Representation of Task-Specific Knowledge in a Gracefully Interacting User Interface

Eugene Ball and Phil Hayes

Computer Science Department, Carnegie-Mellon University
Pittsburgh, PA 15213, USA

Abstract

Command interfaces to current interactive systems often appear inflexible and unfriendly to casual and expert users alike.[1] We are constructing an interface that will behave more cooperatively (by correcting spelling and grammatical errors, asking the user to resolve ambiguities in subparts of commands, etc.). Given that present-day interfaces often absorb a major portion of implementation effort, such a gracefully interacting interface can only be practical if it is independent of the specific tool or functional subsystem with which it is used.

Our interface is tool-independent in the sense that all its information about a particular tool is expressed in a declarative tool description. This tool description contains schemas for each operation that the tool can perform, and for each kind of object known to the system. The operation schemas describe the relevant parameters, their types and defaults, and the object schemas give corresponding structural descriptions in terms of defining and derived subcomponents. The schemas also include input syntax, display formats, and explanatory text. We discuss how these schemas can be used by the tool-independent interface to provide a graceful interface to the tool they describe.

1. Introduction

Command interfaces to most current interactive computer systems tend to be inflexible and unfriendly. If the user of such a system issues a command with a trivial (to a human) syntactic error, he is likely to receive an uninformative error message, and must re-enter the entire command. The system is incapable of correcting the error in the "obvious" way, or of asking him to retype only the erroneous segment, or of providing an explanation of what the correct syntax really is. Anyone who has used an interactive computing system is only too familiar with such situations, and knows well how frustrating and time-consuming they are, for expert as well as novice users.

We are involved in a project to build an interface which will behave in a more flexible and friendly way, one that will *interact gracefully*. As we have described in earlier work [3, 4], graceful interaction involves a number of relatively independent skills including:

- the parsing of ungrammatical input, either to correct it or to recognize any grammatical substrings;
- robust communication techniques to ensure that any assumptions the system makes about the user's intentions are implicitly or explicitly confirmed by the user;
- the ability to give explanations of how to use the system or the system's current state;
- interacting to resolve ambiguities or contradictions in the user's specification of objects known to the system;
- keeping track of the user's focus of attention;
- describing system objects in terms appropriate to the current dialogue context.

Providing these facilities is clearly a major programming task requiring extensive use of Artificial Intelligence techniques (see [2] for just the flexible parsing aspect). We believe that it is unrealistic to expect the designers of each interactive sub-system (or *tool*) to implement a user interface with these capabilities. Therefore, instead of constructing a gracefully interacting interface for a single application, we are attempting to build a tool *independent* system, which can serve as the user interface for a variety of functional sub-systems.

The availability of a tool independent user interface would greatly simplify the construction of new computer sub-systems. Currently, even if the new system is not intended to be gracefully interacting but merely to perform according to minimal standards, a large amount of implementation effort must be devoted to user interface issues. The system designers must decide on a style of interaction with the user, select the general format and detailed syntax of all commands, and provide for the detection of illegal input. The command language must then be thoroughly checked to ensure that it does not contain ambiguities or misleading constructions and that likely error sequences will not be misinterpreted and cause unrecoverable system actions. Often, the design can only be completed after an initial implementation of the system has produced feedback about the usability of the human interface.

This design process represents the *minimum* effort necessary to produce a system that is even *usable* by a large number of people; if a superior (but still far from gracefully interacting) interface or one which can be used by non-programmers is required, much more work must be expended. Editing facilities, which are required in most interactive systems (at least for correction of typed input), must be fully integrated into the sub-system; compatibility with other editors in common use on the computer must be considered, even though this may lead to difficult interactions with the sub-system command language. Error detection and reporting must be improved; generating coherent diagnostics for the inexperienced user can be very difficult indeed. Online documentation must be provided,

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

including reasonable facilities which allow a user to quickly find the answer to specific (although vaguely expressed) questions. The complexity of this task often means that most of the implementation effort in adding a new tool to a computer system is absorbed by the user interface.

