INTRODUCTION

After many years of research, expert systems are beginning to have a widespread commercial impact. However, the rules for constructing enconomically viable systems are not yet clear. Many companies are discovering that while it is simple to gather a few rules from an expert and encode them into a "rule-based" system, building a working and enconomically viable program may be difficult enough to require many man-months or man-years of effort. One reason is that the writing of an expert system is often used to formalize an expert's reasoning process in order to mimic his performance. The software may also need to use complex representations and large knowledge bases which must be developed experimentally, and tailored to the domain of application.

Because these characteristics make it impossible to specify an expert system's behaviour beforehand, developers of this software often use a technique called exploratory programming, in which programs are written and tested in small pieces [Sheil 1983]. This style of programming is used in academic settings, where researchers often develop software to represent or test ideas rather than to process data. Programmers in many commercial application domains could also benefit from it, as the increasing use of Fourth Generation Languages, software for prototyping, and CASE (Computer-Aided Software Engineering) attests.

Conventional computing facilities poorly support exploratory programming. However, the interactive programming environments used by AI researchers are dedicated to this end. These environments are composed of integrated software tools which encourage incremental program development. The tools usually include interpreters, compilers, language-specific editors, debuggers, and a user interface that employs graphics and multiple windows. In a programming environment, these components are tightly linked, and presented to the user as parts of a seamless whole. For example, the compiler may call the editor to open a window and display errors in a source file, or the editor may call the compiler at the user's command. Through such interactions the tools work together to reduce overhead in software development.

However, current interactive programming environments are separated from the world of commercial programming by their use of specialized languages and hardware. They usually support languages such as LISP, not FORTRAN. They run on computers with advanced hardware support for graphics and even LISP code. Often, for efficiency, these environments consist of a kernel written in assembler, and layers of compiled LISP. As a result, they are not portable, and cannot run effectively on typical commercial computer systems consisting of a minicomputer or mainframe and many remote, "dumb" terminals.

Because commercial systems developers are reluctant to pay high per-user costs or replace existing hardware or software, interactive programming environments must fit into current hardware and software configurations to be accepted and successful. They must preserve investments in existing computer

systems while extending programmers' capabilities. These requirements are especially important for the development of cost-effective expert systems, which must run on existing hardware configurations, and incorporate current software and data.

One particular domain that could benefit from the use of expert systems and exploratory programming methods is log analysis, in which experts attempt to infer properties of subsurface rock layers from measurements taken in an oil or gas well. This paper describes an interactive programming environment developed for use in log analysis, and shows how it could be used to develop expert systems in this field. Called Interlog (INTERpreter for LOG analysis), it is designed to work within the hardware and software systems that are currently used in this domain. Because it supports LISP, as well as being able to utilize existing software and hardware for log analysis, it can serve as a tool for making this software more powerful by using expert systems and other techniques of Artificial Intelligence. While the current version of Interlog is still in the research stage, it illustrates many techniques for achieving these goals.

EXPERT SYSTEMS IN WELL LOG ANALYSIS

A well log is a record of measurements of a rock property taken at regular intervals by a tool drawn through a petroleum well bore. Rock properties typically recorded include density, resisitivity, acoustic travel time, radioactivity, and others. Data is usually gathered every fifth of a meter for the entire length of a well, which may exceed 2000 meters; a well can yield one and a half megabytes of digital data. Ofen the logs are represented pictorially, as shown in Figure 1.

From a set of logs taken at a given well, a log analyst seeks to determine the presence and recoverability of hydrocarbons in the formations, or rock layers, the well penetrates. As a means to this end, he (or she) must infer properties of the formation which cannot be measured directly. Particularly vital are the *top* and *base* depths, *lithology* (rock type, e.g., sand or shale), *composition* (fractional volume of rock components such as quartz), *porosity* (the fractional volume of pores in the rock, which may contain water or hydrocarbons), and *permeability* (the ability of fluids to flow through these pores). [Helander 1983] provides an execellent overview of the domain.

Log interpretation has quantitative and qualitative aspects. Many of the properties described above are numeric (e.g., porosity), and can be evaluated using equations which empirically relate them to logging tool responses. Others are symbolic (e.g., lithology), and their determination may entail the use of symbolic processing methods. Often, the two types of techniques are intertwined during the analysis of the data. Because well logs are now recorded in digital form, log analysts use computers extensively for quantitative interpretation [Snyder and Fleming 1985]. However, they still make many qualitative interpretations manually. Current log analysis

