
Inconsistencies in a user interface arise primarily when implicit knowledge about individual applications is encoded. Separation of user interface from applications, together with declarative descriptions of applications, offers a solution.

Declarative Task Description as a User-Interface Structuring Mechanism

Reid G. Smith, Gilles M. E. Lafue, Eric Schoen, and Stanley C. Vestal
Schlumberger-Doll Research

One of the problems that often faces the user of a large software system is a lack of uniformity or consistency in the user interface. Responses that elicit the desired system behavior in one context may fail completely in another—often accompanied by an obscure error message. As a result, the user may be well aware that the user interface is a disparate collection of programs written by different programmers, each with a different view on interface design.

A related problem faces developers of an application intended to fit into an existing system. All too often they have to construct “system-tailored,” low-level user interface routines, usually without the benefit of the facilities and documentation available to the original system developers. This situation almost encourages the disorder and inconsistency that so often plagues an end user.

We have adopted the view that these problems arise primarily when implicit knowledge about individual tasks is encoded in the user interface. What results is an unclear separation between actual computations involved in the execution of a task and data acquisitions (often from an end user) needed to execute a task. We have therefore experimented with a system design in which the user interface has a minimal *a priori* knowledge of individual tasks. The interface has facilities for acquiring data using interaction mechanisms relevant to a particular task domain but has no knowledge of any particular task in that domain.

To effect this separation, we have used hierarchical, symbolic descriptions of computation of the type that has been applied in automatic programming,¹ fault diagnosis,² and digital hardware design.³ These descriptions incorporate a modular, object-oriented encapsulation of the information necessary to execute a task, an explicit representation of control and data flow, a formal description of the

function, and the facilities for viewing computation at a number of levels of abstraction. Existing formalisms, however, have not been generated with the user interface in mind.

Several types of input data information are usefully added for the purpose of structuring the user interface. For example, producing extensive descriptions of input data for the user interface has a two-fold result: the descriptions function as both constraints on the input data and as guidelines to the user interface concerning appropriate interaction mechanisms. The advantages of this approach to user interface design are:

- (1) It leads to greater uniformity in end-user data acquisition because the user interface is the only part of the system that makes the ultimate decisions about how to acquire data. Thus, users are presented with a more consistent view of the suite of programs with which they are interacting;
- (2) Information that specifies constraints on the data can be used both by the user interface (to make sure that the end user supplies valid data) and by the system, itself (for integrity maintenance), which leads to economy of mechanism; and
- (3) Addition of a new application to the system is simplified. The application developer does not have to be concerned with either writing user-input routines or altering the user interface. (This is true as long as the overall system is still operating in the same general applications domain—so that the types of interaction already built in are sufficient.) There is a clear separation between the role of the underlying system and the individual applications that are to operate within that system.

We can further illustrate the types of input data information used to structure the user interface by focusing our attention on a single task domain. For purposes of exposition, we will consider the problem of interpreting oil-well logs. The efficient location and production of hydrocarbons from an underground formation requires a great deal of information about the formation, ranging in scale from the size and shape of the rock's pore spaces to the size and shape of the entire reservoir. Some of this information can be inferred from oil-well logs. These logs contain data collected from instruments (called tools) lowered into the borehole; measurements are recorded as the tools are raised to the surface. Logging tools measure a variety of basic petrophysical properties such as the resistivity of the rock surrounding the borehole. The resulting logs are sequences of values indexed by depth.

Log interpretation is concerned with determining other kinds of information from these basic measurements, such as lithology—the set of minerals (and their relative volumes) that make up the rock around the borehole—and reservoir geometry and structure. Most of today's log interpretation programs apply numerical models to log data collected at the borehole over a number of intervals. In order to guide these interpretation programs, the end user must select appropriate models to be applied, the intervals over which they are to be applied, and a variety of parameters that specialize the models to the specific geologic environment around the well to which they are applied.

Our declarative task description formalism has been used in the construction of an interactive log-interpretation system, called Crystal. This system manages all data about a set of wells, the logging tools that have been used, and the log data. It leads a human interpreter through the process of interpreting the data and generating displays of the results. The system forms a substrate for end users to do interactive log interpretation and for application developers to write new interpretation methods.

We are by no means the first to suggest that the user interface should be a cleanly separated module.⁴ Furthermore, we are not the first to recognize the utility of declarative task description. We do, however, illustrate the advantages to be gained by combining these two ideas in the context of an implemented system.

Prior usage of declarative task description

Declarative task description has been used for a number of purposes: to capture hardware design and program development; to aid in diagnosing failures in digital circuits; and to verify the correctness of digital hardware designs. It permits answers to questions such as "What would be the output for this input?" "What is the role of this component in this circuit?" or "Is this module operating correctly?" It provides more support for reasoning about tasks than the empirical associations encoded in the rules of rule-based expert systems.

A declarative task description must capture (1) the computation carried out in each subtask (or subcomponent) of an overall process; (2) the relationships between tasks and their subtasks; (3) the control flow between tasks and sub-

tasks; and (4) description of input and output data (such as data flow and data specification).

In general, the focus of prior work in this area has been the description of the computation carried out in a task in terms of mappings between input data and output data.

The Rutgers AI/VLSI group^{3,5,6} has used extensive descriptions of the input and output data for digital modules. Its descriptions are intended to represent the *operating conditions* of a module (that is, what must be true of the input data for the module to be executable) and the *behavioral specification* of a module (that is, the relationships between the input data and the output data). These descriptions are used in concert with constraint propagation techniques to propagate constraints between specifications and behaviors, and with a body of circuit design rules to construct detailed designs from abstract specifications.

