R. G. Kayel, C. M. R. Kintala, G. Perlman, R. M. Prichard, W. J. Shugard, and D. E. Swartwout.

REFERENCES

1. G. D. Bergland, "Structure Design Methodologies," Software Design Strategies, IEEE Catalog No. EH0184-2 (1981), pp. 297–315.

2. F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Trans. Soft. Eng., *SE-2*, No. 2 (June 1976), pp. 237–43.
3. F. Beichter, O. Herzog, and H. Petzsch, "SLAN-4: A Language for the Specification and Design of Large Software Systems," IBM J. Res. Dev., *27* (1983), pp. 558–76.
4. A. I. Wasserman, "Characteristics of the User Software Engineering Methodology," IEEE Proc. Soft. Proc. Work, (February 1984), pp. 125–29.
5. G. Robertson, D. McCracken, and A. Newell, "The ZOG Approach to Man-Machine Communication," Int. J. Man-Machine Studies, *14* (May 1981), pp. 461–88.
6. S. R. Bourne, "The UNIX Shell," B.S.T.J., *57* (1978), pp. 1971–90.
7. L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition," IEEE Soft. Eng., *9* (1983), pp. 31–9.
8. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice Hall, 1978.

APPENDIX A

Partial BNF Specification of the Frame Language Syntax

```
frame ::=              "{f" frame_id frame_args
                            frame_blocks
                       "}f"
frame_blocks ::=       [title_block]
                       [descript_block]
                       {context_block | menu_block}*
title_block ::=        "{t"
                            Title_string
                       "}t"
descript_block ::=     "{d"
                            Description_string
                       "}d"

context_block ::=      "{c"
                            {activity}*
                       "}c"
menu_block ::=         "{m" [cond_pair]
                            {option_block}*
                       "}m"
option_block ::=       "{o" [cond_pair]
                            title_block
                            [descript_block]
                            {activity}*
                       "}o"
cond_pair ::=          [(entry_cond)] [: (exit_cond)]
activity ::=           {context_block | menu_block |
                       question_block | write_block |
                       subproc_block | coproc_block |
                       arithmetics | string_operation |
                       network_transition |
                       change_scope | change_env}*
```

```
question_block ::=    "{q" [cond_pair] ~input_var
                           Question_string
                           [descript_block]
                      "}q"
write_block ::=       "{w" [(entry_cond)] [file_name]
                           Format_text
                      "}w"
subproc_block ::=     "{a" [cond_pair] [proc_args] [>ret_
                             vars]
                           Subprocess_program
                      "}a"
coproc_block ::=      "{p" [cond_pair] program [>ret_vars]
                           ~!End_of_input,End_of_output
                           Message_to_program
                      "}p"
```

APPENDIX B

Figures 7 and 8 are the actual programs for the Reminder and Adder frames of the Reminder Service System.

```
{f Reminder
      {t
      Reminder Service
      }t
      {m : (0 != 0)
            {t
            Please pick a service
            }t
            {o a
                  {t
                  Add a new reminder
                  }t
                  ~call Adder
            }o
            {o d
                  {t
                  Delete old reminders
                  }t
                  ~call Deleter
            }o
            {o l
                  {t
                  List reminders already set
                  }t
                  ~call Peeker
            }o
            {o e
                  {t
                  Exit.
                  }t
                  ~exit
            }o
      }m
}f
```

Fig. 7—The Reminder frame of the Reminder Service.

```
{f Adder
      {t
      Adding a New Reminder
      }t
      {c
            {q : (~date != ~null) ~date
            Date (m/d/y):
                  {d
                  A date can be entered in the format:
                      month/day/year
                  or as 'today', 'tomorrow', and weekdays
                  such as 'monday', 'tuesday' and their
                  abbreviations such as 'mon', 'tue'.
                  }d
            }q
            {q ~time
            Time (h:m):
            }q
            {q : (~content != ~null) ~content
            Content of the reminder:
            }q
            {a (~date != ~null && ~content != ~null)
            add "~content" "~date" "~time"
            }a
      }c
}f
```

Fig. 8—The Adder frame of the Reminder Service.

AUTHOR

Kiem-Phong Vo, M.A., 1977 (Applied Mathematics); Ph.D., 1981 (Mathematics), University of California at San Diego; AT&T Bell Laboratories, 1981—. Mr. Vo is a Member of Technical Staff in the Advanced Software Department. His research interests include combinatorial structures and algorithms, and efficiency issues in software development. Member, ACM, AMS, SIAM.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

T—A Data Management System

By R. J. YANOFCHICK*

(Manuscript received March 1, 1985)

T is a data management system running under the *UNIX*™ operating system that provides unique facilities not commonly found in other data management systems. These powerful data manipulation facilities can access data and programs stored in *UNIX* system files, as well as data stored within a T database. T allows new structure to be added to an existing database without modification of existing data. It also allows multiple views of a database, which can be used to prevent access to privileged data by unauthorized users, as well as to provide some fairly sophisticated restructuring capabilities.

I. INTRODUCTION

T is a hierarchical data management system written in the C programming language[1] to run under the *UNIX* operating system. It was designed to experiment with strategies that would impose structure on existing data and easily modify that structure as needs arose. As a consequence, it reduces data duplication and the programming effort necessary to restructure existing data, while it allows users an appropriate level of control over and access to data. By providing powerful data manipulation and restructuring facilities, T allows users to manipulate and extract data in a form suitable for use by analytical tools available on *UNIX* systems or provided by other users; it thereby permits users to combine specialized tools to build more general and useful tools. Keeping the amount of information necessary to describe a database to a minimum and providing a more natural, understand-

* AT&T Bell Laboratories.

able table of contents form for this information makes setting up a database to be managed by T a simple 15-minute task. The query and command language provided is a simple, yet powerful, English-like language that is easy for even the unsophisticated user to understand.

T has the following unique data management features:

- Dynamic modification of views of data
- Access to data and programs stored outside the database
- Dynamic generation of data
- Query logging and modification.

Often, at least part of the source data to be used in a data analysis process already exists as one or more data sets. Rarely, however, are these data sets structured to be used without modification in a specific data analysis process. The data may be stored by a data management system in a structure not compatible with the needs of the current application. These data sets may also be stored as flat data files. In this case, chances are good that the format of the individual records does not conform to the requirements of the application. Before any real analysis begins, programs must be written that operate on these data sets to extract and reformat the data necessary to perform the analysis. The result is duplication of data. The more data are used by different applications, the more data are duplicated and programming effort is expended to extract and reformat data.

A single database managed by T supports many analytical studies, each having different data access requirements. Users of the system range from staff support personnel with little or no programming experience to analysts with experience in some programming language. Because the uses of data vary widely, and the needs of applications include performance as well as functionality, it is generally agreed that no existing data management system suits the needs of all applications. However, the data extraction and restructuring demands that are placed on such a system can be characterized well enough to provide a general framework for most analytical applications.

The characteristics of the computational demands that would be placed on such a system are not as well understood. Simple commands are provided to enable straightforward information generation. For example, these commands enable the user to input, modify, locate, retrieve, and output data. Other computational needs are not as neatly characterized; either these needs are not known ahead of time or else they are expected to be evolving. For this reason, T focuses on data retrieval, extraction, and manipulation, and on providing a method for accessing existing computational processing functions. By providing access to computational facilities outside of T, the analyst is free to choose those facilities that best satisfy the needs of the application.

For example, data on network configuration can be retrieved from a database and used to produce a graphic display of the network.

T also provides password security for individual databases; redirection of input and output; query logging, which allows users to save sessions for reuse later; concurrency control for one writer with multiple readers; and access to data and programs that are not physically part of the database.

II. SYSTEM ARCHITECTURE

T is implemented as three main modules: a file handler, a primitives module, and a language module (see Fig. 1).

2.1 The file handler

The file handler used by T is a set of subroutines that implement B-trees.[2,3] These subroutines access B-trees via a list of key-value pairs, sorted by key. This list is implemented with a prefix-compressed B-tree in which prefixes common to consecutive keys are factored out. These routines support one writer concurrently with multiple readers, preserving read consistency. This read consistency means that each user retains a consistent view of the database during a query session. That is, during a query session, only those updates made during that session are visible to the user. This read consistency can be crucial to meaningful data analysis.