Technological trends are aggravating the problem by raising the level of performance expected of an interface. In particular, as high-resolution graphics displays equipped with pointing devices become available, users expect to use menu-selection and other more sophisticated forms of input, and to see system output displayed in an attractive graphical format. The very recent, but growing, availability of speech input and output will intensify this pressure for sophistication.

An additional reason for constructing a tool-independent interface is to make the computer system as a whole appear consistent to the user. If the interfaces for different tools use different conventions, then no matter how sophisticated each of them is individually, the user is likely to be confused as he moves from one to another because the expectations raised by one may not be filled by the other.

For all these reasons, we are attempting to make our gracefully interacting interface as tool-independent as possible. In the remainder of this paper we outline the system structure we have developed, and go on to give further details about one component of this structure, the declarative format in which information about the tool is made available to the interface, together with sketches of how the tool-independent part of the system uses the information thus represented.

2. System Structure

The basis for our system structure is the requirement that the interface contain no tool-dependent information. All such information must be contained in a declarative data base called the *tool description*. In an effort further to improve portability and reduce duplication of effort between interfaces implemented on different hardware configurations, we have made a second major separation between the device-dependent and -independent parts of the interface. The resulting structure is illustrated in figure 1.

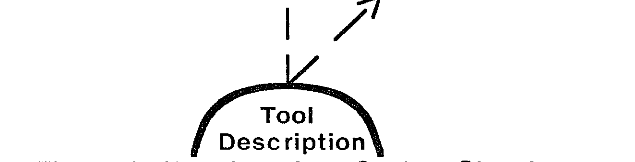

Figure 1. User Interface System Structure

The intelligent functions of the interface, those itemized above, are isolated in a tool and device independent *User Agent*, which interacts with the tool through a narrow interface that is completely specified by the declarative tool description. Communication between the Agent and the user is not direct, but goes via a device-dependent *Front-End*, which allows the Agent to specify its output in a high-level device-independent manner, and which preprocesses the user's input into a standard, device-independent, format. Communication between the Agent and Front-End is thus restricted to a well-defined format of input and output requests. Display formats in which to realize the tool's and Agent's high-level output requests are specified declaratively in the tool description.

The basic function of the Agent is to establish from the user's input what functional capability of the tool the user wishes to invoke and with what parameters he wishes to invoke it. Once this is established, the Agent issues the appropriate request to the tool and reports to the user relevant portions of the tool's response. To make this possible, the tool description includes a specification of all the operations provided by the tool in terms of their parameters and their types, defaults, etc., plus specifications of all the abstract objects manipulated by the tool in terms of their defining (and descriptive) sub-components. This representation of operations and objects follows Minsky's frames paradigm [7] in the spirit of KRL [1] or FRL [13]. The representation allows the Agent to follow the user's focus of attention down to arbitrarily deeply nested aspects of object or operation descriptions to resolve ambiguities or contradictions. This facility depends on the tool to provide resolution of object descriptions into sets of referents.

The tool description also specifies the syntax for the user's input descriptions of the objects and operations. The Agent applies the grammar thus specified to the user's input (as pre-processed by the Front-End) in a flexible way, providing the kinds of flexible parsing facilities mentioned above. The user may also request information about the tool or other help, and the Agent will attempt to answer the query with information extracted from the tool description, and displayed according to tool-independent rules.

Besides requesting that the Front-End output text strings to the user, the Agent may also specify instances of system objects. The Front-End will then display the objects according to a display format specified in the tool description. For the Front-End we are using, which operates through a graphics display equipped with a pointing device, this allows the user to refer directly to system objects by pointing. The Front-End reports such pointing events to the Agent in terms of the system object refered to. Typed input is pre-processed into a stream of lexical items, and other pointing events, such as to menus, can also be reported as lexical items. We are also experimenting with a limited-vocabulary, single word (or phrase) speech recognizer, isolated in the Front-End. It's output can also be reported as a lexical item.