Genesereth⁷ has used a predicate-calculus encoding of similar information in a system to diagnose faults in digital hardware. This system proves the correctness of circuits by showing that their behaviors, inferred from the behaviors of their components, satisfy their specifications. This is done with a resolution-based theorem prover. Barrow⁸ has written a Prolog program with a similar purpose. Davis et al.²⁹ use networks of dependencies between constituents of digital circuits to simulate the behavior of the circuits and isolate faults.

The MIT Programmer's Apprentice group¹ has developed a formalism intended to facilitate modifications to existing programs by providing documentation about design decisions leading to these programs, and by detecting inconsistencies between these decisions and proposed modifications. This formalism, called Plan Calculus, also provides a library of standard constructs for program development. It borrows from control flow charts, data flow schemas, abstract data types, and program transformations. The user can construct or modify abstractions, or *plans*, of programs by using a diagrammatic notation.

The information that we encode indicates our rather different purpose: that of providing instructions to a user interface—information from which the user interface can derive appropriate interaction mechanisms. Our interest leads us to concentrate on control flow, task/subtask relationships, and, chiefly, on data description.

The declarative task description encoding

We are using declarative task description as a means of providing a well-structured interface for the end user and the applications developer. Our encoding formalism concerns data flow, control flow, and task/subtask relationships.

In our formalism, individual tasks are called *modules*. A module may be decomposed into *expansions* that represent its subtasks. Each module has a number of *ports*—either *input* or *output* ports—that specify the data that flow into the module and the data that are generated by the module.

We have encoded modules as objects in a structured object representation language called Strobe.¹⁰⁻¹² They are

subclasses of the class object module.* A module class object describes a type of task; a module instance object describes a particular execution of a task. The attributes, or properties, of a module (for example, its ports) are encoded as *slots*. Each slot is further described by a series of *facets*. For example, a slot is noted as being a port by giving it a *Role* facet whose value is port, and a *Direction* facet whose value is either *In* or *Out*.

The Eigen module shown in Example 1 is part of Schlumberger's Faciolog lithology interpretation module.¹³ We show here only some of the information associated with the module. Additional information will be discussed in later sections.

Object: Eigen
 Type: Class
 Generalizations: Module
 Well:
 Role: Port Direction: In
 Active-Logs:
 Role: Port Direction: In
 Passive-Logs:
 Role: Port Direction: In
 Top-Depth:
 Role: Port Direction: In
 Bottom-Depth:
 Role: Port Direction: In
 Weights-On-Active-Logs:
 Role: Port Direction: In
 Zones:
 Role: Port Direction: In

Weights-On-Zones:
 Role: Port Direction: In
 Log-Means:
 Role: Port Direction: Out
 Log-Standard-Deviations:
 Role: Port Direction: Out
 Eigen-values:
 Role: Port Direction: Out
 Eigen-vectors:
 Role: Port Direction: Out
 PC-Logs:
 Role: Port Direction: Out
 Code: Eigen-Procedure
 Return-Control: Eric-Prime Module Keyest
 Control

(1)

The task of Eigen in the overall computation is to perform a principal component analysis on the *Active-Logs* for *Well* from *Top-Depth* to *Bottom-Depth*. The main output is a set of principal component logs, *PC-Logs* in Example (1), which are projections of the normalized active logs onto principal component axes. Once the principal component logs have been determined, a later part of the computation determines vertical sections of the borehole that are statistically similar. Finally, lithological names are assigned to vertical sections of the borehole (such as sand and shale). Figure 1 shows a sample interpretation.

*We will describe only features of Strobe germane to the present discussion. Readers interested in a more detailed discussion of Strobe are directed to the references.