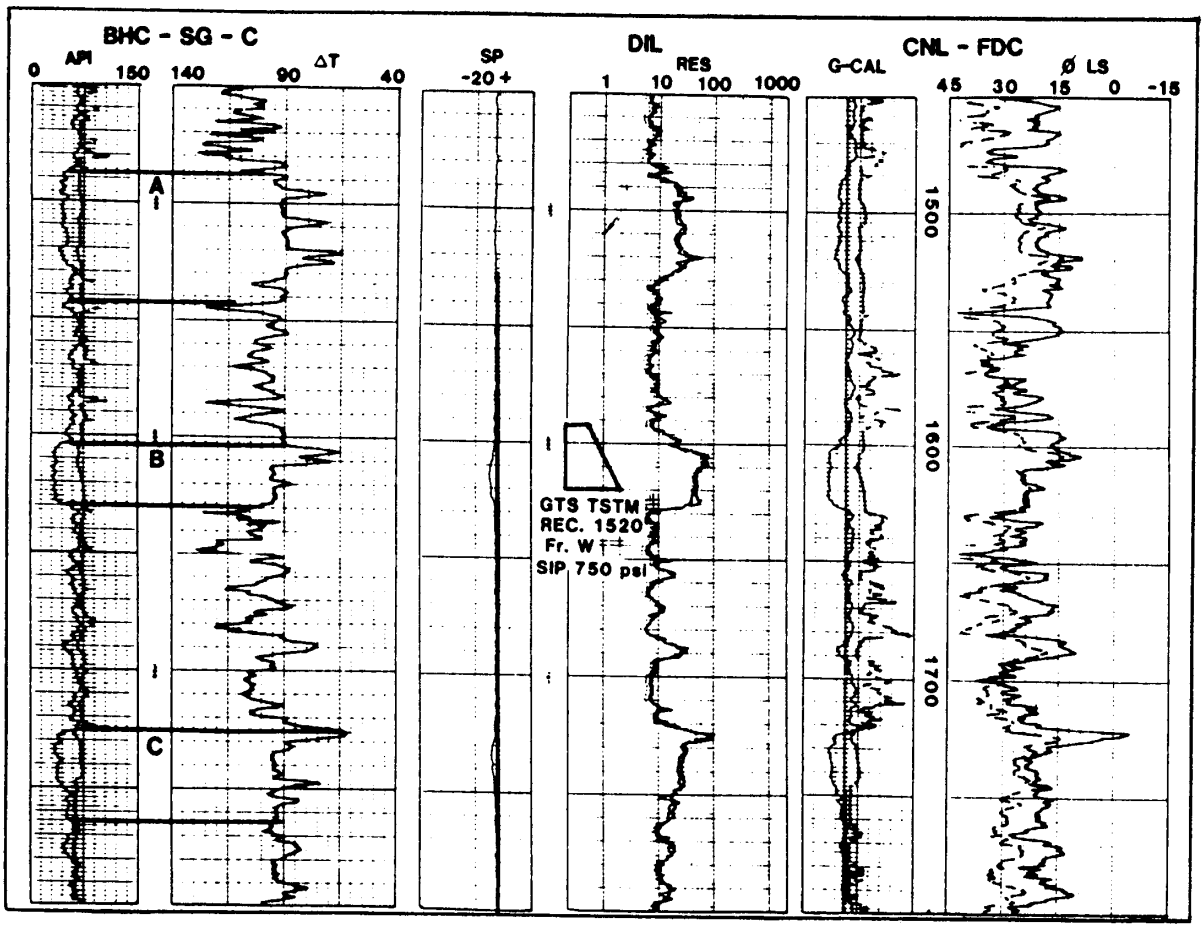

FIGURE 1: TYPICAL WELL LOGS

(after [Connolly and Reed 1983], p. 46)

software is written almost exlusively in FORTRAN, so the combining of numeric and symbolic processing is usually done in that language.

As an example of the interplay between numeric and symoblic processing in log analysis, let us consider the task of determining the lithology of well data. This step is crucial to an accurate interpretation, as the choice of equations and parameters used to analyze a formation depend on its type. Methods of determining lithology often have two parts: one to provide the lithology of the rock at each sample point, and another identify rock layers by recognizing sequences of samples with a common lithology. Such sequences, called *zones*, represent formations and are the structural units for further analysis.

While some log analysts only visually analyze the log curve shapes to delineate the zones and assign each one a lithology, others also use numerical techniques. Schlumberger, a major log analysis service company, has developed the following statistical procedure to find the lithology at each measured depth [Delfiner, Peyret, and Serra 1984]. For a given region, one collects samples of log data for different lithologies from sets of n types of logs. Because each of these samples is a point in the space spanned by the n types of logs, the samples for a particular lithology define a volume in this space. Therefore, to assign a lithology to a sample from a new set of n logs, one finds the enclosing volume, which is usually defined by representative intervals on each of the n axes. For example, a sandstone in a particular geologic region may usually have a gamma radiation level of between 0 and 75 API (American Petroleum Institute units), and an acoustic travel time reading of between 180 and 356 micro-seconds/metre.

In some cases, the sample point will fall within the intersection of two or more volumes, so the given data is not enough to establish the lithology. To provide a unique answer the log analyst has to combine his knowledge of local geology and the curve shapes with numerical results extracted from the curve values. Similar knowledge aids him in grouping the results into zones. Even this small example, in which the symbolic processing of numeric results guides subsequent numeric processing, demonstrates the interdependence of symbolic and numeric knowledge in log interpretation.

To automate the non-numeric components of log analysis, software developers are moving beyond conventional programming into the use of expert systems which combine symbolic and numeric processing. One result is LITHO [Bonnet and Dahan 1983; Bonnet, Harry, and Gascia 1982], an expert system which uses symbolic knowledge to augment the statistical procedure described above. Eliciting geologic knowledge via an interactive consultation, the expert system estimates the prior probabilities for each of the lithologies given by the numerical procedure. LITHO is but one component in a complete log analysis system, providing symbolic knowledge which other programs use to filter numerical results. However, it is not tightly intergrated with the rest of the analysis software, requiring the manual, bidirectional transfer of data between it and the rest of the system.

There are several other programs for log analysis which combine numeric

and symbolic processing. For example, ELAS (Expert Log Analysis System) is an expert systems which is highly integrated with a pre-existing log analysis package [Weiss and Kulikowski 1984]. It represents the tasks involved in log analysis using a spreadsheet, with some cells containing parameters and others naming methods of calculating them. As the user fills in the values of some cells, the spreadsheet calculates the values of other dependent cells. The expert system acts as an advisor, suggesting and executing procedures for obtaining parameters to produce a complete interpretation. Because ELAS uses an expert system shell, called EXPERT, which is written in FORTRAN, it can use rules which gather evidence by activating existing FORTRAN procedures for the numeric processing of well log data. Although this method of communicating results through the parameters of rules has its limitations, ELAS is able to provide log analysis software which combines numeric and symbolic computation. It illustrates one role for expert systems in log analysis, as controllers of numeric software and advisors to its users.