This concludes the overview of the system structure. For the remainder of the paper, we will concentrate on the representation employed in the tool-description, and the way the information, thus represented, is used by the remainder of the system. Our examples will be in terms of the tool being used as a test-bed for the development of the Agent and Front-End: a multi-media message system, capable of transmitting, receiving, filing, and retrieving pieces of electronic mail whose bodies contain mixtures of text, speech, graphics, and fax.

3. Representation of task specific information

A functional sub-system, or *tool*, is characterized for the user interface program by a data base which describes the **objects** it manipulates and the **operations** it can perform. This *tool description* is a static information structure (provided by the sub-system implementor) which specifies everything that the User

Agent needs to know about the tool. We'll first give a brief overview of the structure of the tool description and how the Agent uses it to provide an interface to the sub-system. Then the content of the data base will be explained in more detail, with examples showing how this information is utilized in the processing of commands from the human user. The tool description consists of:

- **Declarations of the data objects used by the tool.**
  These declarations specify the internal structure of each object type defined within a particular sub-system. The tool may also contain references to object types that are defined in a global data base and are used by many different tools (e.g. files, user names, dates and times). The object declaration provides rules for displaying an instance of the object, syntax for descriptions of it in commands, and documentation that can be used to explain its function to the user.

- **Descriptions of the operations which the tool can perform.**
  Each operation entry specifies the parameters that the Agent must provide to the tool to invoke that action. It also defines the legal syntax for the command, provides some simple measures of its cost and reversability, and supplies a text explanation of its purpose.

As mentioned earlier, the primary goal of the Agent is to help the human user to specify sub-system operations to be executed. To carry out this function, it parses the user's commands (including text, pointing, and possibly spoken input) according to the syntax specifications in the tool description. It decides which operation has been selected and attempts to fill out the parameter template associated with it. This process may involve interpreting descriptions of sub-system objects, negotiating with the user about errors or ambiguities that are discovered, and explaining the meaning of command options.

3.1. Object Descriptions

The tool description contains a declaration of each data type that is defined within that sub-system. Data objects which will be manipulated by both the Agent and the tool are represented as lists and property lists (sets of name-value pairs), using the formalism defined by Postel for the communication of Internet messages [9]. This representation is self-describing in that the structure and type of each data element is represented explicity in the object. Thus, complexly structured objects can be transferred between the User Agent and the tool, and the Agent can interpret them according to the information contained in the tool description. For example, the following is the internal representation of a simple message (primitive elements are integers or text strings, and brackets are used to delimit sets of name-value pairs):

```
[ StructureType: ObjectInstance   ObjectName: Message
  Sender:
  [ PersonName: [ First: John  Middle: Eugene  Last: Ball]
    Host: [ Site: CMU   Machine: A ]
  ]
  Recipient:
  [ PersonName: [ First: Phil   Last: Hayes ]
    Host: [ Site: CMU   Machine: A ]
  ]
  Copies: []
  Date: [ Year:1980 Month:April Day:10 Weekday:Thursday
          AMPM: AM   Hour: 11   Minutes: 16   Seconds: 37]
  Subject: "Meeting tomorrow?"
  Body: "Phil, Could we meet tommorrow at 1pm? -Gene"
]
```