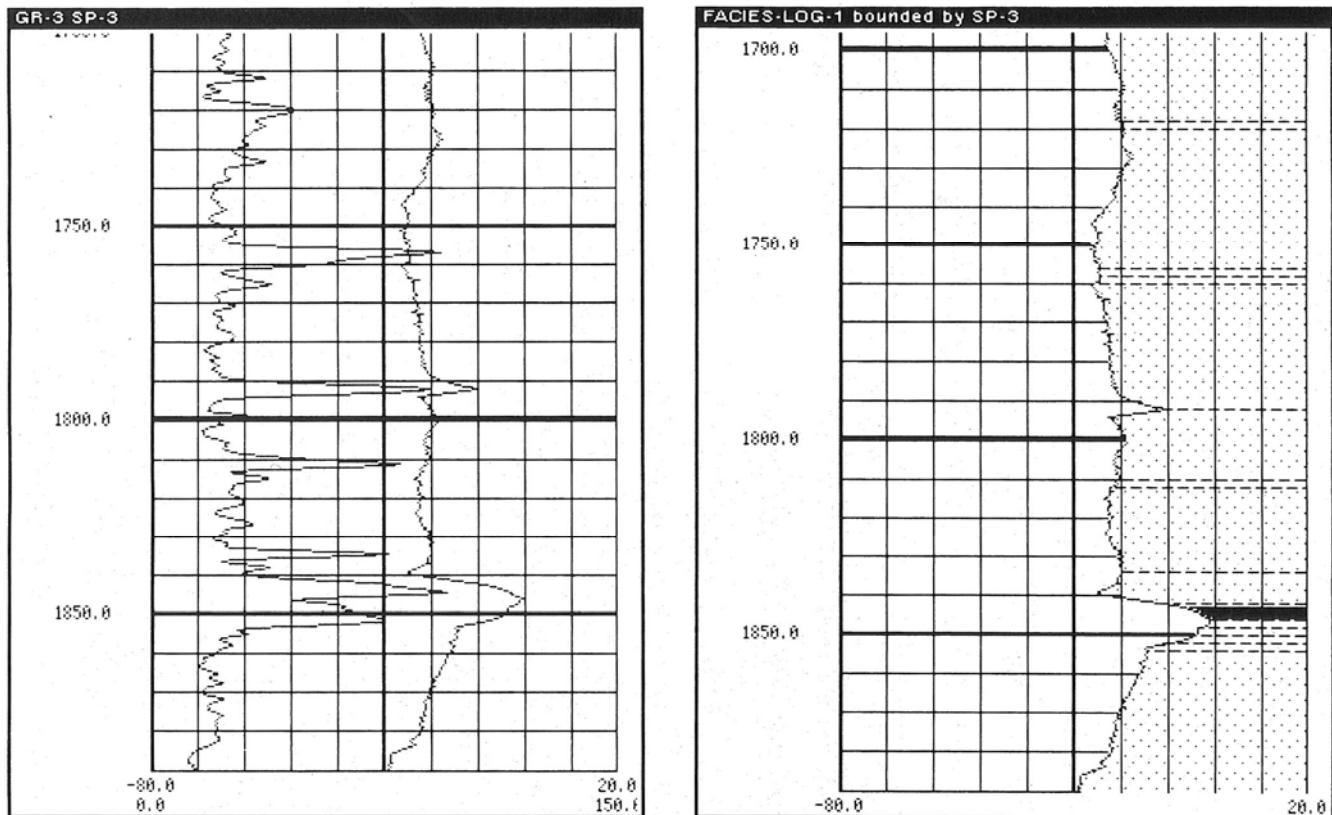


Figure 1. A sample lithological interpretation. The left channel shows a gamma ray log (which measures natural gamma radiation in the formation) plotted together with a spontaneous potential log (which measures variations in potential between an electrode lowered into the borehole and a reference electrode at the surface). The right channel shows the final interpretation of the interval. Each different lithology is shown by shading. The lithology textures are bounded by the spontaneous potential log, as is common for geologists.

Control flow. In keeping with our desire to encode no task-specific information in the user interface of our system, we use a single procedure to execute all modules. The following is a schematic description of the procedure.

```

Procedure Module-Control (Module)
  Acquired Data For Input Ports
  If Expansions Exist
    Then
      Instantiate Expansions
      Set Up Control Flow Information For Expansions
      Module Control (First Expansion)
    Else Execute Module Computation
  Module-Control (Next Module) (2)

```

This procedure is invoked by sending a message to the *Control* slot of the instantiation of the module. This has the effect of invoking the procedure found in the slot.[†] The message handler for the control slot is always inherited from the module class object.

The actual code to be executed to carry out the computation associated with the module is found in the code slot. In the Eigen module, Example (1), Eigen procedure makes the computation.

The *Return-Control* slot contains the address of the continuation of the module; that is, the module that gains control after the current module has been executed. In Eigen, its value is interpreted as “Continue execution by sending a message to the control slot of the *Keyest* object.”[‡]

[†]This type of indirect procedure invocation (by address rather than by name) is standard in object-oriented programming languages.

[‡]Keyest is the name of a program that performs clustering on the results of the principal component analysis, which is performed by Eigen.

in the module knowledge base on the network host machine called Eric-Prime.”

Task/subtask relationships. Example 3 illustrates the encoding of task/subtask relationships between modules. It is drawn from the Dipmeter Advisor system.¹⁴ This module represents the *Validity-Analysis* phase of the interpretation (We show only slots relevant to this discussion). *Validity-Analysis* expands into three modules (subtasks): *Blanks*, which finds zones in the well where no dip estimates could be made; *Washouts*, which finds zones where the borehole has collapsed; and *Mirror-Images*, which finds zones where an electrical disturbance has invalidated tool measurements. The expansions are executed in the order indicated by the *Order* facets. Expansions with the same integer in this facet may be executed in parallel. *Validity-Analysis* is, itself, an expansion of the Dipmeter Interpretation module. When *Validity-Analysis* is instantiated, the *Blanks*, *Washouts*, and *Mirror-Images* slots are filled with the instantiations of the expansion modules.

Object: Validity-Analysis		
Type: Class		
Generalizations: Module		
Blanks:		
Role: Expansion	Order: 1	
Washouts:		
Role: Expansion	Order: 2	
Mirror Images:		
Role: Expansion	Order: 3	
Dipmeter-Interpretation:		
Role: Abstraction		

(3)

There is no *Code* slot associated with the Validity-Analysis module. For simplicity, we have adopted the convention that code may only be associated with modules that do not have expansions. We call these modules *primitive*.

Note that when expansions are instantiated, the Return-Control slots of the instantiations must be filled (usually with sibling expansion instantiations). The last expansion must return control to the parent module.

Data Description. In this section, we discuss the types of information that can be provided by an application developer, and the ways in which this information can be used to provide a consistent interface to the end user. The viability of a user interface that does not encode explicit task-specific information depends critically on the language used to communicate the intentions of the application developer. The language must capture the scope of natural user interactions in the domain of application.

Data descriptions are contained in facets of the ports through which the data flows. Application developers must specify a variety of information about ports. This information falls into three basic categories: (1) specification of how data flows through ports and between modules, (2) constraints on valid input port values, and (3) advice to the user interface about appropriate mechanisms for acquiring data from an end user.

Dataflow. The facets that deal with data flow are *Direction*, *Order*, and *Origin*. As we have noted, the *Direction* facet indicates whether data flows into or out of the port with which it is associated. The *Order* facet indicates the relative order in which input ports must be filled (partial orders are allowed). This is to allow for the existence of dependencies between values selected for one input port and those chosen for other input ports. In the examples to follow, many of the expressions to be evaluated contain free variables in the form of port names. The binding environment for ports associates port names with their values in these expressions. Ordering ensures that ports referred to in this way will have values when required.