Communication with the file handler is through a file handler interface. Essentially, these routines simplify the calls to the file handler and, in some cases, slightly modify the functions of some of the lower-level routines. This interface also provides a method for users to access a database directly from C programs.

2.2 The primitives module

The primitives module implements the user-level functions provided by T. These functions include data input and modification, location and retrieval of data, input/output redirection, provision of alternate views of data, and access to standard *UNIX* system functions. For example, these functions allow a user to access data from a data file and programs stored in *UNIX* system files, as well as data stored within a T database; create temporary data files from data extracted from these sources; and then use these as inputs to external programs

Fig. 1—System architecture of T.

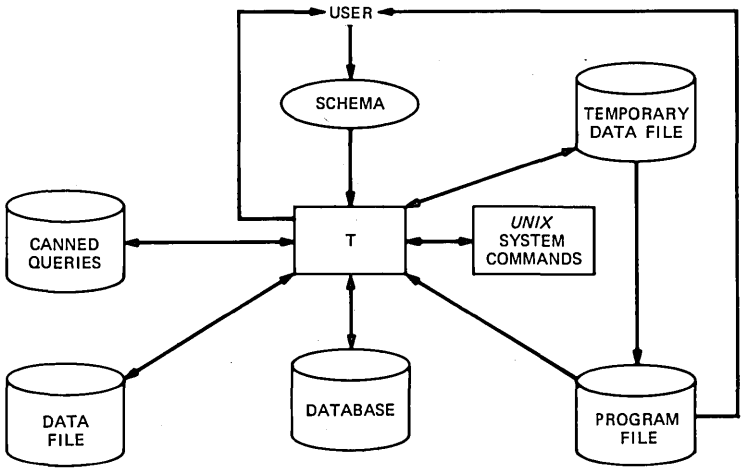

Fig. 2—Communication paths for the primitives module.

or standard *UNIX* system functions (*UNIX* system commands). Figure 2 illustrates the communication paths provided by this module.

2.3 The query language module

As currently implemented, this module accepts input from a user terminal, interprets the query command, checks syntax, and makes appropriate calls to functions in the primitives module. This module is replaceable by other query languages such as HISEL.[4] The query language currently implemented with T is English-like and fairly nonprocedural.

III. SYSTEM ATTRIBUTES

3.1 Database structure

A user-supplied schema file, which represents a hierarchy in a table of contents format, describes the structure of the database to T. A schema file contains a line for each record type (node). Each line contains the name of a node, the attributes associated with that node, and the hierarchical level of the node. The level of each node in the hierarchy is indicated by the number of tab characters preceding the node description: no tabs indicating level 1, one tab indicating level 2, etc. For example, the following schema describes a database for a sales organization:

```
market-segment (name;)
    sales-rep (name;)
        customer (name; sales; address; contact; telno;)
            product-line (name;)
                product (name; price;)
```

Fig. 3—Hierarchical structure of the schema file.

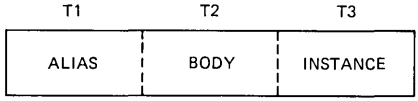

Fig. 4—Key structures.

The semicolon-separated list enclosed by parentheses identifies the fields associated with each node. These are optional. The hierarchical structure for this schema is shown in Fig. 3.

To initialize a database the user supplies the name of the database and the name of the schema file that describes its structure. During the initialization process the user is prompted for a password. This password will be requested whenever the database is accessed using this master schema. An encrypted version of the password and the name of the master schema file are stored within the database. In addition, each node (record type) is assigned a unique numerical alias that is used internally in all query processing and to link nodes when alternate views of the database are requested. Each data record in the database has a unique key associated with it. These keys are generated during data entry. Aliasing node names, which may be long, permits keys to be more compact. The structure of a key is shown in Fig. 4. To ensure key uniqueness the instance (T3) is an integer indicating the number of nodes of this type that have previously been stored in the database. The body (T2) is a concatenation of T2 and T3 from the parent key. This structure provides efficient traversal of the database. Given a node key, it is possible to locate a child of this node simply by replacing T1 with the alias of the child. A partial key search will return the first child of the type requested or indicate that no such child exists. Locating the parent of a node is similar. Replacing T1 with the alias of the parent and truncating T3 produces the key of the parent node.

3.2 Subschemas

Subschemas in T are used to invert database structure, shield data from users, and provide efficiency in retrieval. An alternate view or

subschema is an abstract model of a portion of the conceptual database or master schema. In addition to promoting logical data independence, a subschema may also provide a convenient data protection facility.[5] For example, there are situations in which the owner of a database may wish to create a subschema allowing other users access to part of the database but shielding some nodes from public access. In another case, the relationships among nodes may be different in a subschema from what they are in the master schema. For example, a market manager may wish to view the database in Fig. 3 as

```
market-segment
  product-line
    product
```

whereas a product manager may wish to view the database as

```
product-line
  product
    market-segment
```

When utilizing these subschemas neither the market manager nor the product manager has access to sales representative and customer information, since they are not contained in their conceptual views. These subschemas provide different users with their own conceptual view of the database, regardless of how the data have been stored. While other data management systems provide access to subschemas, they typically construct secondary indices by processing the entire database. Because of this processing, the database administrator, not the user, typically generates the subschema. This method also involves overhead and maintenance problems as new data get added to the database. In the approach taken by T, since the schema is separate from the actual database, and does not involve secondary indices, generating a new subschema becomes a simple mapping of one structure to another. Since this mapping does not involve accessing the actual data, no overhead is incurred, and adding new data has no effect on a subschema. In addition, generating a new subschema becomes a process available to users.

Defining a subschema for a T database is easy and does not require the assistance of a database administrator. The user simply creates a schema file that defines the new structure. A subschema may be installed when a query session is initiated or during a query session via a command provided by the query language. When subschemas are defined, there is no need to identify the fields for each node; all properties of a node are carried forward to the subschema. Some restrictions are placed on subschemas. One is that the list of nodes contained in a subschema must be a subset of those defined in the master schema. The relationships between nodes may change, but no

new node types may be defined by a subschema. Another restriction is that users employing a subschema have read-only access to the data. Without these restrictions, the original contents of the database could be corrupted. In addition, there are some query commands that are locked when using a subschema. For example, the subschema defined for a product manager is an inversion of the master schema that may result in a many-to-one relationship between products and market segments. Thus, a request to fetch the next market segment within the parent (product) may be ambiguous, since the current market segment may have several parents. In general, T attempts to make available to subschema users only those commands that have unambiguous interpretations when the subschema is mapped to the underlying database. Users of subschemas in T incur no performance penalty. Rather, in some circumstances, using subschemas can simplify queries and provide more efficient access to information. Figure 5 shows the relationship between schemas and T.

While T does not allow new nodes to be defined in a subschema, it is possible to define new nodes in the master schema at any time without restructuring the database. These new nodes may be inserted at any level of the hierarchy beneath the root (level 1). For instance, to add service-center as a child of product-line in the database shown in Fig. 5, one need only insert a description of service-center in the master schema, as shown below:

product-line (*name*;)
  service-center (*name*; *address*; *state*; *telno*;)
  product (*name*; *price*;)

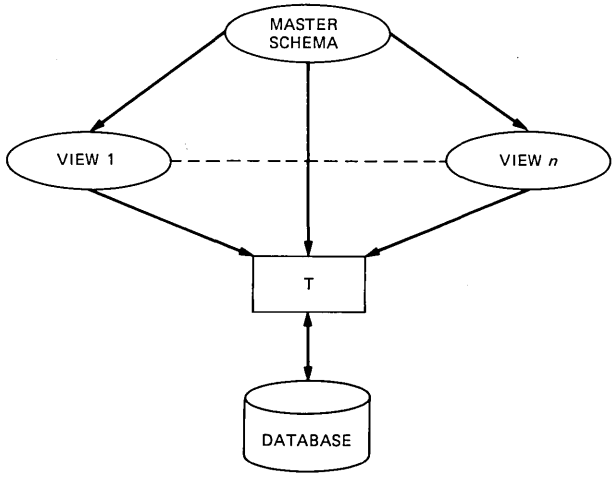

Fig. 5—The relationship between schemas and T.