The structure of a **Message** is defined in the tool data base by a message *schema*. This schema declares each legal field in the object and its type, which may be a primitive type like TEXT or an object type defined by another schema in the tool description. The schema also specifies the number of values that each field may have, and may declare default values for new instances of the object. The following is a simplified schema for a message object and some of its components:

```
[ StructureType: ObjectSchema   ObjectName: Message
  DescriptionEvaluation: ToolKnows
  Schema:
  [ Sender: [ FillerType: Mailbox ]
    Recipient: [ FillerType: Mailbox  Number: OneOrMore ]
    Copies: [ FillerType: Mailbox   Number: NoneOrMore ]
    Date: [ FillerType: Date ]
    Subject: [ FillerType: MultiMediaDocument ]
    Body: [ FillerType: MultiMediaDocument ]
    After: [ FillerType: Date   UseAs: DescriptionOnly ]
    Before: [ FillerType: Date   UseAs: DescriptionOnly ]
  ]]

[ StructureType: ObjectSchema   ObjectName: Mailbox
  DescriptionEvaluation: ToolKnows
  Schema:
  [ PersonName: [ FillerType: PersonName ]
    Host: [ FillerType: Host ]
  ]]

[ StructureType: ObjectSchema   ObjectName: PersonName
  DescriptionEvaluation: OpenEnded
  Schema:
  [ First: [ FillerType: TEXT ]
    Middle: [ FillerType: TEXT   Number: NoneOrMore ]
    Last: [ FillerType: TEXT ]
  ]]

[ StructureType: ObjectSchema   ObjectName: Host
  DescriptionEvaluation: ToolKnows
  Schema:
  [ Site: [ FillerType: TEXT Default: CMU ]
    Machine: [ FillerType: TEXT Default: A ]
  ]]
```

In addition to defining the structure of an *instance* of a message object, the schema includes fields which are used by the Agent to interpret descriptions of messages. The **DescriptionEvaluation** field tells the Agent how to evaluate a description of an object in this class. For example, **ToolKnows** indicates that the sub-system is prepared to evaluate a *description structure* and return a list of instances matching the description. A description structure is an instance of the object with special wild card values for some fields and with possible extra **DescriptionOnly** entries, such as the **After** field in the example above. Since a description of one object may reference other objects ("the messages from members of the GI project"), the Agent uses the hierarchy defined by the object declarations to guide its evaluation of the user's commands. Each level generates a new sub-task in the Agent which processes that portion of the description, and is responsible for resolving ambiguities that it may encounter. This structure also makes it possible to follow the user's focus of attention, since new input may apply to any of the currently active subgoals ("No, only the ones since October" or "No, only the ones at ISI").

Each object declaration also includes information which is used by the Front-End module to display instances of that object type. Several different formats may be defined; the tool (or Agent) selects an appropriate format by name each time it displays an object. The format declaration specifies which fields of the object to display and their layout; it may also provide parameters to the Front-End which invoke special capabilities (highlighting, font selection) of the display hardware. For example, the following section of the description for a **Message** defines two different styles of message header display.

```
DisplayFormat:
[  ShortHeader:     // style: From Hayes on 18-Mar
    [ Text: "From &Sndr& on &Day&"
      Sndr: [Field: Sender/PersonName/Last FaceCode: Bold]
      Day: [ Field: Date   Style: ShortDay ]
    ]
   FullHeader:      // From Eugene Ball on 10-Apr-80 11:16am
        about 'Meeting tommorrow?'
    [  Text: "From &Sndr& on &Day& about '&Subj&'"
       Sndr: [ Field: Sender/PersonName   Style: FullName ]
       Day: [ Field: Date   Style: FullDate ]
       Subj: [ Field: Subject   Style: FullSubj ]
   ]]
```

Each object declaration also defines the legal syntax that can be used in commands that refer to objects of that type. In order to understand a phrase like "the messages from Phil since March", the Agent uses the syntax definition associated with the object type **Message**, which in turn refers to the syntax for other objects like **Date** and **PersonName**. In the example below, question marks indicate optional syntactic elements, asterisks mark fields that can be repeated, slashes indicate word class identifiers, and ampersands mark places where the syntax for other object types is to be expanded. Some syntactic entries also specify correspondences to particular fields in the object; as a command is parsed, the Agent builds *description structures* which represent the objects referred to in the command. Thus, the phrase "since March" results in an appropriate **After** clause in the **Message** description. The grammar defined by these syntax entries is applied to the user's input in a flexible way [2], so that grammatical deviations such as misspellings, words run together, fragmentary input, etc. can still be parsed correctly.

```
Syntax:
[
   Pattern: (?/Determiner /MessageHead */MessageCase)
   Determiner: (the (all ?of ?the) every)
   MessageHead: (messages notes letters mail)
   MessageCase:
   ( [ Syntax: (/From &Mailbox)
       StructureToAdd: [ Sender: &Mailbox ]]
     [ Syntax: (/Before &Date)
       StructureToAdd: [ Before: &Date ]]
     [ Syntax: (/After &Date)
       StructureToAdd: [ After: &Date ]]
   )
   From: (from (arriving from) (that came from) (/Mailed by))
   Mailed: (mailed sent delivered)
   Before: (before (dated before) (that arrived before))
   After: (after since (dated after))
]
```