The *origin* facet indicates how the data for a port value can be obtained. Three possibilities have been found useful:

- (1) The data is produced by the module, itself.
- (2) The data is found in another module. In this case, the *Origin* facet contains a path to the data. For instance, Keyest is an expansion of the Faciolog module. It is executed after the Eigen module discussed earlier. Keyest has a Bottom-Depth port whose data is found in the Bottom-Depth port of Eigen. The *Origin* facet contains the expression (Bottom-Depth-Eigen-Faciolog). This is interpreted as “The origin of the Bottom-Depth port of the current Keyest is the Bottom-Depth port of the Eigen expansion of the current Faciolog.”
- (3) The data is obtained from the end user.

Constraints on port values. The facets used to specify constraints on port values are *Datatype*, *Candidates*, *Predicate*, and *Set-Predicate*. They can be used both by a user interface and by a system integrity maintenance mechanism.

The Datatype facet is used to express constraints that are independent of the particular module with which the port is associated. The facets associated with the Active-Logs port of Eigen, for example, are shown in Example 4. The datatype of this port is *Log*. This means that the port value may be a Log or a set of Logs. It indicates that the value (or its elements) must obey all of the defining characteristics of the Log object. *

```
Object: Eigen
Active-Logs:
  Role: port
  Direction: In
  Origin: User
  Order: 2
  Datatype: Log
  Minimum-Cardinality: 1
  Maximum-Cardinality: 30
  Candidates: (Message Well 'Logs)
  Set-Predicate: (< = (for Log in Active-Logs count
    Log when (Generalization? Log 'Gamma-Ray)) 1)
Set-Predicate-Error-Message: "You cannot select more than one gamma ray log."
Bottom-Depth
  Role: Port
  Direction: In
  Origin: User
  Order: 5
  Datatype: Depth
  Predicate: (> Bottom-Depth Top-Depth)
  Predicate-Error-Message: "The Bottom-Depth
    must be greater than the Top-Depth."
  Default: (Message Active-Logs 'Bottom-Depth) (4)
```

Restrictions associated with a particular datatype may reference other facets associated with the port (such as *Minimum-Cardinality* and *Maximum-Cardinality*).

The Candidates facet contains an expression that evaluates to the set of valid alternatives. Active Logs, for example, must be drawn from the set of logs associated with the Well that has been selected for Eigen. (Note, here, the use of a free variable, Well, that is bound to the particular well that has been selected.)

The Predicate facet contains an expression that must evaluate to “true” for the port value to be valid. This facet encodes constraints that are specific to a particular port. For example, the value of the Bottom-Depth port of Eigen must be greater than the value of the Top-Depth port.

When the port value is a set, the Set-Predicate facet encodes constraints on the set, itself, and the Predicate facet encodes constraints that must be met by its elements. For example, the value of the set of Active-Logs of Eigen must be such that it includes only a single gamma ray log.

The current system simply rejects port values that violate constraints. We plan to extend it to consider other actions in response to violations.

Advice to the user interface. The remaining facets are used to provide advice to the user interface about appropriate interaction mechanisms for acquiring data from an end user. We expect that a user interface designed for interactions within a particular task domain can infer most of what it needs from the information just presented in

* All Strobe slots have datatype facets. The operations and constraints that define a datatype are contained in an object in any knowledge base.

order to select an interaction mechanism. For example, we assume that the Datatype facet will suggest, in many cases, the appropriate interaction mechanism. It might lead, for example, to a keyboard interaction for acquiring port values with datatype Integer. Similarly, the Candidates facet information, together with the datatype, well, might suggest a menu interaction.

The facets discussed here provide information for a user interface but cannot be inferred from the other facet information. They are *Interact-For-Value*, *Default*, *Default-Or-No-Value*, *Help*, *Predicate-Error-Message*, *Set-Predicate-Error-Message*, and *Associations*.

The Interact-For-Value facet contains an expression that evaluates to true if the end user should indeed be queried for the port value. It allows for optional ports—those that are not always required to execute the module.

The Default facet contains an expression that evaluates to a default value. The default must obey the constraints on the port value.

The Default-Or-No-Value facet is used if the Interact-For-Value facet evaluates to false and there is a default. If Default-Or-No-Value evaluates to true, then the default will be used as the value. Otherwise, the port is considered to have no value.

The Help facet contains an expression that evaluates to a help message in case the end users do not understand what is expected of them.

Predicate-Error-Message contains an expression that evaluates to a message that can be used to reprompt end-users in the event they supply data that violates the Predicate constraint. Set Predicate-Error-Message plays a similar role with respect to the Set-Predicate constraint. Our current implementation is relatively crude. We intend to extend this mechanism to use information on how the constraints have been violated to generate the error message.

The Associations facet identifies information whose association with ports is useful or necessary for selecting port values. Several kinds of information can be involved in such associations. This is another example of knowledge in the user interface about the particular task domain in which the system is intended to operate.

In the log interpretation domain, parameters may be associated with particular logs—either because their values are to be selected from these logs or because the display of the logs is a helpful aid for the end user in selecting values. For the GRF/GRB-Zones port of the *Bound-Water-Saturation* module shown in Example 5^{*} an association is made between this port and the GR port. In the Crystal system, the user interface notes that GRF/GRB-Zones has datatype Zone. Because it is tuned to the log interpretation domain, it knows that zones are most naturally entered by buttoning in a log display. From the association indicated between the GRF/GRB-Zones port and the GR port, the user interface determines that the particular log display from which the zones should be selected is a display of the GR log. Similarly, in the GRF and GRB ports of the same

module, the user interface can use the associations together with its knowledge of the task domain to infer that GRF and GRB are most naturally selected by buttoning in a display of the GR log. In this case, however, the value to be returned from the display is the log value, rather than the depth values as in the GRF/GRB-Zones case.