3.3 Derived data

A unique and powerful feature provided by T is access to data that is not physically stored within the database itself. Data of this type are called virtual or derived data.[6] Three variants of derived data are supported by T: type 0, type 1, and type 2. Type 0 specifies that the data to be accessed either already exist in a file or will be the result of the execution of some process. In either case, the entire data set is treated as one logical record. For example, document abstracts can be stored in *UNIX* system files and accessed as part of a T database. Each abstract is treated as a single record during processing. In contrast, type 1 specifies that each line in the external file, or each line returned by some process, is to be treated as an individual instance of its node type. For instance, data of this type might be generated periodically by some external process, and rather than updating the database, it might be stored in a standard *UNIX* system file that is accessed whenever this type of node is referenced. Data of type 2 are always an executable process. Here, values are generated that do not exist in the database. A process or chain of processes to be executed occurs at retrieval time and, while the actual results do not exist, the ability to generate them does. For example, a report can be generated using data extracted with a prior query request, or the results of a query request may be displayed graphically rather than as lines of text. Using this methodology, data analysis may become a natural extension of the retrieval process. For instance, if a set of mathematical models and other analytical functions are described as data elements contained in a database, access to these functions becomes a simple retieval command, thus allowing analysis and modeling to be done without leaving the current query session. This can help to create a simple, yet extensible environment for the analyst.

Whether or not a node is actually derived is determined at run time. Nodes that could potentially be derived are identified in the schema by prefixing the node name **n*, where *n* is either 0, 1, or 2. The entry

*1 product (*name*; *price*;)

indicates that product information may be derived, and, if it is, each line is to be treated as a separate product record instance. Identifying a node as being derived in the schema file does not mean that each instance of that node actually in the database must be derived. During data entry the user specifies whether a particular instance of a node will be derived.

An interesting side effect of derived data is that the data item associated with a derived node is the name of the file to be accessed on retrieval. Since T provides interactive update for data items, a user can use this facility to dynamically change the source from which data

are retrieved. A user could, for instance, switch among several analytic models simply by interactively changing the name of the file to be executed. Since all users of a database retain a consistent view of the database during a query session, this dynamic switching does not affect other concurrent users of the database.

Accessing derived data with T is identical to accessing data actually stored in the database. There are no semantic differences or subtleties to contend with. The primary command to retrieve data with T is the `find` command. In its simplest form the command

`find` *customer*

retrieves the first customer node and displays the contents. The command

`find all` *customer*

locates all customer nodes. Instead of immediately displaying the results, T responds with the number of records retrieved. The user can then decide to display all the data retrieved, a portion of each record, or ignore the results entirely. To display output T provides the commands `print` and `fprint`. The command

`print` [*attribute list*]

where the optional attribute list contains the names of individual fields separated by spaces, will display the requested results at the user's terminal. If no attribute list is provided, the entire record is displayed. The command

`fprint` [*attribute list*] > *file*

redirects the results to *file* rather than displaying them at the user's terminal. The `fprint` command provides a data extraction capability from T databases.

Assume that we had included a node, c-report, as a part of our master schema, with the definition

*2 c-report

This node will access a report generator and produce the requested report. The set of commands

```
find all customer where sales gt 1000000
fprint name sales > foo
find c-report
```

would retrieve all customer records having sales in excess of 1000000, place the customer names and sales figures in file foo, and generate the customer report. It is assumed that the report generator being used here expects its input to be in a file named foo.

3.4 Query logging

At times the set of commands that comprise a query session need to be resubmitted periodically. This could, for example, be done to generate periodic reports using data extracted from a database. If the exact set of commands to be executed is known ahead of time, they can be entered in a file that is passed as input to the query processor. There are times, however, when the exact syntax of the commands or the proper sequence in which they should be executed is not known ahead of time. In other cases a sequence of queries pertaining to one set of data could, with slight modification, be used to retrieve a different set of data. For example, to modify the report in the example given above to select only customers with sales up to 1000000, the operator `gt` can be changed to `le` and the sequence of commands resubmitted.

During a query session, T keeps a log of those user commands that do not modify data or previous queries. The commands `retype`, `copy`, `remove`, `edit`, and `redo` provide the user the ability to manipulate previous queries and resubmit an individual query or a group of queries with a single entry. At the end of a query session, the user is given the opportunity to save a copy of this query log. Query sessions that have been saved need not be entered manually each time they are used.

IV. PERFORMANCE

Real-time response to T queries is acceptable. In timing experiments run on an AT&T 3B20S computer, under normal load—running System V, Version 2.0.2—queries involving key retrieval and pattern matching searches executed against a file containing 1.1 megabytes (10,654 records) in less than 10 seconds real time. These queries were constructed to ensure that the entire database was searched and only the last record in the file satisfied all constraints. The largest database known to have been accessed by T contained about 40 megabytes. More typical applications vary from 1 to 20 megabytes. It is extremely difficult to obtain meaningful performance figures for an interactive data management system. This is especially true of the hierarchical model. Much of the performance depends on the actual structure of the data and the type of information requested. Actual performance is also affected by the load distribution of the system at the time operations are initiated. While response time is an important factor in performance evaluation, consideration should also be given to whether the system makes efficient use of user's time. In providing unique features such as access to derived data, dynamic modification of views of data, and query logging, it is felt that T does help users in this area.

V. CONCLUSIONS

T was developed to experiment with concepts that would make data access, retrieval, and manipulation easier for certain types of database users. These concepts include (1) the ability to interactively modify user views of data; (2) the ability to access data and programs stored outside of the database, which provides a more natural interface to existing information and computational functions; and (3) the ability to access, rearrange, and modify previous query commands. Several of these have proven useful in a variety of analytical studies. The ease with which alternate views of a database can be constructed and the fact that they may be invoked dynamically during a query session have proven to be valuable assets to data analysts using the system. The concept of derived data has reduced the amount of redundant data stored on disk and provided greater flexibility by permitting access to existing computational and graphical functions. It makes access to a variety of heterogeneous capabilities natural within the same query language. Alternate input and output facilities provide simple mechanisms to access standard queries and provide a data extraction capability. Query logging provides a simple and flexible method of generating standard query procedures, as well as providing language extensibility and query reuse.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall, New Jersey, 1978.
2. P. J. Weinberger, private communication.
3. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Reading, Mass.: Addison Wesley, 1968.
4. E. R. Gansner et al., "Semantics and Correctness of Query Language Translation," Proc. 9th Principles of Programming Languages, Albuquerque, N.M., January 20, 1982.
5. J. D. Ullman, *Principles of Database Systems*, Potomac, Maryland: Computer Science Press, 1980.
6. G. Wiederhold, *Database Design*, New York, N.Y.: McGraw-Hill Computer Science Series, 1977.

AUTHOR

Raymond J. Yanofchick, B.A. (Economics), 1979, Rutgers University; M.S. (Computer Science), 1983, Stevens Institute of Technology; Bellcomm Inc., 1966–1972; AT&T Bell Laboratories, 1972—. Mr. Yanofchick has been involved with applied research into minicomputer-based data management systems. He currently works in the Marketing Analysis Systems Department, exploring software tools and environments appropriate for market analysis. Member, ACM, AAAI, IEEE.

AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

Design of the S System for Data Analysis*

By R. A. BECKER and J. M. CHAMBERS†

S is a language and system for interactive data analysis and graphics. It emphasizes interactive analysis and graphics, ease of use, flexibility, and extensibility. While sharing many characteristics with other statistical systems, S differs significantly in its design goals, its implementation, and the way it is used. This paper presents some of the design concepts and implementation techniques in S and relates these general ideas in computing to the specific design goals for S and to other statistical systems.

1. BACKGROUND

S is a language and system for the interactive analysis of data, developed at AT&T Bell Laboratories, and currently in use on the UNIX‡ operating system. An extensive user's guide, *S: An Interactive Environment for Data Analysis and Graphics* [7] is available. As of April 1983, about 250 sites had obtained S and over 4,500 copies of the previous user's manual had been distributed. S is being used at universities, research laboratories, and other organizations. While sharing many characteristics with other statistical systems, S differs significantly in its design goals, its implementation, and the way it is used.