THE INTERLOG PROGRAMMING ENVIRONMENT

As the developers of the Dipmeter Advisor, an expert system for interpreting one type of well log, observe, "it is becoming increasingly important to integrate large, traditional interpretation programs ... with advisory systems that typically employ symbolic methods" [Smith *et al.* 1984, pg. 37]. Interlog is designed to ease the development of such advisory systems. However, it is also written to be familiar to current log analysts and FORTRAN programmers, thereby minimizing the effort needed to make it useful (and regularly used) in current computing environments. (Kernighan and Mashey [1981, pg. 194] label this effort the "gulp factor.") Interlog runs in a computing environment currently in use in log analysis, and can incorporate existing FORTRAN software. It also encourages programmers to reproduce user interfaces to current log analysis packages within the windowing mechanism it provides.

As Figure 2 shows, Interlog contains interpreters for F-LISP (a dialect of LISP), I-4 (a FORTRAN look-alike), and FLOP (an object-oriented language); libraries for graphics and data management; log analysis software; and a screen editor and other programming tools.

The central component of Interlog is the F-LISP interpreter, a version of LISP written in FORTRAN. Written to support the development of AI programs which incorporate conventional numeric software, F-LISP is based on LISP 1.5 [McCarthy et al. 1962]. While not yet a "full" LISP, F-LISP contains many standard LISP predicates and functions. It also has special support for numeric operations. For example, the interpreter includes "real" and "integer" scalar and array data types. It provides arithmetic operators and functions which work on entire segements of arrays. To facilitate the calling of FORTRAN subroutines from LISP, the interpreter has special types of procedure calls which allow functions to alter the values of their arguments, a practice not allowed in LISP.

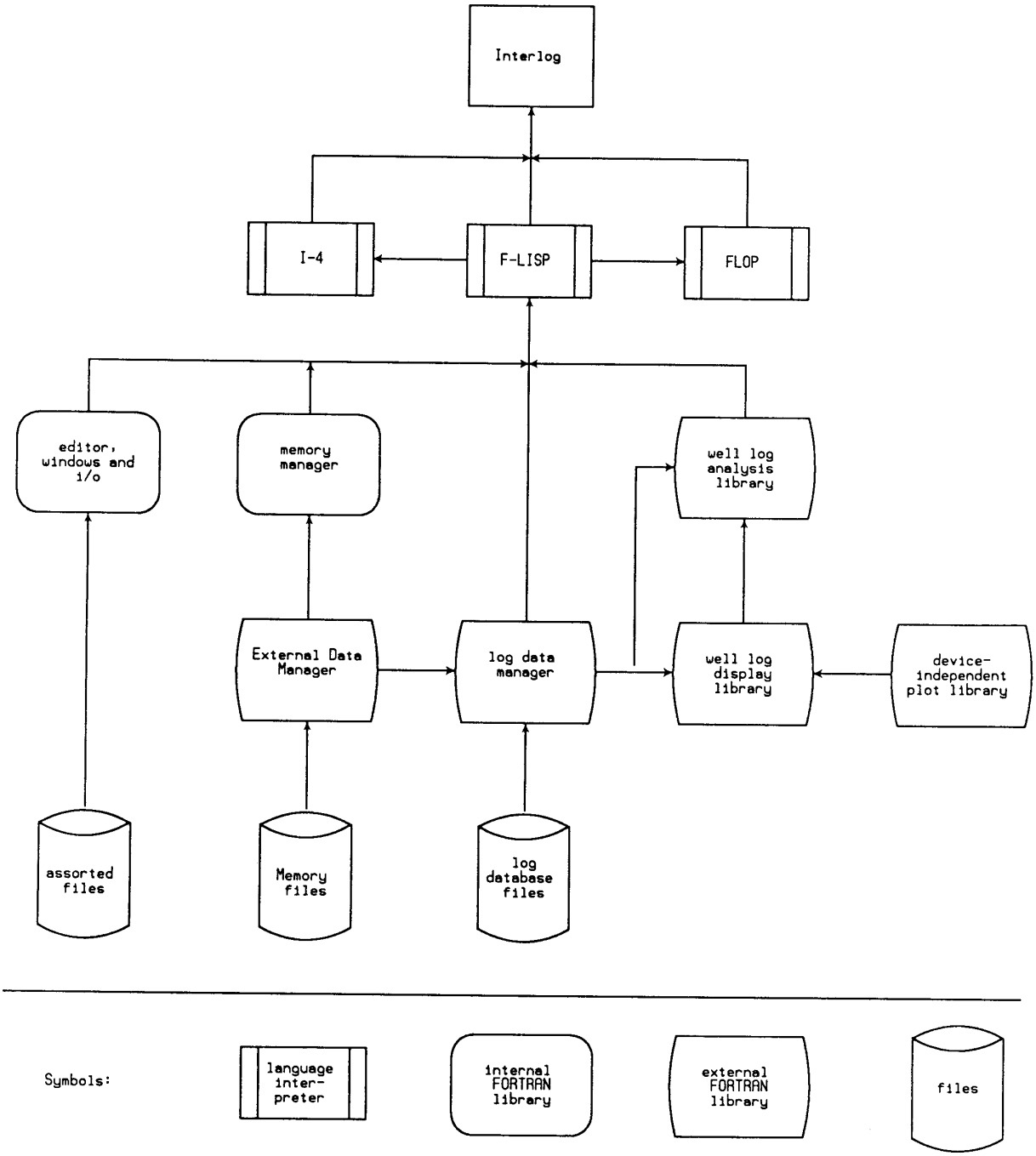

FIGURE 2: STRUCTURE OF INTERLOG

In order to support the development of conventional log analysis software, Interlog provides I-4, an interpreted subset of FORTRAN. Using this interpreter, which translates FORTRAN-like statements into F-LISP code and then executes them, programmers in log analysis could take advantage of the features of an interactive programming environment without learning a new language. Besides being useful for numeric processing in log anlaysis, I-4 will also represent the procedural components of expert systems.