Another example of the utility of associations in the log interpretation domain involves port values associated with vertical zones in a borehole (that is, ports whose values vary from one zone of the borehole to the next). The zones, themselves, are also ports and provide indices for the zoned values. This indexing allows the user interface to identify all the ports indexed by the same zones. It can then ask the user to select the values of these ports at the same time. In the GRF and GRB ports of the Bound-Water-Saturation module, for example, the parameters GRB and GRF are indexed according to the same zone because their Associations facets point to the zone port GRF/GRB-Zones. Note, also, that the values of these two ports can be selected at the same time, as indicated by the fact that they have the same order.

Object: Bound-Water-Saturation

GRF/GRB-Zones:

Role: Port
Direction: In
Origin: User
Order: 3
Datatype: Zone
Default: (Message GR
'Depth-Interval)
Associations: ((Log . GR))

GRF:

Role: Port
Direction: In
Origin: User
Order: 4
Datatype: Radio-Activity-
Measurement
Default: 10.0
Associations: ((Log . GR)
(Zones . GRF/GRB-Zones))

GRB:

Role: Port
Direction: In
Origin: User
Order: 4
Datatype: Radio-Activity-
Measurement
Default: 70.0
Associations: ((Log . GR)
(Zones . GRF/GRB-Zones))

(5)

We have already noted that constraints on port values can be used both by the user interface to make sure that an end user has entered valid data and for integrity maintenance. It is worth noting that the complete flow of information associated with a port can be used to prevent costly integrity-maintenance checking when it is not required. For example, it can be inferred that the checking normally associated with a particular datatype is not necessary when the value is selected from a set of candidates and when the method used to generate these candidates can be assumed sound. The same reasoning applies to defaults.

*This is an expansion of a module that performs volumetric analysis of the rock surrounding a borehole. It computes the value of the Bound-Water-Saturation of a zone from the gamma ray log. GRF refers to the value of the gamma ray log in pure sand, and GRB to the value of the gamma-ray log in pure shale.

Interaction with the end user

Figure 2 is a snapshot of Crystal running on a Xerox 1100 series workstation during an interpretation session.[†] Commands are issued primarily through overlaying icons, much as in the Xerox Star user interface.¹⁵ Some icons represent devices, others represent interpretation tasks. In Figure 2, the user has placed the Compute-Facies icon on top of the Vax icon; in response, the system has started a Faciolog module, which contains the Eigen and Keyest expansions.

Each running task has its own terminal interaction window. This window, divided into three sections, is comprised of a menu listing the expansions of the module with indentations to represent expansion nesting; a menu of commands the user may issue to affect the execution of a

[†]Crystal is implemented in Interlisp-D, a dialect of Lisp that includes windows, menus, and bitmaps as primitive datatypes.

module; and a typescript and keyboard interaction region. As execution of a module progresses, a gray bar covers the expansion in progress in the expansion menu. The module structure leads the user through the interpretation. The user is prompted to supply input port values when necessary in the lower half of the window; values enclosed in brackets indicate defaults, which are obtained from the port descriptions. Menus appropriate for selecting symbolic parameters (such as names of wells or logs) often appear connected the window; as discussed in the previous section, numeric parameter menus associated with log values appear near the appropriate log display.

Two log displays appear across the top half of the screen (see Figure 2). On the left, the raw gamma ray and spontaneous potential logs are plotted in a single window; these logs served as input data for a Faciolog interpretation. The results of this Faciolog execution are displayed on the right. In Figure 2, the system is querying the user for