The design goal for S is, most broadly stated, *to enable and encourage good data analysis*, that is, to provide users with specific facilities and a general environment that helps them quickly and conveniently look at many displays, summaries, and models for their data, and to follow the kind of iterative, exploratory path that most often leads to a thorough analysis. The system is designed for interactive use with simple but general expressions for the user to type, and immediate, informative feedback from the system including graphic output on any of a variety of graphical devices. In addition, the system is open to change: Even though the current system has many capabilities, a

† Authors are employees of AT&T Bell Laboratories.

‡ UNIX is a trademark of AT&T Bell Laboratories.

Table I—Design features of S

Topic		See Section
Syntax	Expression language	4
	Uniform treatment of function arguments/results	4
	Formal grammar	6
	Precedence	6
Data Structures	Self-describing attribute-value pairs	5
	Hierarchical	5
	Vector structures	5
Implementation	Structured code	9
	Software tools	9
Portability	FORTRAN (as portable assembly language)	9
	Isolation of machine dependencies	9
	Uniform executive/function interface	6
Extensibility	Macroprocessor	7, 9
	User-written functions (interface language)	7
Other	Device-independent graphics	8
	Online documentation	7

variety of ways are available to extend the system as new applications and techniques appear.

The implementation of S draws on a number of modern computing principles and techniques. Table I summarizes some of these. Many, of course, are popular concepts, although few statistical systems apply them together consistently. Some, such as hierarchical data structures, seem to be unique to S among statistical systems. Vector structures and our approach to an interface language are also novel.

Work on S began at Bell Laboratories in 1976; an initial implementation on a large Honeywell mainframe system was in use late that year. Starting in 1978, a version of S was developed for the *UNIX* operating system. Since 1981, this version has been distributed outside Bell Laboratories. S represents both an evolution from earlier statistical computing work at Bell Laboratories, particularly program libraries and graphics software (see [10]), and also our opinions about what was good and bad in the software used for data analysis at the time. (For a more complete description of how S is used in actual data analysis, see [7].)

2. S AND OTHER SYSTEMS

When the design of S began, a group of us at Bell Laboratories considered the then existing statistical software in terms of our goal

of good data analysis, particularly in an interactive, exploratory environment. There were three main approaches to doing statistics on the computer: programming in a conventional language, usually FORTRAN (this had been our own previous approach); mainframe statistical packages such as BMD, SAS, and SPSS; and a few interactive languages, notably APL. We recognized the need for better use of human resources than having to write FORTRAN programs, but found problems with the existing alternatives.

Statistical packages arose in the 1960s and were closely modeled on the idea of sequentially processing a series of records on punched cards or magnetic tape. Relatively recent user guides to BMDP [8] and SAS [16] still picture the user input as a card deck. This model has several bad influences. Good data analysis is highly iterative, responding to important facts observed in the analysis itself. Picturing analysis as processing a sequence of records through a limited set of statistical commands discourages this freewheeling interaction with the data. In particular, interactive use of the statistical packages was either not available or consisted largely of the ability to set up the card deck and run it from a terminal. S, on the other hand, was designed with the model of a language operating on complete data sets, interactively, in a nonsequential manner. A number of modern statistical techniques, e.g., robust estimation, cannot easily be expressed in the sequential form, and are therefore hard to incorporate in some of the packages.

Another result of the batch approach was the tendency to "shotgun" output, printing all the summaries likely ever to be relevant from a particular model or process. Instead, S tries to provide a wide variety of displays, particularly graphical, that can be used interactively to see the summaries that are relevant to the particular user. Graphics, like interaction, was not part of the original design of the mainframe packages. Since 1976, many of them have added graphical facilities; however, the graphics tend to be viewed as "reports," rather than being integrated into the analysis. For example, most of the graphics add-ons do not include graphic *input* which in our opinion is essential for identifying important features observed in the plots.

The APL language, while not designed for statistical computing, offered a very different, and in many ways, more attractive approach. It was intended for interactive use, with users typing expressions that operate on whole data sets and produce immediate output at the terminal. Users can extend the language by defining interpreted "functions" that can then be used in the same way as primitive APL operators. These are all features that contribute to APL's usefulness for data analysis, and which we have incorporated into S. The consistency and functionality of APL's operators is also present in S; however, in S, such operations are normally carried out by functions

rather than operators. The main problems with APL are its syntax, its data structures, and its isolation from other languages. APL has only operators, i.e., functions with one or two arguments, and its precedence rules are different from those of ordinary algebra. For statistical applications, the latter is inconvenient for many users, and the former is a serious drawback. Statistical functions usually have a few main arguments (the data to work on) and any number of additional optional parameters or auxiliary data. They are generally awkward to express as unary or binary operators, as noted in Section 4. In S, we responded by allowing general function calls and by using common algebraic notation for expressions.

The APL data structure is the multiway array, while the result of most statistical functions tends to be less regular. A regression, for example, needs to be described by coefficients, residuals, and summaries of the numerical and statistical methods applied. Fitting this into a single multiway array is unnatural. Allowing completely general, hierarchical data structures in S let the results be expressed naturally, while allowing any data structure to be the value returned by a function hid the structure from users who had no need to extract the pieces explicitly.

The interface to user-written primitive functions discussed in Section 7 allows new functions to be defined when a purely interpretive form would be difficult to write or very inefficient. Both APL and the mainframe statistical packages made the process of interfacing to, say, a new FORTRAN-based algorithm either severely constrained, (e.g., only one user-defined extension) or complicated (involving the implementation details of function interfaces). The substantial number of high-quality algorithms published by journals, such as *Transactions on Mathematical Software* and *Applied Statistics*, makes them an important source of extensions to statistical systems.

Changes in packages and languages since the development of S have often reflected similar concerns to those we felt. Many packages have added graphics and interactive modes. A recent new version of APL moves toward more general data structures. A system built on APL, STATGRAPHICS [24] adds graphics and hides the syntax behind menu-driven interfaces. These are beneficial changes for the users of such systems; however, designing interaction, graphics, and generality in from the beginning makes for a cleaner result.

In retrospect, it is clear that the evolution of S, in many respects, parallels a number of other contemporary computing activities. Our emphasis on user-extensible data structures and operations, and on removing details of data management and implementation from the user is similar to Smalltalk [18]. The approach in S to data structures, dynamic determination of their properties, and a blending of data and

"program" (in macros) has some of the flavor of many LISP-based systems. Speakeasy [17] has some of the S flavor of building an interactive user interface to make mathematical and statistical computations user-friendly, although it is more restrictive in terms of data structures and extensibility.

S represents a growing approach to computing that emphasizes the effectiveness of the *human* as the most important design criterion, as shown by the emphasis on friendly interactive access to computing, on information hiding, and on greater flexibility through delayed binding. Our philosophy is that the effectiveness of the *human* is the most important criterion for design of a computer system.

3. OVERALL ORGANIZATION

An S user types *expressions* that describe the analysis to be done. Some examples are in Table II.

The expressions involve a wide variety of *operators* and *functions* which carry out arithmetic and mathematical operations, statistical analyses, graphics, data manipulation, and other computations. Expressions also use and create *data sets* containing data structures, e.g., vectors, arrays, time series, tables. Data sets are automatically accessed by name. The S *executive* interactively parses expressions and controls their evaluation.

The organization of S resembles that of an interactive operating system: The executive corresponds to a command interpreter, the data sets to files, and the functions to the individual commands. The specific similarity to the UNIX system organization [25] is probably not coincidental, although it was not conscious. There are significant differences, however. The expressions for data analysis need a richer syntax than the commands in an operating system, particularly for algebraic expressions, and data for arguments and results have more structure (for example, commands in the UNIX system operate largely on unstructured streams of bytes).