The third programming language within Interlog is FLOP (F-LISP Object-oriented Programming language). As its name implies, FLOP is used for *object-oriented programming*, in which one models data items by creating *classes* of *objects*. These *objects* resemble database records in that they have fields, called *instance variables*, which contain values of specified types. However, unlike a database record, an object has associated procedures, called *methods*, which can be used for computations involving the instance variables. For example, an object *x* of class *logCurve* may have instance variables *topDepth*, *baseDepth*, and *depthInterval*, and a method *nvals* which calculates the number of data items in the curve. An FLOP program would retrieve this value using the F-LISP call *(: x nvals)*, while an I-4 program would use the call *x:nvals*. By encapsulating the representation and access procedures of an entitiy within a class definition, object-oriented languages encourage the development of modular, flexible, and robust software. For an introduction to object-oriented programming, please see [Xerox Learning Group 1981].

Object-oriented languages are capable of representing databases, but they are more useful for representing knowledge bases. Although computer scientists as yet disagree upon which characteristics, if any, distinguish these two types of data collections, we can consider three candidates. First, knowledge bases can contain executable procedures, as well as data. Second, knowledge bases can contain data which is implicit, and calculated upon demand, rather than being explicitly stored. Third, entities in a knowledge base can inherit properties from other objects. For more information on knowledge bases, see [Smoliar 1983].

As an example, consider the *nvals* property of the objects of type *logCurve* discussed above. In a database, this value will commonly be explicitly stored; the user cannot provide an arbitrarily complex procedure to calculate the value upon retrieval. On the other hand, the knowledge base definition for the class *logCurve* can contain such a procedure, for use whenever the retrieval call *(: x nvals)* is executed (*x* representing any instance of the class *logCurve*). Any instances of sub-types of *logCurve* can also use this procedure in response to an *nvals* message, if the sub-type does not provide a new definition for it. For example, if *gamma curve* is a sub-type of *logCurve* and *y* an instance of that sub-type, the call *(: y nvals)* will cause the procedure in the definition of *logCurve* to be executed, using the instance variables of *y*.

Because Interlog supports several languages which encourage different styles of programming, a developer of expert systems will be free to use the appropriate style for writing individual components of the software. As

Bobrow and Stefik [1986, p. 584] note, "many expert systems are delveloped in which object-oriented programming is used for representing the basic concepts, rules are used to specify the inferences... and procedures are used for the overall control structure."

Interlog allows the use of a fourth programming language for the writing of components of expert systems -- compiled FORTRAN. Because the Interlog programmer can write F-LISP, I-4, or FLOP code which calls FORTRAN routines directly, he or she can incorporate existing software into an expert system. Not only can these routines provide efficient numerical processing, but they also allow the AI programs to share data with other, external FORTRAN packages. Thus, Interlog's use of FORTRAN helps preserve an investment in existing software, while enhancing the use of that software through an interactive programming environment.

For example, Interlog contains several independently developed, commercial FORTRAN libraries. One consists of subroutines for device-independent vector graphics. This library is used by a number of programs at the site where Interlog was written. Another set of routines provides file storage of named data objects, both scalars and arrays. FORTRAN programs use this library to share data without the overhead of a full database mechanism. Interlog uses it to store arrays, and save the internal defintions of functions and FLOP classes. When these latter items are referenced in an Interlog session subsequent to the one they were defined in, Interlog can quickly restore them. The library thus provides a kind of permanent virtual memory.

Because Interlog was written to enhance programming in log analysis, it contains FORTRAN software specific to that domain. Taken from a stand-alone, commercial log analysis package, this software includes a log database manager, a log display manager, and a rperesentative group of log analysis routines. The log database manager provides standard access to log data stored in specially formatted files, one per well. The log display mananger uses this library to display logs on a CRT or other plotting device. Finally, the log analysis routines use both of these libraries and provide curve manipulation and other features. By tightly linking all of these routines into the programming environment, Interlog is able to provide familiar tools to its users, while at the same time making the full power of those tools available to software such as expert systems.

Interlog's use of the log database manager is an example of this integration. To provide its programmers and end users with interactive access to log data, it represents the structure of a log database with FLOP classes, and the data itself using instances of those classes. When an *open* message is sent to the class *logDatabase*, it creates FLOP objects, in memory, to represent the database and any desired curves. Curve values are stored in F-LISP vectors, which are in turn stored in the external data manager. Since there are now two copies of the data, one in memory or an F-LISP vector and the other in the database file, the FORTRAN log database routines were altered to modify only the data in memory. This data is written to the file when the database is closed.

The scheme described above gives Interlog users interactive access to the data, using FLOP statemtents, as demonstrated in the next section. At the same time, it enables other FORTRAN routines to be incorporated into Interlog to modify this same data, without any changes to the source code.

