Int. J. Man–Machine Studies (1983) **19**, 285–306

Graceful interaction through the COUSIN command interface

PHILIP J. HAYES AND PEDRO A. SZEKELY

Computer Science Department, Carnegie–Mellon University, Pittsburgh, Pennsylvania, U.S.A.

(Received 8 April 1982, and in revised form 21 January 1983)

Currently available interactive command interfaces often fail to provide adequate error correction or on-line help facilities, leading to the perception of an unfriendly interface and consequent frustration and reduced productivity on the part of the user. The COUSIN project of Carnegie–Mellon University is developing command interfaces which appear more friendly and supportive to their users, using a form-based model of communication, and incorporating error correction and on-line help. Because of the time and effort involved in constructing truly user-friendly interfaces, we are working on interface system designed to provide interfaces to many different application systems, as opposed to separate interfaces to individual applications. A COUSIN interface system gets the information it needs to provide these services for a given application from a declarative description of that application's communication needs.

1. Introduction

Many of today's interactive interfaces to computer systems are sources of great frustration to their users. The simplest error or incompleteness in a command to such a system is likely to elicit a more or less informative error message and a request to try again. Different parts of the same interface may use quite different syntax or conventions for essentially similar functions. The on-line help, if it exists, may come in chunks too big to be useful for interactive use, and may be indexed and cross-referenced inadequately to permit easy location of the information desired. These and other problems with interactive interfaces have been discussed at length by numerous authors including Hansen (1971), Miller (1968) and Hayes, Ball & Reddy (1981). In the COoperative USer INterface) project at Carnigie–Mellon University, we are working towards user interfaces that appear more friendly and supportive to their users, and thus reduce frustration and enhance productivity.

While the COUSIN project is wide-ranging in its overall goals and scope,† the present paper is concerned with our work on user-friendly interactive command interfaces. In particular, it deals with a coarse-grained semantically-constrained style of command interaction in which the user repeatedly specifies a command together with a set of dependent semantically-typed parameters. This style of interaction typically arises at the top command level of an operating system (e.g. manipulating files, invoking application subsystems), and in interaction with some common applications (e.g. electronic mail manipulation, magnetic tape management). We are not

† It also covers the work on graceful interaction in natural language interaction by Hayes & Reddy (1983).

currently looking at the finer-grained kind of interaction that occurs, for instance, with a screen-oriented text editor. Nor are we concerned with support for naive users, assuming instead that users have a basic familiarity with computers and command interaction in general and the kinds of objects they are dealing with (files, directories, pieces of electronic mail) in particular. Familiarity with the specifics of individual commands, however, is not assumed.

Given these scope restrictions, our approach to interface design is based on three key concepts.

One interface for all applications. A single monolithic interface system provides interface services for a wide variety of different applications. The interface system is data-driven from declarative descriptions of the interface needs of the various applications. Given the large amount of time and effort needed to construct user-friendly interfaces, some method of sharing interface code across applications appears to be necessary if such interfaces are to be introduced widely. A single data-driven application independent interface achieves such sharing with maximum interface consistency across applications while also reducing implementation effort for individual applications.

Communication via forms. A user and an application program communicate indirectly by reading and updating fields in a form specific to that application, access to the form being controlled via the COUSIN interface system. Fields correspond to pieces of information that need to be communicated, such as the initial input parameters of a printing application (e.g. number of copies to be printed), or the output list of messages for an electronic mail application. This kind of form-based communication makes it straightforward to separate the specification of what information needs to pass between the user and application from the way in which the communication is realized, and so facilitates the construction of a data-driven interface system that can service many applications.

Intelligent support for form-filling. Form fields can have types and defaults, and the sympathetic enforcement of the type restrictions can provide a major contribution to user friendliness. The "files to print" field of the print application would be required to contain readable files, for instance, so misspellings or abbreviations can be checked and possibly corrected against the names of readable files that actually exist. Informing the user of what fields are available and what the types and defaults for these fields are also fulfills a major part of a user's need for on-line help.

After discussing and justifying these three features of our approach to command interaction in a little more detail, we will look at the practical realization of the ideas in an implemented interface system. We are currently working with two implementations:

COUSIN–Unix. An alternative shell (operating system command interface) for Unix, operating on standard terminals, and

COUSIN–SPICE. A command interface for the SPICE computing environment for the Perq, a powerful personal computer with bit-map display and pointing input.

An initial implementation of the former has been completed and is in limited use; some detailed examples of it in operation will be presented. The latter is in a much earlier stage of development and will not be discussed in any detail.