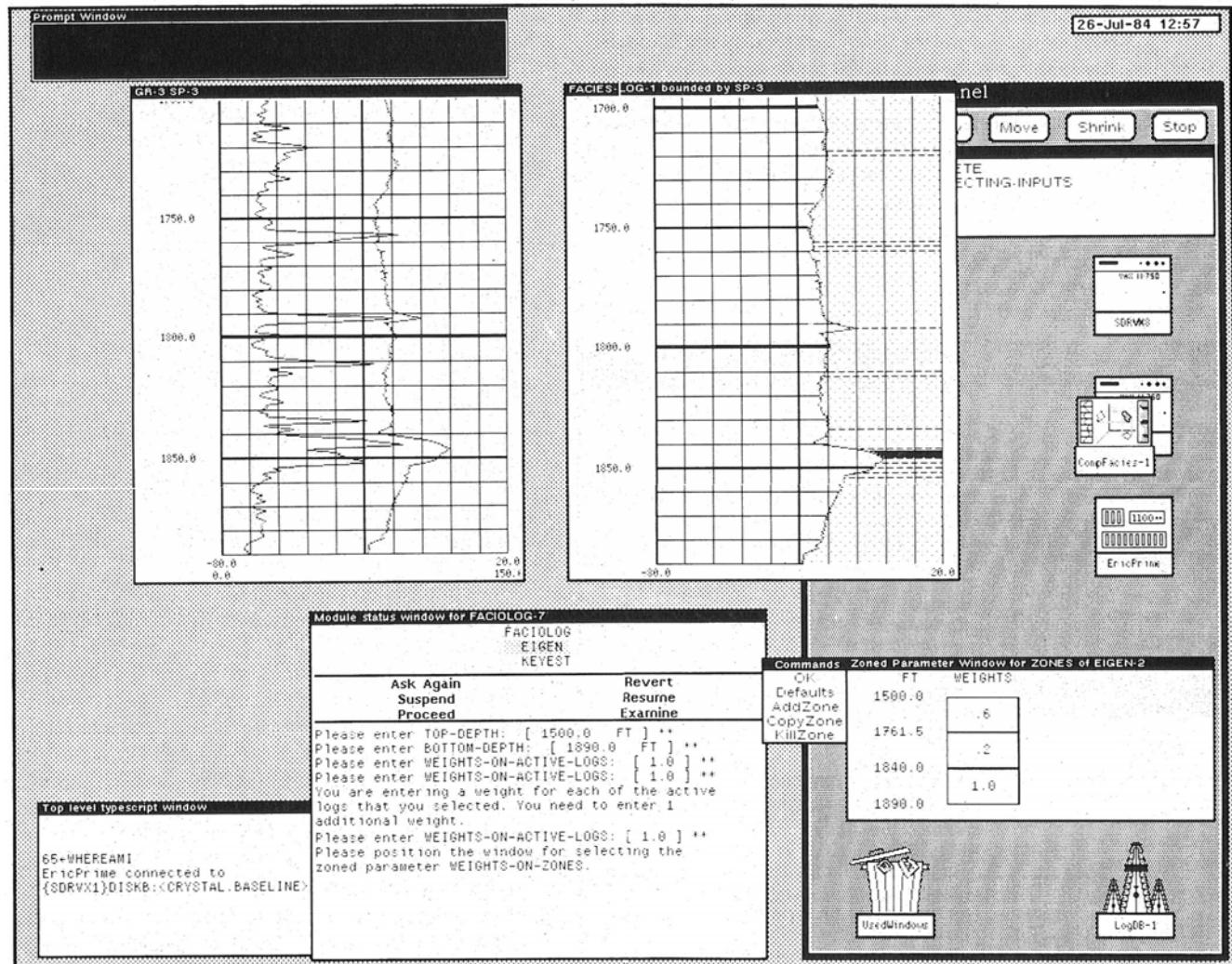


Figure 2. A snapshot of Crystal, running on a Xerox 1100 series workstation, during an interpretation session. The tail, partially obscured window on the right is the "Control Panel," which serves to focus the user's attention for the purpose of issuing commands. Commands are primarily issued through overlaying icons. In this example the user has placed the Compute Facies icon on top of the Vax icon; in response, the system has started a Faciolog module, which contains the Eigen and Keyest expansions. The Faciolog interaction window, lower center of the screen, contains (1) a menu listing the expansions of the module, (2) a menu of commands, and (3) a typescript and keyboard interaction region. The two log displays (see Figure 1) appear across the top of the screen.

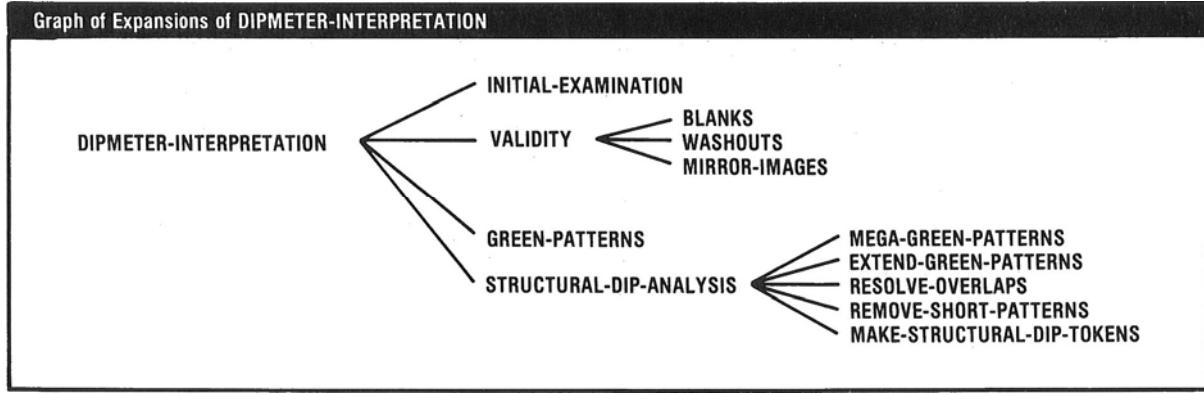


Figure 3. Expansions of Dipmeter Interpretation.

zone boundaries through a menu near the spontaneous potential log display. (If the user supplies a weight of 0.0 for a zone, that zone will be ignored during the Eigen computation). At this point in the interaction, the system has already queried the user for the particular well of interest, the well-logs to be used in the computation, and the zone of interest (as delimited by its top and bottom depths). Two logs were, in fact, selected by the user. As a result, the system asked twice for *Weights-On-Active-logs* (one weight for each log). Help messages are generated in response to the question.

Interaction with the application developer

Impulse, the primary tool we provide to an application developer, permits rapid, interactive construction of collections of modules.*

Interaction mechanisms. During a development session, the application developer often makes changes to several modules in concert. In the course of developing an application, changes to the control or data flow in one module may require corresponding changes in other modules. Impulse supports this style of editing by providing simultaneous access to any number of editor windows. The editor automatically assumes the proper editing "context" when a command is issued in an editor window. This means the application developer need not keep track of the implicit editor state. Furthermore, in order to assist the developer to focus more clearly on individual contexts, Impulse partitions displayed information according to different levels in the Strobe hierarchy (that is, Knowledge Base, Object, Slot, and Facet). For example, information pertinent to a particular module is restricted to a single editor window, information about other modules or about collections of modules appears in different editor windows.

Comfortable interaction with the system is also important. We have observed that mixed mouse and keyboard interaction is awkward; a typical user prefers not to remove his hand from the mouse to type on the keyboard. We have therefore chosen to emphasize one style of interaction over the other; a menu-driven style of interaction offers major advantages over a multiple-keystroke-driven style. As a result, Impulse editing contexts are composed of windows, which contain information, and associated menus, which contain commands. In addition, a number of editing functions require the user to supply some keyword such as a module name. Wherever feasible, Impulse provides menu selection in these cases. In practice, the repertoire of editor commands available through menus has proved complete enough that developers rarely need to resort to typing. However, when typing is unavoidable, Impulse supplies spelling correction and partial name recognition.