Interlog also uses FLOP to incorporate the log analysis routines of the stand-alone package into the programming environment. The commands of the latter package were rewritten as FLOP methods which apply to the given log database. For example, the *list database* command of the stand-alone package is expressed as *(: <logDatabase> list)* in FLOP or F-LISP. This representation allows the writing of expert systems which can perform analyses using the routines of the stand-alone package. The modular design of the latter software, especially in its partitioning of functions and use of the log database manager, allowed it to be integrated into Interlog without alteration of any of the log analysis code. Only the top-level command interpreter and a few of the supporting log database routines were changed.

At the moment, due to program size restrictions imposed by the current hardware, only a small number of the log analysis commands are operational. However, the techniques discussed are applicable to all.

The FORTRAN library for graphically displaying log data was brought into Interlog in a similar manner. In this case minor changing of the code was necessary to represent some necessary data objects in FLOP while providing timely updates both to and from the FORTRAN routines. To a large extent, the ease of integrating FORTRAN software tightly into Interlog depends on modularity and data storage techniques of the FORTRAN code.

The third major component of the Interlog programming environment is the set of software development tools it provides. Most of these tools are designed to work with the languages interpreters described above. The advantages of using interpreters for software development are well documented elsewhere (see, for example, [Sandewall 1978]), and will not be repeated here. However, in regard to exploratory programming and expert system development, we should note that the mechanism which interpreters can provide for recovery from run-time errors is extremely important. When such an error occurs in an Interlog procedure, the F-LISP interpreter suspends it and invokes an *error loop*, which behaves like the top-level interpreter for the current language and lets the programmer examine variables, define functions, etc. It also allows him (or her) to resume execution from the point of the error. Since this facility is available through a function call, an expert system developer could use it to alter variables and rules during execution, and thus "fine-tune" his software.

Another major programming tool that Interlog provides is a screen editor. Using this editor, a programmer can split the screen into multiple windows, editing text files in some or running programs in others. For example, a programmer may use one window to display source code while executing it in a second one. In this case, the editor automatically saves the window contents in a file, so the programmer can review his entire

session and re-enter commands easily. This facility resembles the one provided by versions of the Emacs editor for interacting with the host's operating system during text editing.

In fact, the screen editor is a major part of Interlog's user interface, as a programmer would typically use the language interpreters through a window. When a window is used at the command level, the editor will not only record the session, but echo the input lines to another file. The user can then edit this file to form new expressions for input.

EXAMPLE OF APPLICATION

As we have seen, some tasks in log analysis require both symbolic and numeric processing. The determination of lithology, or rock type, is one such task. While a statistical technique applied on a point-by-point basis can yield ambiguous results, an expert system can be used to apply symbolic knowledge to choose among them. Furthermore, the results for a sequence of depths are often combined to delineate formations. This step may require additional symbolic processing (e.g., of the curve shapes).

We shall now examine some Interlog software to determine lithology. Although the example is restricted in scope, it demonstrates how Interlog software can integrate numeric and symbolic processing. By providing a representation for a central task of well log analysis, it also illustrates the suitability of the Interlog tool set for the development of expert systems in this domain.

The Interlog software in our example has two parts, corresponding to the two stages of lithological determination discussed above. The first is an FLOP representation of a *lithology classifier*, which assigns one of several lithology classes to a log data point on the basis of log data values (in this case, from the gamma and sonic curves). The second part is a method, or procedure, which applies this classifier to a given log database, to find zones of a particular type of lithology (in this case, *sand*).

To see the lithology classifier in action, let us follow the script of Figure 3. From the main log analysis menu (not shown), we open a log database using the *select* option, and print its characteristics with an I-4 statement. After entering the lithology analysis menu (also not shown), we choose the classifier *sandmod* shown in Figure 4.

The principal component of this FLOP lithology classifier is an F-LISP expression which associates numeric intervals with symbolic lithology names. While a detailed explanation of the numeric values used in this expression is beyond the scope of this paper, we can note that the gamma ray curves (GAM1 and GAM2) indicate the amount of clay in the rock, while the sonic curves (ITT1 and ITT2) indicate the porosity. Thus, in the classifier *sandmod*, a rock is a sandstone if it has low clay content and medium porosity. The particular numeric values used in the classifier depend on the geologic area

```
>select
=   Enter database name:
>ldat
=   LDAT
>ldat:print
=name     ldat
=top     5.0
=base     60.0
=incr     0.2
=units     2
=type     1
=nvals     276
=curves     nil
=cnames     (lith cnst vphi vqtz vcla odep itt2 phic cnls gaml vshl)
=   nil
>lithfind
=File "SANDMOD.INT" consulted.
=
%select
=                     ***** Lithology Analysis System *****
=                   Lithology classifier currently open:  nil
=
=   Please enter new value.
=Must be one of: (SANDMOD)
%sandmod
=   SANDMOD
%'set top,base'
=   Enter top and base depth for model application:
%5.0 54.2
=Top,base = 5.0,54.2
=   nil
%apply
=
=   *** lithology classification ***
=
=(plambda (d)
=         (cond ((and (in (gaml d) 0.0 75.0)
=                     (in (itt2 d) 180.0 356.0))
=                sand)
=               ((and (gt (gaml d) 75.0)
=                     (in (itt2 d) 180.0 356.0))
=                shale)
=               ((and (gt (gaml d) 0.0)(gt (itt2 d) 356.0)) coal)
=               (t other)))
=LDAT -- zone (SAND)  (5.0:25.2), (1:102)
=LDAT -- zone (SAND)  (28.6:37.2), (119:162)
=LDAT -- zone (SAND)  (43.8:47.2), (195:212)
=LDAT -- zone (SAND)  (50.4:54.2), (228:247)
>   nil
```