Table II—Some S expressions

```
  # read a vector of numbers from a file, create data set mydata
mydata ← read("my.data.file")
mydata — mean(mydata)                    # subtract the mean from each value
  # Given a matrix of predictor variables longley.x
  # and a response variable longley.y
  # get the residuals from a multiple linear regression model
r ← regress(longley.x, longley.y)$resid
  # compute the residuals
  # larger than the median absolute residual
r [abs(r) > median(abs(r))]
```

S was designed in a research environment with statisticians who continually develop new techniques, so it was essential that the system be extensible. Some of this extension (macros and new data structures) can be done within the interpretive S language itself. Other extensions involve the creation of new S functions. S includes an *algorithm language* for writing computational algorithms, an *interface language* for describing the interface between the algorithms and the interactive user, and utilities to create new functions. All of these facilities for extension are intended for *users*; they are not restricted to those familiar with the internal workings of S.

4. EXPRESSIONS: THE STATISTICAL LANGUAGE

The user who types expressions into an applications system wants a combination of simplicity and flexibility. Simple requests should be straightforward and brief. At the same time, unusual but sensible requests should not be impossible or unreasonably complicated. Novice and expert users will place different emphasis on the simple or on the unusual.

In S, all user commands follow one general syntax: *Everything is an expression.* The expressions that are given to S may be as short or long as is comfortable for the user.

Expressions in S use functional and algebraic syntax as shown in Table II. (The formal syntax rules are given in Table III.) For users with some background in mathematics, science or engineering, this syntax is readable and familiar. Extensions to ordinary algebraic notation introduce a few special operators, for example, a colon is a sequence operator so that **x:y** is a vector going in steps of ± 1 from **x** to **y**.

When an expression is given to S, it is evaluated. The result may be assigned a name and thus saved as a data set. If the result of an expression is not assigned or used inside another expression, it is printed for the user.

Algebraic notation, i.e., prefix or infix operators, is natural for functions with one or two arguments. However, data analysis quickly becomes involved with functions having many arguments. Functions in S can have arbitrarily many arguments which can be specified positionally or by name. Typical functions to carry out statistical or graphical analysis will have a few arguments to say what data is to be analyzed or plotted as well as many optional arguments to control details. Options are most easily supplied in the form *name=value*; the options of interest can be specified in any order. Functions return data structures that may have arbitrarily many named components; thus, functions may have any number of inputs and produce any number of outputs.

Table III—S grammar rules*

```
expr      : NAME ( arg.list )                          #function call
          | expr OP expr                               #binary operator
          | UNARY expr                                 #unary operator
          | sub.expr [ arg.list ]                      #subset
          | asn.expr ← expr                            #assign or replace
          | expr → asn.expr                            #same, to the right
          | INT                                        #literals
          | REAL
          | STRING
          | sub. expr                                  #can be subsetted
          | asn.expr                                   #can receive assignments
          | control                                    #iteration, conditional, etc.
          ;
asn.expr  : com.name                                   #name or component
          | com.name [ arg.list ]                      #subsetted
          ;
com.name  : NAME
          | com.name $ NAME
          | com.name $ [ expr ]
          ;
sub.expr  : ( expr )                                   #anything in parens
          | NAME ( arg.list )                          #function call
          | sub.expr $ NAME                            #component
          | sub.expr $ [ expr ]                        #numbered component
control   : if ( expr ) expr
          | if ( expr ) expr else expr
          | for ( NAME in expr ) expr                  #iteration
          | while ( expr ) expr
          | repeat expr
          | break                                      #loop control
          | next
          | { exp.list }                               #compound
exp.list  : expr
          | exp.list ; expr
          ;
arg.list  : arg
          | arg.list , arg
          ;
arg       : #empty                                     #empty
          | expr
          | NAME =
          | NAME = expr
          ;
```

* Lexical tokens are capitalized; key words are in boldface.

One of the most powerful functions in the S language is represented by the subscripting operator. Since S deals with vectors, it is natural that subscripts are also vectors. Thus

```
x[ 1:5 ]
```

returns the first five values in **x**. Since it is frequently necessary to exclude data from statistical analysis, negative subscripts specify the values to be excluded:

```
x[ -6 ]
```

returns **x** with the sixth value omitted.

Subscripting can also be used to answer database-like queries. Logical expressions used as subscripts cause the selection of data corresponding to TRUE values in the subscript.

```
name[ salary > 30000 & age < 25 ]
```

The operation extends naturally to multiway arrays, and in this context, an empty subscript denotes all values in that subscript position. For a matrix **y**

```
y[ , 6:2 ]
```

returns all rows of columns six through two. As this example illustrates, the subscript operator can also permute data values (here reordering columns 6 through 2).

The function **order** generates subscripts corresponding to a sorted version of its argument. Thus

```
x[ order(x) ]
```

is equivalent to

```
sort(x)
```

Using **order**, it is possible to do passive sorting simply:

```
name[ order(salary) ]
```

lists names in increasing order of salaries.

The **print** function, implicitly invoked whenever a result is not assigned, represents numerical results to the appropriate number of decimal places and can neatly lay out matrices, time series, multiway tables, and character data.

The function **apply** (similar to "mapfun" in Lisp [23]) is able to invoke another function repeatedly on portions of data structures. In its simplest form, **apply** invokes a function on each of the rows or columns of a matrix. Thus

```
apply( y, 1, "mean" )
```

invokes **mean** once on each row (Dimension 1) of the matrix **y** and returns the vector of row means. With other choices for its second argument, **apply** can deal with slices of multiway arrays. Functions can also be applied over hierarchical data structures and ragged arrays.

5. DATA STRUCTURES AND DATA MANAGEMENT

Data sets in S contain self-describing, hierarchical (list-like) data structures. Data sets are created automatically by assignment expressions; no user control of storage is required. The elementary data structures are *vectors* of numbers, logical values or character strings:

```
> response
  1.01  0.97  3.1  7.21
> response > 2.5
  F  F  T  T
> species.name
  "Setosa"  "Virginica"  "Versicolor"
```

(Here the ">" is the S prompt for an expression.)

The numeric *data modes* are "real" and "integer," but for the most part, the distinction is unimportant to the user. In S, the value of the expression "3/2" is 1.5, even though in many programming languages, integer arithmetic would produce an integer result of 1. A special operator is provided for integer division when it is needed.

There is a special value, NA (not available), which can be used to signify missing data. Any arithmetic operations on NAs produce NAs.

General data structures consist of any number of components, each component being either a vector or another general data structure. Each component has a *component name*; syntactically, the component named **Label** of a structure **z** is denoted **z$Label**.

We designed S so that most users are unaware of the details of data structures, but also so that structures can be defined and manipulated easily to handle new analyses. Simplicity for the user is obtained by having all functions that deal with a given type of data structure (e.g., matrices, time series or tree structures from clustering) recognize the structure type by looking for components with certain specific names. Functions that produce such structures as their value simply return structures with the appropriately named components. For example, a multiway array is defined as a structure with two vector components: one named **Data** containing the data values for the array (listed column by column), and one named **Dim** containing the extents of the array on each dimension. A 2 by 3 matrix, **x**, with data value $2i + j$ in the $[i, j]$ position corresponds to the following list representation:

```
( "x" STR
   ( "Dim" INT 2 3 )
   ( "Data" REAL 3 5 4 6 5 7 )
)
```

Certain functions make use of a list representation of S data structures to enable structures, or entire databases, to be written to files in character form and subsequently read back in.

The ordinary user does not see this structure, however; **x** just appears to be a matrix. When a matrix or array is printed, it is laid out conventionally with no explicit reference to the components of the structure:

```
>x
Array:
2 by 3
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    5    6    7
```

Matrices and arrays are created and manipulated by a large number of S functions. Data structures such as arrays or time series are so widely recognized that they are considered to be built into the language. In particular, they can be declared as special structures in the interface language (see Section 7) that defines S functions. Most of the basic functions, such as arithmetic, logic, printing, and plotting include some special facilities for treating these structures sensibly. For example, the result of adding together two time series is a time series on the intersection of the two time domains.