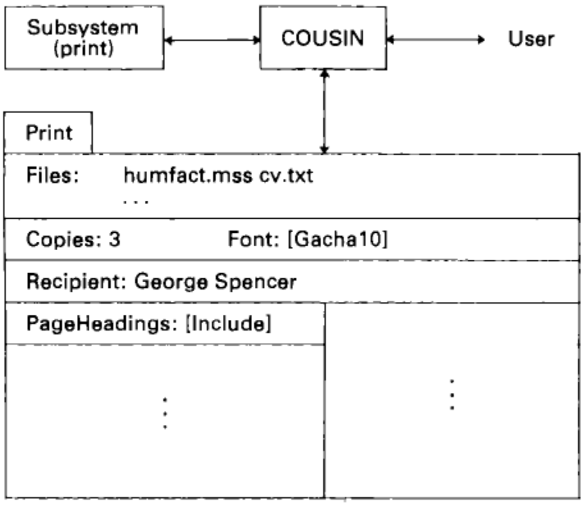

Fig. 1. Communication by form-filling.

2. Communication via forms

The notion of form-based communication lies at the heart of the COUSIN approach to command interaction. It is based on the view that the user and the application he wants to use have certain pieces of information that they wish to exchange one or more times during an interactive session: the input parameters (number of copies, files to print, font to use, etc.) for a print command, the output list of messages for an electronic mail application, the invocation of the delete command and the list of messages to be deleted for that same application. For a given application, each of these pieces of information is assigned a *field*, and the collection of these fields constitutes the *form* for that application. The lower part of Fig. 1 shows the form for a generic print application program. In this example, the fields all correspond to input parameters. Some fields have default values as indicated by the brackets for "Font" and "PageHeadings". Such defaults can be overwritten by the user on input fields as happened in this form instance for "copies" and "Recipient" (defaults 1 and "Self", respectively). Some input fields, "Files" in this example, have no default and must be specified by the user. Not shown in the diagram are the types associated with each field. The types can be of varying levels of specificity, ranging from "String" for "Recipient" through "Integer" for "Copies" and "ReadableFile" for "Files" to enumerated types for "Font" and "PageHeadings", the latter being an enumeration of size two.

As indicated by the upper part of Fig. 1, a user and an application program communicate indirectly by reading and updating fields of the form for that application with all access to the form controlled by the COUSIN interface system. This effectively decouples the application system from direct interaction with the user. The application need only specify via its form what information it wishes to have input and output, and COUSIN will manage the interaction that realizes that transfer of information to and from the user with all the user friendly support that COUSIN can incorporate, including enforcement of the field types on input fields through error-correcting dialogues with the user.

The decoupling of applications from direct contact with the user makes it feasible to provide interface services for a wide-range of applications through a single monolithic interface system such as COUSIN, and was the basic motivation in adopting the form-based model of communication. Since our goal is to construct practical interfaces with many sophisticated user-friendly features, we needed to adopt an approach that would allow us to share the considerable implementation time and effort necessary for such interface sophistication across many application systems. Any approach which did not allow such sharing would not allow user-friendly interfaces to be constructed on a routine basis. The use of a single data-driven interface system across a wide variety of application systems is a clean and attractive method of achieving the sharing we desire and is highly compatible with the form-based approach to communication. The notion of sharing an interface across a variety of applications was previously investigated by Lantz (1980), in the context of a distributed computing environment, and the work on COUSIN has benefitted significantly from his experience.

In addition to making user-friendly interfaces for a wide variety of applications a practical proposition, a data-driven application-independent interface based on communication via forms has several other advantages.

Reduced implementation effort. Since the application system has no direct interaction with the user, and since COUSIN ensures that field values are of appropriate types, the application system need not perform these interaction and checking tasks itself. In many cases, this represents a substantial savings in implementation effort over an interface built specially for the application, even if it is simple and relatively unfriendly.

Consistent interface. Because all interaction is conducted through COUSIN, responses to command errors, requests for missing information, etc., are uniform and consistent across all application systems.

Immediate availability of interface features. All the advanced interface features of COUSIN are immediately available for any application on construction of the appropriate blank form.

Test facility for interface features. Since any new user-friendly interface feature incorporated into COUSIN is immediately available with all applications which have forms, COUSIN can be used as a vehicle for experiments on new features of uncertain usefulness and for performance checking of variations on other interface features. These experiments can be made more useful by performing them across a wide range of applications without having to change the applications themselves [see Ball & Hayes (1982) for a more detailed discussion of the potential of COUSIN as a test-bed for interface features].

Error correction and abbreviated input. Since each form field has a type, COUSIN can detect and attempt to correct invalid values that the user might place in input fields through spelling correction against the appropriate list of correct values (e.g. the dynamically determined set of available files for file types, or the enumerated set for enumeration types). The type information can also be used to allow the user to fill the fields through the use of unique abbreviations and/or menu selection.