FIGURE 3: LITHOLOGY CLASSIFIER IN USE

```
name     sandmod

expr
   '(((and (or (in (gam1 d) 0.0 75.0)
               (in (gam2 d) 0.0 75.0))
           (or (in (itt2 d) 180.0 356.0)
               (in (itt1 d) 180.0 356.0)))
      sand)
     ((and (or (gt (gam1 d) 75.0)
               (gt (gam2 d) 75.0))
           (or (in (itt2 d) 180.0 356.0)
               (in (itt1 d) 180.0 356.0)))
      shale)
     ((and (or (gt (gam1 d) 0.0)
               (gt (gam2 d) 0.0))
           (or (gt (itt2 d) 356.0)
               (gt (itt1 d) 356.0)))
      coal)
     (t other))

curves   (gam1 gam2 itt2 itt1)

tfun     nil

selcur   nil

comps    (lith)
```

FIGURE 4: SIMPLE LITHOLOGY CLASSIFIER

under study.

As Figure 4 shows, the instance variable *expr* contains this F-LISP expression. Other instance variables provide information about the expression, such as the necessary input curves, for use by the procedures which manipulate the object. These procedures are displayed in Figure 5, which gives the FLOP class definition for a lithology classifier.

Having selected the *sandmod* classifier, we apply it to a portion of the current log database by invoking the *apply* method shown in Figure 5. This method generates and prints the F-LISP expression actually used to classify the log data, finds the *sand* zones, and plots the input and output curves on the screen.

To use the classifier on log data from a specified depth interval, the *apply* method evaluates the expression in *expr* for each included data point. In this sense the classifier behaves in a similar fashion to the statistical method described above. However, to minimize computation, and avoid evaluating boolean terms which refer to non-existent curves, it modifies the expression before applying it. In this example each disjunction is reduced to a single term, the one referring to the curve in the data set which is the most reliable and covers the most of the given depth interval out of all the curves used in the disjunction. (The terms in the disjunction are ordered according to the reliability of the curves they reference.) The resulting expression links rock properties with single curves, providing a simple but effective approach for a preliminary analysis of the data. Instead of applying the symbolic knowledge after the use of numerical methods, as LITHO does, Interlog uses it beforehand, to guide their application.

Using the modified boolean expression, the *apply* method assigns a lithology to each point in the specified interval, constructing zones of log data with a lithology of *sand*. To qualify as a *zone*, a series of points should have the same lithology and cover a given minimum length. However, a zone can contain *breaks* of points with a different lithology, provided these breaks do not exceed a given maximum length. The algorithm used is similar to one used in the Dipmeter Advisor [Smith and Young 1984]. As the zone objects are created, they are printed (Figure 3).

To allow the user to verify the classifier's choices, the *apply* method then generates a lithology curve using numeric codes for the lithology classes, and calls the *plot* method to display the input and output curves. The *plot* method provides a further example of Interlog's use of symbolic processing of the classifier to provide flexible software. When the *apply* method prunes the body of the classifier to create the test function for the given log database and depth interval, it fills in the *tfun* instance variable with the name of the function, the *selcur* variable with a list of the input curves, and the *comps* variable with a list of the computed curves (in this case, the single lithology curve). As the source code in Figure 5 shows, the *plot* method uses these variables to construct a plot which specifically displays the desired results without additional programming.

```
class              lithmod

cvars              (members)

cmethods (

  (make (lname lexpr)             ; create a new lithology classifier
     (: (: lithmod create lname) expr lexpr)
     (: (car lname) getcurves)))

superclass         object

vars               (name expr curves tfun selcur comps)

methods (

   (init ()                       ; initialize lithology classifier
      (setq comps '(lith)) self)


   (getcurves ()                  ; get list of curves used in lithmod
      (setq curves (lm:getcurves expr)))


   (defTest (database)            ; prune expression, define "(test d)"
      (prog (newmod)              ; ... which gives lithology for index "d"
      (setq newmod                ; ... (top at index=logstart)
         (lm:modunits             ; Method:  generate priorized curve list
            (lm:prune expr        ; ... from data, use it to reduce the
               (cvorder database  ; ... lithmod expression, using right units
                  (: database depth logstart)
                  (: database depth logstop)
                  (intersectq curves (: database cnames))))
            (: database units)))
      (setq selcur (lm:getcurves newmod))
      (setq tfun 'test)           ; now generate "test" function
      (put 'test 'expr
         (list 'plambda '(d) (cons 'cond newmod)))
      (pp (get 'test 'expr))      ; print new expression
      'test))


   (apply (idb)                   ; Apply lithmod to log database.  Top,base
                                  ; ...already set.
      (findzone idb self sand)                ; get "sand" zones
      (: scancmnd continue)                   ; pause for display
      (: curve open2 'lith dbalt)             ; open lithology curve
      (for d = logstart logstop               ; fill lithology curve
         (= (lith d) (getf (test d) 2)))      ; "(getf <lithology> 2)"
                                              ; ... same as "(: (test d) val)"
      (: self plot 0))                        ; plot results


   (plot (num)                                ; plot lithmod curves on screen
      (prog (tp bs)
      (real tp bs)                            ; top and base depths
      (= tp (: dbcurr depth logstart))        ; translate indices to depths
      (= bs (: dbcurr depth logstop))
      (vpinit dbcurr)                         ; initialize plotting
      (vpinit dbcurr selcur tp bs)            ; prepare to plot input curves
      (vidplot "PLOT")                        ; display curves
      (cond ((eq num 0)                       ; add output curves to picture
         (vpinit dbalt comps tp bs) (vidplot "ADD")))
      (vplotcmnd)                             ; allow user to enter plot cmnds
      nil)))
```