Finally, the object description provides information which is used to automatically construct documentation and provide answers to user requests for help. Each object contains a brief text explanation of its structure and purpose in the sub-system, which can be presented to the user in response to a request for information. The documentation is also placed into a Zog [14] information network at a node representing the data type, which is connected to Zog frames for other objects referenced by that type. The user can find information about a related sub-system object by choosing a link to follow (with a menu selection); that frame is quickly displayed. The legal syntax for descriptions of the object and links to frames for operations which manipulate it are also included in the Zog network. In the following example documentation frame for **Message**, the italicized entries are buttons which can be activated to request more information about a specific topic:

> <u>Message</u>: (Multi-Media Message System)
>
> Each message contains the author and date of origination and is addressed to (one or more) destination mailboxes; copies may optionally be sent to additional destinations. The body of a message may contain uninterpreted text, images, sketches, and voice recordings.
>
> Example syntax:
> 'messages [from **Person**] [before/after **Date**]
> [about **String**]'
>
> *Detailed syntax*

Related object types:	Operations:
Mailbox	*Send Date*
Display Multi Media Document	*Edit*

3.2. Operation Descriptions

Each sub-system operation which can be invoked by the Agent is also described by an entry in the tool data base. An operation entry specifies the parameters that the Agent must provide to the tool to have it perform that action. The object type of each parameter is declared and the tool description can optionally indicate that a parameter position may be filled by a set of such objects. In addition, constraints on the legal values of a parameter are sometimes provided, which can help the Agent to avoid requesting an illegal operation.

```
[  StructureType: Operation   OperationName: Forward
   Reversible: false
   Cost: moderate
   Parameters:
   [  Message: [ FillerType: Message  Number: OneOrMore ]
      Recipient: [ FillerType: Mailbox  Number: OneOrMore ]
      Forwarder: [ FillerType: Mailbox  MustBe: CurrentUser ]
   ]
   Syntax:
   [  Pattern: (/Forward %Message to %Recipient)
      Forward: (forward send mail (pass on) deliver redeliver)
   ]
   Explanation: "Message Forwarding
A copy of a message that was delivered to you can be sent to
another person with the Forward command. You must specify
the message to forward and the destination mailbox. Sample
syntax: 'Forward the last message from Phil to Adams at ISIE'"
]
```

The example entry for the **forward** operation also includes a declaration of the legal syntax for the command, and a text entry which will be included in its documentation frame. It also indicates that this command is not reversible (once executed it cannot be undone), and that it is moderately expensive to execute. This information is used by the Agent to select an appropriate style of interaction with the User; for example, irreversible operations will usually require explicit confirmation before the request is given to the sub-system for execution.

4. Conclusion

The design and implementation of a good user interface for a computer sub-system is a difficult and time-consuming task; as new techniques for communication with computers (especially high resolution displays and speech) gain widespread use, we expect this task to become even more expensive. However, we also feel that the typical user interface must be made much more robust, graceful, and intelligent. For this goal to be feasible, substantial portions of the interaction with the user must be independent of the details of the application, so that the development cost of the user interface code can be shared by many sub-systems.

Therefore, we are designing a generalized User Agent which can be used to control a variety of different sub-systems. The Agent carries on a dialog with the human user; it can understand a variety of different command styles, recognize and correct minor syntactic or spelling errors, supply default values for command arguments based on context, and provide explanations when requested. All of the information that the Agent needs to know about the application system is explicitly stored in a *tool description* provided by the sub-system implementor. This paper has concentrated on the content of that data base, detailing the information represented there and demonstrating how the Agent can apply it to provide a sophisticated interface to a specific application system.