A broader special class consists of *vector structures*: data structures that *can* act like vectors, but have special structure in addition. Such structures can be used in arithmetic, and in general, can act as a vector argument to any S function. Arrays and time series are examples of vector structures, but the class is open-ended. Internally, any structure with a vector component named **Data** is considered a vector structure. The **Data** component is the part that acts like a vector when necessary. Functions that operate element-by-element on a vector structure change the data values but leave the other components unaltered. If **x** is the matrix above, **sin(x)** produces a 2 by 3 matrix with data **sin(3)**, etc., and **x<4** is a matrix of logical values. Functions that rearrange the order of elements, on the other hand, throw away the structure and leave just the data: **sort(x)** sorts the data values in the matrix but its result is a simple vector. Since the original design of S, vector structures have been added to represent such structures as distance measures, categorical variables, and multiway tables. These structures can be used as vectors throughout the language with no modification of the various S functions involved.

Other structures may not be interpretable as a vector, but may still be recognized by groups of functions. For example, a tree structure is used to represent the result of hierarchical cluster analysis [22]. Its components are a matrix describing the merging that took place in the analysis, a vector of the distances at which merges took place, and a vector giving the reordering of the original objects needed to plot the tree. S functions exist to produce such trees, plot them, and extract subtrees or nonhierarchical clusterings. Users of these functions need not know how the tree is represented, so long as the various functions agree among themselves.

6. THE EXECUTIVE

The S executive performs tasks roughly comparable to an operating-system command interpreter. It controls most interactions with the user, parses user expressions, schedules the execution of various functions, and handles interrupts and error recovery. The expressions and the data structures in S are considerably more general than the commands and files handled by most operating systems, so that parsing and data transmission in the S executive must be correspondingly more general.

User expressions are accepted by a parser built using a version of the YACC compiler-compiler [19] with a customized lexical analyzer. The use of YACC allows the syntactic rules in the language to be compact and relatively readable; with just a little cleaning up, they are included in Table III. Occasional changes in the syntax are made easier to implement as well.

The result of the parse is a hierarchical structure representing the parse tree; in fact, it is another of the general S data structures. It is evaluated by performing a depth-first traversal of the tree. When parsed, each function invocation becomes a structure with one component for each argument: As the functions are evaluated, the *value* returned by each function replaces this substructure in the parse tree. When the traversal is complete, the parse tree has been replaced by the value of the expression.

Down to function evaluation, the process is essentially portable. However, the process by which the executive invokes an S function is crucially system-dependent. S consists of a large collection of functions (currently around 300). Furthermore, users must be free to write and use their own functions. The facilities of the operating system running S determine how such a collection can be maintained and used in a reasonably efficient way. Table IV lists advantages and disadvantages of various implementation techniques that can be used to execute functions. Operating system constraints have forced us to use several different strategies. For the original version, on a Honeywell computer with a relatively primitive operating system (no virtual memory or process control), we wrote our own dynamic loader. Each S function was an overlay, read in by the executive; control was passed by a standardized transfer vector.

Running on 16-bit hardware without virtual memory, the major constraint is that program address space is limited. For this environment, we implement S functions as independent programs. Control of execution and data transmission, respectively, are handled by simple process-signaling facilities and by file input/output.

The implementation on 32-bit hardware exploits the larger address space to incorporate some, or all, of the S functions as part of the

Table IV—Advantages and disadvantages of implementation of S executive

Overlay Loading	
Common code resides in the executive.	Execution-time overlay loading is relatively slow. Many loaders do not allow partial linking—the entire system must be linked at one time. Tools are required to allow user-written overlays. Functions which get into trouble may harm the executive.
Dynamic Linking	
The operating system does most of the work.	Most operating systems do not have this facility. Functions which get into trouble may harm the executive. First-time invocation is slow because of resolving external references.
Large Virtual Memory Executive	
The operating system does most of the work.	Functions which get into trouble may harm the executive. The entire system must be linked together. Recursive calls involving the **apply** function may be needed. A facility for user-written functions is necessary.
Independent Processes	
It is easy to create functions—user and system—since they are independent. The executive is insulated from functions.	Common object code in many modules means large disk storage for executables; it is hard to correct bugs in common code. Function invocation is slow due to process creation/startup. Interprocess communication is needed.

program containing the executive. We speculate that an ideal environment may be one in which dynamic linking is combined with virtual-memory facilities, although we have no experience with such a system.

For our goals of flexibility and extensibility, it is essential that these changes in implementation affect only the executive, not the source code for the individual functions. Even in the executive, only a relatively small fraction of the code is system-dependent. However, this code has an importance to the reliability and efficiency of the system much greater than its size. Adapting the control of such a large-application software system to the features of an interactive computer system is likely to be the most important and difficult implementation step.

7. FUNCTIONS: ALGORITHMS AND INTERFACES

There are basically two ways to extend the S language to perform

Table V—Comparison of macros and functions

Macros
Easy to write
Little programming skill needed
Uses existing S functions—a limitation
Slow execution
Functions
Programming needed
Harder to create than macros
Creation process is slower
Unlimited flexibility
Fast execution

new computations: macros and functions. A comparison of macros and functions is included in Table V.

Macros combine existing S expressions into a named entity. They are easy to write and are often useful for providing sets of operations for specific applications. As an example, consider a macro designed to produce a scatter plot which includes a fitted regression line:

```
MACRO line(x, y)
plot(x, y)
abline(reg(x, y))
END
```

The body of the macro, with appropriate parameter substitutions, is inserted when the user types an expression such as

```
?line(x, y)
```

Details of macro writing are given by Becker ([7], Chapter 6). The macroprocessor itself is an extension of the M4 processor [21].

Some operations are difficult or inefficient to carry out using existing S functions. Sometimes, new algorithms appear that would be useful in S. In these cases, it is desirable to write a new S function.

All S functions are defined by *interface* routines. The interface routine describes the arguments that the user may supply, how these arguments are to be interpreted, and what default actions will be taken when arguments are omitted. It checks for errors in the arguments that would prevent successful execution. It allocates space dynamically for data structures needed for the value of the function or for temporary storage. It invokes *algorithms* to do the actual computation (numerical, graphical, etc.) and returns the appropriate result. The interface routine is written in an interface language which is compiled using FORTRAN as an intermediate language. The function **gs**, for example, takes a matrix and uses a Gram–Schmidt algorithm, which could be written in the algorithm language, FORTRAN or C, to

decompose it into the product of an orthogonal matrix **q** and an upper-triangular matrix **r**. The corresponding interface routine is:

```
FUNCTION gs(x/MATRIX/)
STRUCTURE( q/LIKE(x)/,
  r/MATRIX,NCOL(x),NCOL(x)/ )
call gs(n,NROW(x),NCOL(x),q,r)
RETURN(r,q)
END
```

The first line says that the function has one argument and that this argument should be interpreted as a matrix, if possible. The next statement allocates two structures whose sizes are determined at execution time, the fourth line invokes an algorithm (subroutine) to do the calculations, and the fifth line specifies the result of the function as a structure with the two components **q** and **r**.

It is important to our design that interface routines be as simple and readable as possible. Users, not just system programmers, should write them. Most interface routines are longer than this example, since most S functions have several arguments. However, more detailed or complex interface routines are usually still readable.

S also provides a tool to create a partial documentation file from an interface routine. This helps ensure accuracy in documenting arguments and results, and also makes it easier for the author of a function to document it in a manner consistent with the standard S online documentation. (A similar tool is also available for user documentation of macros and data sets).

The interface routine is also an interface between the S data structures and the vectors and arrays that the FORTRAN-based algorithms can handle. Typically, an algorithm will require both the *data* part of a structure and some *attributes* such as the length of a vector or the number of rows and columns of a matrix. These attributes are implicit in the self-describing S data structures and are supplied to algorithms by attributes defined through the interface language.

An extensive set of computational algorithms is required to support a large collection of functions. The algorithms need to be of good quality, they should if possible *not* be restricted to use within S itself, and the human effort required for producing and maintaining them should be kept small. In particular, we take advantage of published or otherwise generally available algorithms. Our algorithms are generally written in or converted to our *algorithm language.* This language is a combination of RATFOR [21] or EFL [15] with a set of macros to provide machine constants, error handling, user messages, dynamic storage, and various specialized facilities. Algorithms can be converted mechanically from FORTRAN into RATFOR [1], but additional effort

is usually required to make error handling, use of scratch space, and other environmental interfacing compatible. Ideally, this does not require knowledge of the details of the algorithm.

In addition to imported algorithms, S contains a large collection of graphics routines, facilities to support reading and printing of data, and algorithms to handle the S data structures.