FIGURE 5: LITHOLOGY CLASSIFIER DEFINITION

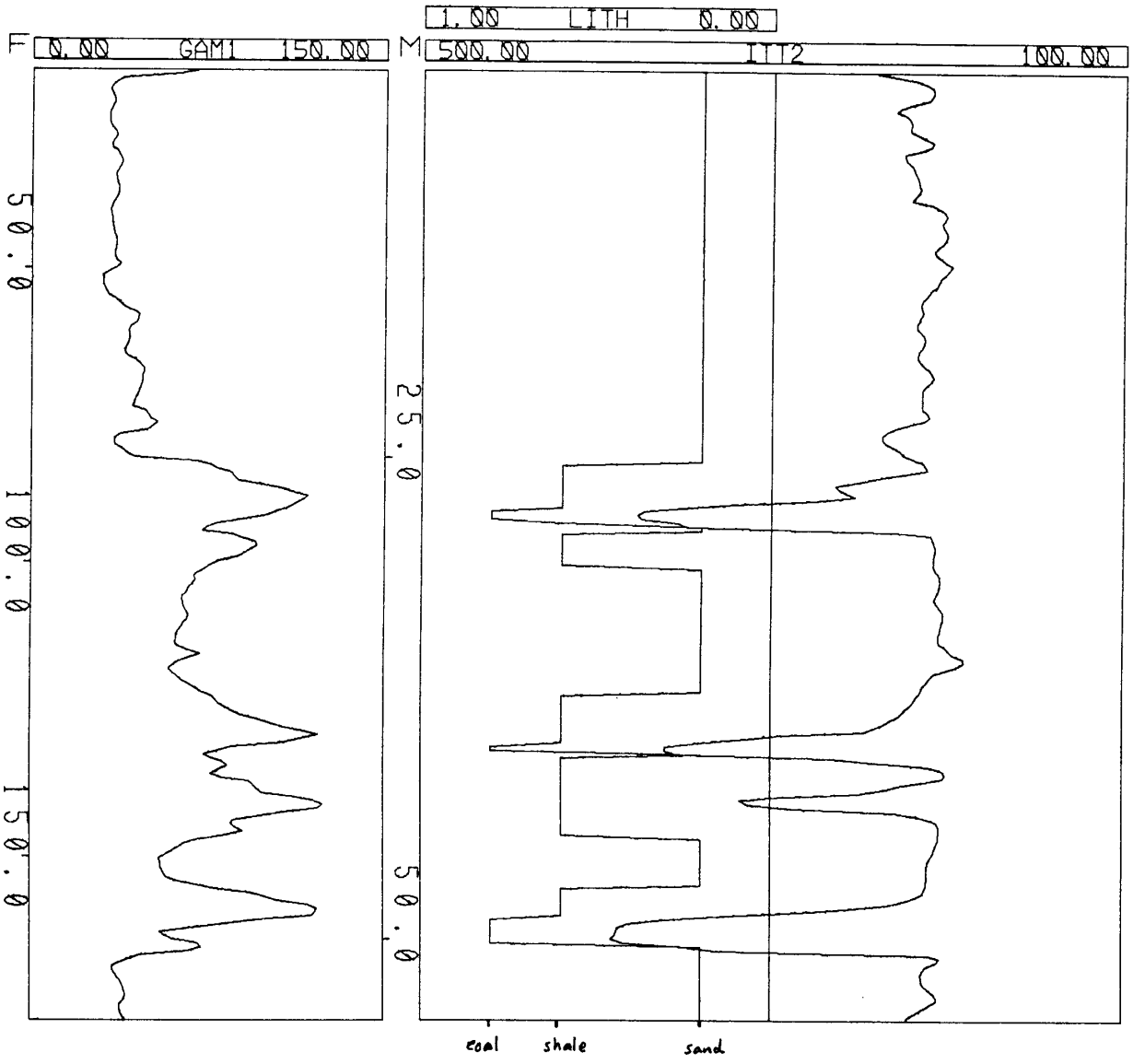

FIGURE 6: LITHOLOGY PLOT

Figure 6 displays the plot generated for the script of Figure 3.

Interlog's ability to treat programs as data, derived from its basis in LISP, allows log analysts and expert system builders to automate existing methods of log interpretation, and develop new ones, in ways not feasible with conventional software. As the *apply* method shows, the environment can provide a kind of "automatic programming," taking a general classifier provided by a log analyst and adapting it to different data sets to increase the efficiency of computation and the reliability of the result. This process can be readily extended. As the complexity of the lithology classifiers increases, the amount of symbolic knowledge necessary to apply them to specific cases may also increase; this knowledge could be encoded in a rule-based system. Finally, as the *plot* method illustrates, Interlog can use general-purpose procedures for manipulating entire families of these classifiers by referring to their structure. Such "metaknowledge" is essential for expert systems which reason about and employ numeric methods to achieve their results.

CONCLUSIONS

Expert systems for use in a particular domain should be able to use currently available data and software in that domain. Their developers should also be able to mix modules for symbolic and numeric processing, each written in an appropriate language. The Interlog programming environment illustrates one method of achieving these goals.

Interlog brings some of the facilities found in interactive programming environments such as Interlisp [Teitelman and Masinter 1981] to current time-sharing computer systems. For example, the screen editor provides a multi-window interface that is integrated with other programming tools, and works with remote, "dumb" terminals. The current implementation of Interlog runs only on Perkin Elmer minicomputers, but we plan to move it to the VAX/VMS environment. This step will allow Interlog to take advantage of a virtual memory operating system to incorporate many more FORTRAN routines for log analysis and other tasks. It will also greatly increase the size of F-LISP programs that can be run effectively.