The tool description is represented in a unified formalism, which enables us to maintain a single data base which specifies all of the task-specific attributes of a particular sub-system. Because the information is stored in a single format, it can easily be utilized by multiple portions of the interface system. For example, a single syntax description is used to parse user commands, to generate explanations of system actions, and to construct documentation of the options available in the tool.

The initial implementation of the Agent will provide the user interface for the Multi-Media Message System; an electronic mail facility which manipulates messages containing mixtures of text, recorded speech, graphics, and images. The system is being implemented as a multiple machine, multiple language distributed system: screen management (multiple windows) and graphics support are provided in Bcpl [11] on a Xerox Alto [15] with a high resolution raster display and pointing device; audio recording and playback is controlled by a DEC PDP-11 in C [5]; the Front-End module and User Agent are implemented in C and LISP respectively, on a VAX-11/780 running Unix [12]; and the tool (message system) runs in C on the VAX. The system modules communicate using a message based Inter-Process Communication facility [10] within Unix, and a packet broadcast network (Xerox Ethernet [6]) between machines. Most of the system components are currently running as individual modules, the first version of a single integrated system should be completed by June 1980. Because of our goal of a smoothly working, robust, and graceful system, we expect to continue tuning and improving the implementation for at least another year. The system will eventually be moved to a single powerful personal computer, where we expect it to make substantial contributions to the CMU Spice (Scientifc Personal Integrated Computing Environment [8]) development effort.

References

1. Bobrow, D. G. and Winograd, T. "An Overview of KRL-0, a Knowledge Representation Language." *Cognitive Science 1*, 1 (1977).

2. Hayes, P. J. and Mouradian, G. V. Flexible Parsing. Proc. of 18th Annual Meeting of the Assoc. for Comput. Ling., Philadelphia, June, 1980.

3. Hayes, P. J., and Reddy, R. Graceful Interaction in Man-Machine Communication. Proc. Sixth Int. Jt. Conf. on Artificial Intelligence, Tokyo, 1979, pp. 372-374.

4. Hayes, P. J., and Reddy, R. An Anatomy of Graceful Interaction in Man-Machine Communication. Tech. report, Computer Science Department, Carnegie-Mellon University, 1979.

5. Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* Prentice-Hall, Inc., 1978.

6. Metcalf, Robert and Boggs, David. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Comm. ACM 19*, 7 (July 1976), 395-404.

7. Minsky, M. A Framework for Representing Knowledge. In Winston, P., Ed., *The Psychology of Computer Vision*, McGraw Hill, 1975, pp. 211-277.

8. Newell, A., Fahlman, S., and Sproull, R.F. Proposal for a joint effort in personal scientific computing. Tech. Rept. , Computer Science Department, Carnegie-Mellon University, August, 1979.

9. Postel, J. Internet Message Protocol. Draft Internet Experiment Note , Information Sciences Institute, Univ. of Southern California, April, 1980.

10. Rashid, R. A proposed DARPA standard inter-process communication facility for UNIX version seven. Tech. Rept. , Computer Science Department, Carnegie-Mellon University, February, 1980.

11. Richards, M. BCPL: A tool for compiler writing and systems programming. Proceedings of the Spring Joint Computer Conference, AFIPS, May, 1969, pp. 34:557-566.

12. Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Comm. ACM 17*, 7 (July 1974), 365-375.

13. Roberts, R. B. and Goldstein, I. P. The FRL Manual. A. I. Memo 409, MIT AI Lab, Cambridge, Mass., 1977.

14. Robertson, G., Newell, A., and Ramakrishna, K. ZOG: A Man-Machine Communication Philosophy. Tech. Rept. , Carnegie-Mellon University Computer Science Department, August, 1977.

15. Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. Alto: A personal computer. In *Computer Structures: Readings and Examples*, McGraw-Hill, 1980. Edited by D. Siewiorek, C.G. Bell, and A. Newell, second edition, in press.