Our experience shows that a large body of algorithms to support a scientific system can be developed by a small programming group if: (1) advantage can be taken of the many existing algorithms: and (2) there is a good collection of tools to make error handling, dynamic storage, and other support features easy to implement.

8. GRAPHICS

Data analysts use plots iteratively as an intimate part of their study of data. The unique role of plots comes from their information content: No other form of output conveys so much information so quickly. Users often react to plots by finding the unexpected and using this new information to shape the subsequent analysis. A variety of graphical techniques for data analysis is presented in a recent book [11].

S emphasizes interactive graphics as one of the most important tools in data analysis. Graphics functions in S provide the simple displays that are predominant in statistical graphics, most notably the scatter plot, in a flexible and easy-to-use form. For example:

```
plot(x,y)          # scatter plot
qqnormal(x)        # Normal probability plot
```

The general data structures and expressions in S help to provide graphical output from a variety of sources. Many statistical analyses produce results that define a scatter plot; for example, a *probability plot* [11] shows an ordered set of data plotted against corresponding quantiles of a probability distribution. Deviations from a straight-line pattern help assess distributional assumptions. Rather than duplicating scatter-plot software for each such plot, S functions return as their value a *plotting data structure*, which is passed automatically to the **plot** function to be displayed. The expression

```
qqnorm(mydata)
```

produces a probability plot of **mydata** against quantiles from the standard normal distribution. Internally, **qqnorm** only generates the plotting data structure and then invokes the scatter-plot function; **qqnorm** need know nothing about plotting. The data structure consists of two vector components for the **x** and **y** coordinates of the points to plot. Once the probability plot is seen as a data structure, it

is straightforward to use this structure for further analysis, for example, by fitting some suitable line to the points in the plot.

While statistical graphics characteristically make wide use of simple plots, such as the scatter plot, there is also the need to develop and use other special displays. The interface language and underlying graphical algorithms provide support to make the writing of new graphical functions straightforward. There are high-level graphical algorithms for generating complete displays such as scatter plots, low-level algorithms to produce components of a plot (lines, points, axis labeling, etc.) and a set of macros in the algorithm language to specify and query *graphic parameters* (Table VI) controlling details of how plotting is to be done. Parameters include specifics such as the symbol to use for a scatter plot, and generalities such as the style of axis labeling. Parameters can also be controlled by the user of S, by supplying them as optional arguments to S graphics functions. For example, a scatter plot in (device-dependent) color 6 with "O" as the plotting character could be produced by

plot(x,y,col=6,pch="O")

The goal, again, is to provide the maximum flexibility to both the interactive user and the writer of new S functions, while still keeping the ordinary use of plotting simple. Graphic input is supported in S and is important when data values or areas of interest are to be identified on the plot, as a guide to further analysis.

The graphical functions are *device-independent* in that both the user-typed expression and the underlying interface routines and algorithms are written independently of specific graphic devices. Actual graphical output is produced through a *device driver* which converts the graphics output, at a relatively low level, into commands for a particular device (see Figure 1). Drivers exist for devices including ordinary printing terminals and a range of interactive plotting terminals. A driver is written by implementing routines to carry out a specified set of graphic primitives (e.g., draw a line or plot a character),

Table VI—Some graphical parameters

User (world) coordinate system
Viewport
Plot aspect ratio
Size of margin surrounding plot (used for labelling, titles, etc)
Color
Line style
Character rotation
Character size
Plotting character
String justification (left, right, center)

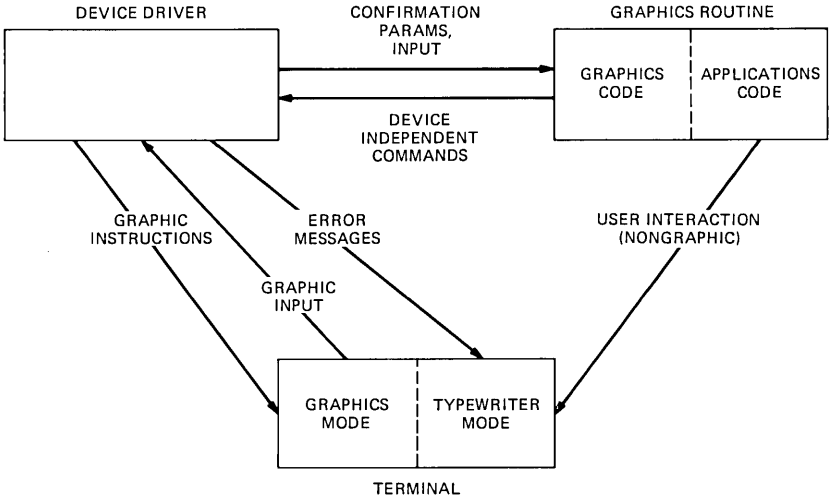

Fig. 1—Operation device-independent graphics.

and by providing a definition of the device in terms of basic graphic parameters (e.g., the device coordinate system, raster size). Incorporating a new device typically takes a few days or less; the process is sufficiently straightforward that we include instructions in the user's manual. Users can write their own device drivers.

The intimate role of graphics in interactive data analysis means that users should have interactive graphic terminals available locally. The terminals must not be too expensive and they should be of sufficient quality that the detailed information in many statistical plots can be seen clearly. Both scope terminals and pen plotters are popular with some users (mainly for rapid plotting and good graphical quality, respectively).

Our graphics routines have much in common with the CORE graphics standard [13], although our work was independent [2, 3, 5]. Relative to CORE, graphics for data analysis consists mostly of applications software; the quantity of device support software is not large. A number of the more elaborate features of the standard, on the other hand, are not often used in data analysis, e.g., retained segments. A CORE implementation would be more than sufficient to define one of our device drivers, but would not provide much of our device-independent support code.

9. TOOLS: THE OPERATING SYSTEM

The discussion so far has shown the basic design of S and the components of its implementation: execution of user's expressions; data structures and management; source code for interface routines

and algorithms. The complete system contains about 6,000 lines of interface language, 35,000 lines of algorithm language, and 9,000 lines of C code. Development and maintenance of S by a small group of people requires efficient use of time. Our experience is that three aspects of the design particularly affect human efficiency: the *languages* in which programming is done, the *tools* for maintaining the application system, and the *operating system interface.*

The approach to language was described in Section 7. Developing our own interface language and algorithm language may have taken perhaps 10 to 15 percent of the total effort, but this development has been cost-effective. If interface routines were written directly in a general language like FORTRAN, they would be much more complicated and error-prone, and all but the most sophisticated users would find it impossible to write their own S functions. During compilation, an interface routine typically expands into a much larger FORTRAN routine (an order of magnitude more lines of source code). Much of this expansion reflects inherent clumsiness in using FORTRAN to express the argument processing, dynamic storage management, and generation of results in an S function. At the same time, the use of FORTRAN as an intermediate language is important. We could not reimplement all the basic statistical algorithms previously written in FORTRAN.

The use of software tools is essential for creating and maintaining a system such as S. Compiler-compilers, macroprocessors, and more specialized tools ease the burden of system development. The interface language goes through our own simple compiler, two passes of the M4 macroprocessor, RATFOR, and FORTRAN during compilation. Obviously, we are not trying to optimize compilation time. However, this multistep process leaves us able to modify individual steps as our needs change.

Other tools are used to provide specific utilities for S developers. The *make* system for maintaining programs [14] is used to generate the S executive and the individual functions.

For tools to be useful in large applications systems, they should themselves be easily adaptable. For example, our use of *make* is highly specialized. The interface routines and the support programs, whether based on RATFOR or on the C language, all take advantage of special S facilities. We therefore replace and extend *make*'s built-in rules for compiling to include these special features. The result is a customized tool of our own (itself built from a number of tools).

The ease with which tools are put together is also a function of the operating system environment. The UNIX environment is convenient for developing a system such as S, both because of specific facilities and because the operating system tries not to be unnecessarily restric-

tive. Facilities such as pipes and a flexible command interpreter make the creation of customized tools much easier. The *absence* of complex rules about file formats and interprocess protocols, on the other hand, has meant fewer barriers to our implementation.