Because Interlog contains interpreters for LISP, an object-oriented language, and a subset of FORTRAN, it can serve as a vehicle for the development of expert systems. Furthermore, because the LISP interpreter is written in FORTRAN rather than assembly language, Interog programs can call FORTRAN subroutines directly, and share data with them. These features allow programmers to build AI software which maximizes the use of existing data and code, as well as integrating numeric and symbolic processing.

---

1 Perkin Elmer is a trademark of Concurrent Computer Corporation

2 VAX and VMS are trademarks of Digital Equipment Corporation

Finally, Interlog provides programming tools specifically for the domain of log analysis. It uses a FORTRAN library and object-oriented representation to manage log data and share it between LISP and external FORTRAN programs. It also contains command interpreters which can be used to integrate a commercial log analysis package directly into the programming environment.

We could extend the Interlog programming environment to be useful in other scientific domains by incorporating external LISP and FORTRAN software. For example, while the current version of Interlog lacks many of the facilities found in full-fledged programming environments such as Interlisp, such as interactive debuggers and cross-reference generators, Interlog could acquire these features via LISP programs if appropriate. By the same token, it could also take advantage of existing LISP software for the development of expert systems. Of course, incompatibilities between data management techniques or even LISP dialects could be hurdles to making these improvements.

As some programs and expert systems are being developed with Interlog, the programming environment is acquiring facilities of the types described above. One project in progress is an expert system for the determination of formation water resistivity, a parameter used in log anlaysis. Development of this software has added modules for trend surface anlaysis and a rule base and inference engine. As FLOP objects, these components are readily available to future Interlog programs.

By providing tools for the development of expert systems in log analysis which can use existing data, software and hardware, Interlog illustrates several ways of bridging the gap between Artificial Intelligence techniques and commercial software development. The resulting programs could combine the best of both worlds, giving us expert systems which enhance existing software and are effective in current computing environments.

REFERENCES

Barstow, D.R., Shrobe, H.E., Sandewall, E., ed. (1984), *Interactive Programming Environments*. New York: McGraw-Hill, 1984, 609 pp.

Bobrow, D.G., and Stefik, M.J. (1986), "Perspectives on Artificial Intelligence Programming," *Science*, **231**, February 1986, 951-957; reprinted in *Readings in Artificial Intelligence and Software Engineering*, 581-587

Bonnet, A. and Dahan, C. (1983), "Oil-well data interpretation using Expert System and Pattern Recognition Technique," *Proc. IJCAI* **8** (August 1983), 185-189

Bonnet, A., Harry, J., and Gascia, J. (1982), "LITHO, an Expert System in Underground Geology," *Technique et Science Informatiques*, **1**, 5 (1982), 393-402 (in French)

Connolly, E.T., and Reed, P.A. (1983), "Full Spectrum Formation Evaluation," *CWLS J.*, **12**, 1 (December 1983), 23-70

Delfiner, P.C., Peyret, O., and Serra, O. (1984), "Automatic Determination of Lithology from Well Logs," *59th Ann. Fall Techn. Conf. SPE of AIME*, Houston (September 1984), Paper SPE 13290

Helander, D.P. (1983), *Fundamentals of Formation Evaluation*. Tulsa, Oklahoma: Oil and Gas Consultants International Publications, 1983, 332 pp.

Kernighan, B.W. and Mashey, J.R. (1981), "The UNIX Programming Environment," *Computer*, **14**, 4 (April 1981), 25-34; reprinted in *Interactive Programming Environments*, 175-197

McCarthy, J., Abrahams, P., Edwards, D., Hart, T, Levin, M. (1962), *LISP 1.5 Programmer's Manual*. Cambridge, Massachusetts: MIT Press, 1962

Rich, C., and Waters, R.C., ed. (1986), *Readings in Artificial Intelligence and Software Engineering*. Los Altos, California: Morgan Kaufmann, 1986, 602 pp.

Sandewall, E. (1978), "Programming in an Interactive Environment: the LISP Experience," *CACM*, **10**, 1 (March 1978), 35-71; reprinted in *Interactive Programming Environments*, 31-80

Sheil, B.A. (1983), "Power Tools for Programmers," *Datamation*, February 1983, 131-143; reprinted in *Interactive Programming Environments*, 19-30, and *Readings in Artificial Intelligence and Software Engineering*, 573-580

Smith, Reid G. and Young, Robert L. (1984), "Design of the Dipmeter Advisor System," Schlumberger-Doll Research, Technical Report SYS-009, September 1984, 16 pp.

Smith, Reid G., Lafue, Gilles M.E., Schoen, Eric, and Vestal, Stanley C. (1984), "Declarative Task Description as a User-Interface Structuring Mechanism," *IEEE Computer* **18**, 9 (September 1984), 29-38

Smoliar, S.W. (1983), "Software Specifications, Databases and Knowledge Bases," *Proceedings of the IFIP 9th World Computer Congress*, Paris, France (September 1983), 219-221

Snyder, D.D. and Fleming, D.B. (1985), "Well Logging -- A 25-Year Perspective," *Geophysics*, **50**, 12 (December 1985), 2504-2529

Teitelman, W. and Masinter, L. (1981), "The Interlisp Programming Environment," *Computer*, **14**, 4 (April 1981), 25-34; reprinted in *Interactive Programming Environments*, 83-96

Weiss, S.M. and Kulikowski, C.A. (1984), *A Practical Guide to Designing Expert Systems*. New Jersey: Rowman and Allanheld, 1984, 174 pp.

Xerox Learning Research Group (1981), "The Smalltalk-80 System," *Byte*, **6**, 8 (August 1981), 36-48