*Impulse is, in fact, the Strobe editor and is in common use throughout Schlumberger for development of knowledge-based systems. It serves much the same role in Strobe that the Browser does in Smalltalk.¹⁶

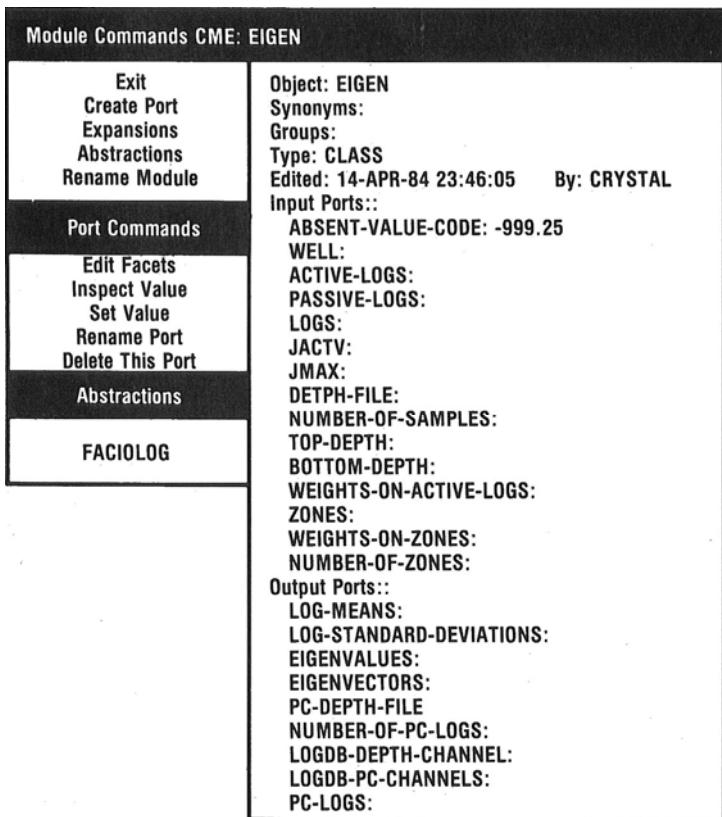


Figure 4. Eigen as depicted by the module editor.

A menu-driven approach does pose a problem: menus with many commands. These menus force the user to search for a desired command. They also consume precious screen space. We solved this problem in three ways: by partitioning command menus according to function, by using pop-up subcommand menus for related, but more specific, commands, and by using fixed-height scrolling menus.

Displaying knowledge-base structure. One of the most important facilities a knowledge-representation system editor can provide is a means of visualizing various relationships in a knowledge base. Not only does visualization give the developer a grasp of the current state of the system, but it allows easy access to collections of interrelated objects for editing. Impulse satisfies this need through its facilities for generating graphics of interobject relationships. Three standard graph styles are built into Impulse; two trace taxonomic hierarchies, either by ancestry or by progeny links; the third follows "slot succession," that is, objects linked by a common slot name (for example, an object might contain a *Parts* slot containing a list of subsidiary objects). The nodes of an Impulse display are *live*; buttoning a node which represents an existing object opens an object editor window for that object.

In addition to these predefined graphs, Impulse allows the user to specify graphs following any arbitrary relationship. The developer is prompted to define a primitive function to return a node's children; Impulse, itself, causes the function's recursive invocation and collects the graph nodes as they are generated. As an example, in the Module knowledge base, the Expansions graph is produced by recursively calling a function which collects the names of a module's Expansion slots. The resulting graph depicts the expansion hierarchy for a particular module in the knowledge base. Figure 3 is the Expansions graph of the *Dipmeter-Interpretation* module during development.

Customized editor interfaces. Impulse was originally designed for developing large object-oriented systems in Strobe. We perceive much of its functionality to be unnecessary and distracting for the much simpler task of designing new modules within our formalism. A more appropriate interface for the applications developer is one in which the general Impulse menu-and-window structure is preserved, but whose display and command set creates and edits module descriptions.

For this reason, Impulse was recently reimplemented to allow declarative representations of editors, themselves. Then it was simple to create a *module editor* that offered commands related specifically to editing modules and ports within the Crystal framework (see Figure 4).

We have described the utility of declarative task description as a way of structuring user interfaces for two types of user: the application developer and the end user. It has been shown that explicit representation of control and data flow, and hierarchical task/subtask relationships can be used to guide an end user through a complex computation and to provide a uniform framework for application developers. We have used extensive descriptions of in-

put data for modules to advantage in specifying data flow, constraints on port values, and advice to a user interface.

We have illustrated the utility of clear separation of the user interface from the modules described by application developers. The separation, however, is not total. Encoding task-specific information in the user interface makes it difficult to add new applications. However, tuning the user interface to a particular task domain enables it to use all the information in module data descriptions, including information not specifically intended for the purpose, to identify natural interaction mechanisms in that domain.

The work that we have done fits into a larger context—that of constructing more versatile and robust expert systems. It is becoming increasingly important to integrate existing large, traditional interpretation programs (often involving application of numerical models) with advisory systems that typically employ symbolic methods. Early approaches to this problem have involved writing production rules to guide the actions of the numerical programs and, to some extent, explain their actions.^{17,18} One of the long-term goals that we are pursuing is improvement upon these approaches through the use of hierarchical structural and functional descriptions of the tasks that are carried out by an existing assemblage of programs.*

Acknowledgments

The authors wish to acknowledge the rest of the team that designed and constructed the Crystal system: Peter Day, Bob Dennis, Ed Dolph, Dan Ehrlich, Bob Everett, Ray Kocian, Scott Marks, Peter Schachte, Bob Snow, and Peter Will. Dave Barstow, Brij Masand, Tom Mitchell, Stephen Smoliar, and Bob Young made a number of helpful comments on an early draft of this article.