The dependence of the current version of S on its operating system environment involves both the internal dependencies and the use of operating system features in the tools. The dependencies on computer *hardware*, such as machine accuracy, are relatively easy to handle. The large majority of S code passes through FORTRAN during compilation. Nonportable features such as the choice of special characters and machine precision are isolated in the macroprocessing phase and kept in a single file.

The use of FORTRAN as an intermediate language and the parametrization of machine dependencies make the S source code quite portable. On the other hand, implementing and using a system like S benefits from a good general computing environment. Among current computer systems, the UNIX system is relatively well-designed, allowing us to combine and modify tools to put together a system like S. In a more restrictive system, we would be obliged to provide more of the support environment. We also observe substantial interest in porting the UNIX system to a variety of new computer systems, and when that is done, S goes along for free.

10. EXPERIENCE AND EVOLUTION

A significant fraction of the evolution of S has come from users' activities and experience. By far, the majority of our users are not professional statisticians. Instead, they are professionals in other areas with a need for data analysis, graphics or other S facilities to enhance their own work. In a number of cases, their specialized use of S has led them to develop, in effect, specialized systems for their own user community, built on S. This is usually done by creating a set of S macros to translate users' requests from the terminology of specific applications into the S expressions generating the results. Less frequently, more ambitious projects may include user-written S functions or interfaces between S and other large application systems. The relative simplicity of writing S macros means that a new system tailored to a particular user community can be written with a fraction of the programming effort required for a corresponding project using a general programming language. Also, the system development effort grows naturally (often unintentionally at first) out of direct use of S to solve the user's problems; there is no large initial investment in programming before any of the proposed uses can be tested and evaluated.

One of the more difficult tasks in user training has been convincing FORTRAN programmers that it is ordinarily not necessary to write explicit loops to operate on collections of data. Most of the nonprogrammers, however, seem to have little difficulty with the implicit iteration provided by S.

Specific user suggestions and our general recognition of the pattern of use have contributed many of the enhancements in S. An early user suggestion was the inclusion of a right-facing assignment arrow, →, for the occasion when one decides to save a result *after* typing a long expression. Our use of tools like syntax-driven parsing makes such changes easy. Interestingly, the interactive use of S in this case has direct implications for the syntax. Other enhancements in response to user needs include: a simple mechanism to edit and rerun expressions after errors; a "diary" feature to provide a history of the expressions executed during a session; tools to help users create online documentation for their macros, data sets, and new S functions; and facilities for moving large collections of S data sets among different machines in a portable way. We have also provided a mechanism for running S noninteractively for large or repetitive analyses, and a technique for creating device-independent graphics metafiles which can be plotted later on interactive or batch devices. The ability to provide such facilities with a limited expenditure of our own time derives from our modular, tool-oriented design and from the similar orientation of the UNIX environment.

11. FUTURE PLANS

Future plans for S concentrate on improving the human interface, particularly for use with the new generation of work stations. These offer substantial local computing power, high-quality graphics, and often, novel forms of interface using multiple windows and input devices such as the "mouse." These features allow design of nonprogramming interfaces to statistical systems with greater flexibility and more sophisticated user support than previously possible. We also plan to make the underlying S language itself more efficient and to simplify the user's view of writing new functions.

REFERENCES

1. Baker, B. S. An algorithm for structuring flowgraphs. *J. ACM 24,* (Jan. 1977), 98–120.
2. Becker, R. A. and Chambers, J. M. On structure and portability in graphics for data analysis. In: *Proceedings of the Ninth Interface Symposium on Computer Science and Statistics,* 1976.
3. Becker, R. A. and Chambers, J. M. GR-Z: A system of graphical subroutines for data analysis. In: *Proceedings of the Tenth Interface Symposium on Computer Science and Statistics,* National Bureau of Standards Special Publication 503, 1977, 409–415.

4. Becker, R. A. and Chambers, J. M. Design and implementation of the S system for interactive data analysis. In: *Proceedings COMPSAC 78,* IEEE 78CH1338-3C, 1978, 626–629.
5. Becker, R. A. and Chambers, J. M. Computer graphics for interactive statistics. In: *Proceedings of NCGA,* 1980.
6. Becker, R. A. and Chambers, J. M. *S: A Language and System for Data Analysis,* Bell Laboratories, Jan. 1981.
7. Becker, R. A. and Chambers, J. M. *S: An Interactive Environment for Data Analysis and Graphics,* Belmont, CA: Wadsworth, 1984.
8. *BMDP Statistical Software: 1981,* Dixon, W. J., ed., Berkeley: University of California Press, 1981.
9. Chambers, J. M. *Computational Methods for Data Analysis,* New York: John Wiley & Sons, 1977.
10. Chambers, J. M. Statistical computing: History and trends. *The American Statistician 34,* 4 (Nov. 1980), 238–243.
11. Chambers, J. M., Cleveland, W. S., Kleiner, B., and Tukey, P. A. *Graphical Methods for Data Analysis,* Belmont, CA: Wadsworth, 1983.
12. Cleveland, W. S. Robust locally weighted regression and smoothing scatterplots. *J. American Statistical Association 74,* 829–836.
13. CORE, Status report of the graphics standards planning committee. *Comput. Gr. 13,* 3 (Aug. 1979).
14. Feldman, S. I. Make—A program for maintaining computer programs. *Software—Practice and Experience 9,* (1978), 255–265.
15. Feldman, S. I. Bell Laboratories Memorandum: The programming language EFL. 1979.
16. Helwig, J. T. *SAS Introductory Guide,* Raleigh, NC: SAS Institute, Inc., 1978.
17. Hynes, G. C. *Speakeasy User's Manual,* Tech. Rpt. BDX-613-2569, Bendix Corporation, 1981.
18. Ingalls, D. H. H. Design principles behind smalltalk. *Byte 6,* 8 (Aug. 1981), 286–298.
19. Johnson, S. C. and Lesk, M. E. Language development tools. *Bell Syst. Tech. J. 57,* 6 (July–Aug. 1978), 2155–2175.
20. Kernighan, B. W. RATFOR—A preprocessor for a rational Fortran, *Software—Practice and Experience 5,* (1975), 395–406.
21. Kernighan, B. W. and Plauger, P. J. *Software Tools,* Reading, MA: Addison-Wesley, 1976.
22. Mardia, K. V., Kent, J. T., and Bibby, J. M. *Multivariate Analysis,* London: Academic Press, 1979.
23. McCarthy, J. et al., *LISP 1.5 Programmer's Manual,* Cambridge, MA: MIT Press, 1962.
24. Polhemus, N. W. Interactive statistical graphics in APL: Designing a versatile user-efficient environment for data analysis. In: *Computer Science and Statistics: Proceedings of the 14th Symposium on the Interface,* New York: Springer-Verlag, 1983, 10–19.
25. Ritchie, D. UNIX: A retrospective. *Bell Syst. Tech. J. 57,* 6 (July–Aug. 1978), 1947–1970.

CR Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*statistical computing, statistical software*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*application packages*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*device independence*; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*mathematics and statistics*
General Terms: Algorithms, Languages
Additional Key Words and Phrases: data analysis

Received 7/82; revised 9/83; accepted 11/83

Authors' Present Address: Richard A. Becker and John M. Chambers, Statistics and Data Analysis Research Department, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

AT&T TECHNICAL JOURNAL is abstracted or indexed by *Abstract Journal in Earthquake Engineering, Applied Mechanics Review, Applied Science & Technology Index, Chemical Abstracts, Computer Abstracts, Current Contents/Engineering, Technology & Applied Sciences, Current Index to Statistics, Current Papers in Electrical & Electronic Engineering, Current Papers on Computers & Control, Electronics & Communications Abstracts Journal, The Engineering Index, International Aerospace Abstracts, Journal of Current Laser Abstracts, Language and Language Behavior Abstracts, Mathematical Reviews, Science Abstracts (Series A, Physics Abstracts; Series B, Electrical and Electronic Abstracts; and Series C, Computer & Control Abstracts), Science Citation Index, Sociological Abstracts, Social Welfare, Social Planning and Social Development,* and *Solid State Abstracts Journal.* Reproductions of the Journal by years are available in microform from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106.


AT&T