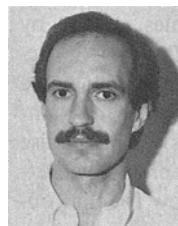
References

1. C. Rich, "A Formal Representation For Plans In The Programmer's Apprentice," *Proc. Seventh Int'l Joint Conf. on AI*, Aug. 1981, pp. 1044-1052.
2. R. Davis and H. Shrobe, "Representing Structure and Behavior of Digital Hardware," *Computer*, Vol. 16, No. 10, Oct. 1983, pp. 75-82.
3. T. M. Mitchell et al., "Representations For Reasoning About Digital Circuits," *Proc. Seventh Int'l Joint Conf. on AI* Aug, 1981, pp. 343-344.
4. D. A. Norman, "Design Principles For Human-Computer Interfaces," *Proc. CHI 1983 Conf. on Human Factors In Computer Systems*, ACM-SIGCHI, Dec. 1983, pp. 1-10.
5. V. E. Kelly and L. I. Steinberg, "The Critter System: Analyzing Digital Circuits By Propagating Behaviors And Specifications," *Proc. Nat'l Conf. on AI*, Aug. 1982, pp. 284-289.
6. T. M. Mitchell et al., "An Intelligent Aid For Circuit Redesign," *Proc. Nat'l Conf. on AI*, Aug. 1983, pp. 274-278.
7. M. R. Genesereth, "Diagnosis Using Hierarchical Design Models," *Proc. Nat'l Conf. on AI*, Aug. 1982, pp. 278-283.
8. H. G. Barrow, "Proving The Correctness Of Digital Hardware Designs," *Proc. Nat'l Conf. on AI*, Aug. 1983, pp. 17-21.

9. R. Davis, et al., "Diagnosis Based On Description Of Structure and Function," *Proc. Nat'l Conf. on AI*, August 1982, pp. 137-142.
10. R. G. Smith, "Strobe: Support For Structured Object Knowledge Representation," *Proc. Eighth Int'l Joint Conf. on AI*, Aug. 1983, pp. 855-858.
11. E. Schoen and R. G. Smith, "Impulse, A Display-Oriented Editor for Strobe," *Proc. Nat'l Conf. on AI*, Aug. 1983, pp. 356-358.
12. R. G. Smith, "Structured Object Programming In Strobe," tech. report SYS-84-08, Schlumberger-Doll Research, Ridgefield, Conn., March 1984.
13. M. Wolff and J. Pelissier-Combescure, "Faciolog: Automatic Electrofacies Determination," *23rd Annual Logging Symp. SPWLA*, July 1982, pp. 1-23.
14. R. G. Smith and J. D. Baker, "The Dipmeter Advisor System: A Case Study In Commercial Expert System Development," *Proc. Eighth Int'l Joint Conf. on AI*, Aug. 1983, pp. 122-129.
15. D. C. Smith, et al., "The Star User Interface: An Overview," *Proc. NCC*, AFIPS Press, 1982, pp. 515-528.
16. Adele Goldberg, "The Influence of an Object-Oriented Language on the Programming Environment," *Interactive Programming Environments*, D. Barstow, H. Shrobe, and E. Sandewall, eds., McGraw-Hill, Hightstown, N.J., 1984, pp. 141-174.
17. J. S. Bennett and R. S. Engelmore, "SACON: A Knowledge-Based Consultant For Structural Analysis," *Proc. Sixth Int'l Joint Conf. on AI*, Aug. 1979, pp. 47-49.
18. S. Weiss, et al., "Building Expert Systems For Controlling Complex Programs," *Proc. Nat'l Conf. on AI*, Aug. 1982, pp. 322-326.



Reid G. Smith is the program leader for Expert Geology Systems at Schlumberger-Doll Research, Ridgefield, Connecticut, where he has been since 1982. His current research focuses on expert systems that explain failures and develop justifications for the information in their knowledge bases. He has been involved in the Dipmeter Advisor project and in the development of tools for expert system construction. Smith has also worked on knowledge-based systems for passive sonar interpretation. Smith received his B. Eng. (electrical) and M. Eng. (electrical) degrees from Carleton University, Ottawa, in 1968 and 1969 respectively, and PhD degree in electrical engineering from Stanford in 1979. He is the author of *A Framework for Distributed Problem Solving* (UMI Press). He is a member of IEEE, ACM, AAAI, and CSCSI.



Gilles M. E. Lafue has been a staff member at Schlumberger-Doll Research since 1982. Previously he was on the computer science faculty at Rutgers University. His current research interests are in expert systems and databases. He is working on knowledge representation, automatic management of data semantic integrity and on combining A.I. knowledge representation systems with DBMS's data management capabilities. He has published in Computer-Aided Design and in Databases. He received a Master's degree in architecture from the Ecole Nationale Supérieure des Beaux-Arts in Paris, France, in 1972, and a PhD in computer-aided design from Carnegie-Mellon University in 1979. He is a member of ACM, IEEE Computer Society, and AAAI.



Eric Schoen holds BS and MS degrees in electrical engineering from Stanford University. From 1982 to 1984, he worked at Schlumberger-Doll Research, in Ridgefield, CT, designing tools to support both knowledge representation and distributed problem solving systems. He is currently enrolled in a computer science doctoral program at Stanford.



Stanley C. Vestal is software coordinator for the Austin Engineering Center of Schlumberger Well Services. The work described in this paper was performed while he was the Program Leader for the Advanced Computing Environments Program at Schlumberger-Doll Research Center. His technical interests include man-machine interfaces, distributed problem solving and computing and programming environments to support scientists. He previously worked with Honeywell in the areas of programming methodologies, large scale software development machines, and the Ada programming language. Vestal received a BA in Mathematics and an MS in computer science from the University of Kansas. He is a member of IEEE Computer Society, ACM, and AAAI.

Questions about this article can be directed to either Smith or Lafue at Schlumberger-Doll, Old Quarry Rd., Ridgefield, CT 06877.