Interactive error resolution. When COUSIN's attempts to correct or resolve the abbreviation of a field value fail or when they generate several possible acceptable values, the problem can be resolved by interaction with the user based on the field's type and COUSIN's success in correction. The user's attention can also be drawn to fields that do not have defaults and for which the user has not supplied a value.

Adaptab:lity. The form based model of communication is a powerful metaphor that is readily adaptable to various kinds of I/O hardware. Forms can be displayed as attribute/value lists for alphanumeric terminals or in graphical format for a bit-map display. Also, forms can be updated by constructive command lines, by within-form editing, or by menu-selection (using either a screen pointing device or isolated phrase speech recognition).

Integral on-line help. The display of a form with mnemonically named fields is in itself a form of on-line help. Through it the user can determine what kinds of information can be communicated to the application (input fields) and what assumptions the application is currently making (defaults). This information can be supplemented by making the field type information available in response to a simple command applicable to all fields.

Automatically generated on-line documentation. The blank forms already contain most of the information a user is likely to wish to know about individual commands. When the basic information is supplemented by some sentences describing the purposes of commands and the fields of their forms, COUSIN can reformat the form automatically to provide on-line documentation. The resulting documentation is consistent and uniform in format, and is always up to date with changes in the application that are reflected in changes to its form [see Hayes (1982) for a detailed description of how this documentation is produced in the current COUSIN implementation].

So far, we have confined our attention largely to forms containing only input fields. This kind of form is suitable for the initial specification of the parameters of non-interactive applications, but is clearly less than sufficient for interactive applications. Nevertheless, form-based communication can be used with interactive applications. In fact, through observation of some currently available command interfaces we have identified three general styles of communication with applications that can be supported through a form-based approach.

Non-interactive. Parameters are specified, usually in a command line which is collected by a system command interpreter, before execution of the application begins. The application normally runs to completion after being invoked in this way.

Information collecting. The application may accept (or request) additional information after it gets control, either because necessary parameters were omitted in the initial command line or because a need for additional information is discovered after execution begins.

Command loop. After start-up, the application enters an interactive command loop: repeatedly accepting commands, executing them, and presenting the results to the user, who then composes his next request.

The corresponding scenarios for the form-based approach of COUSIN are as follows.

Non-interactive. This is the simplest case and the one we have been mostly dealing with so far. Form fields correspond directly to application parameters. The user invokes the application through a menu or a command line, which may specify values for some (or all) of the input fields for the application. COUSIN obtains the form for the application thus specified, and sets up its defaults. If a command line was used, COUSIN parses it and transfers the various parameter values thus obtained to the appropriate fields of the form. If after this, all parameter fields are correctly filled, COUSIN executes the application in the normal way. If, on the other hand, information is missing or incorrectly specified, COUSIN reports the problems to the user, and gives him an opportunity to correct the situation by editing the form. When both the user and COUSIN are satisfied with the way all the fields are filled, the user may start execution of the application explicitly. However, COUSIN will not allow him to start execution while problems remain with the form. If the user is unable to correct the form satisfactorily, he must either abort the attempt at application invocation or save the form in its current state for later correction.

Information collecting. This situation is similar to the previous one, except that COUSIN will start execution of an application with some of the required parameter fields unspecified, although they cannot, of course, be specified incorrectly. After the application is started, it can request the value of any field in its form. If a requested field is undefined, COUSIN will inform the user that a value is required and suspend execution of the application until the user specifies the required value which is then passed back to the application.

Using this type of interaction, an application can be started without fillers for any of the fields in its form being specified, and the user interface will prompt for whatever parameters are needed when they are first referenced. It is a good example of how COUSIN insulates the application from concerns about how and in what order its parameters are acquired, and yet can make the parameters available as they are required.

Command loop. The user specifies interactive commands to the application by inserting the name of a command into a field whose type is an enumeration of all the commands available; this insertion could be done by direct type in or by menu selection. Alternatively, there could be a boolean valued field for each possible command. In either case, the field used to communicate the command would have a special *active* status which means that a message† is sent to the application by COUSIN every time the field changes value, thus allowing the application to avoid inefficient polling of the field's value.

When not actually executing one of its own commands, the application would wait for notification that one of these *active* fields had been modified. Additional parameters for application commands can be specified through other fields in the form in the same way as the two previous cases. The display of the form can be organized in such a way that the correspondence between the command fields and the fields that act as their parameters is clear to the user. Facilities are also needed to allow the application to determine whether such parameter fields are up to date or are merely an inappropriate leftover from earlier invocations of subcommands.

In each of the above cases, results can be transmmitted back from the application to the user as the values of non-parameter fields reserved for that purpose, and modifiable only by the application. COUSIN will display these field values to the user.

† We are assuming that COUSIN and the application are separate processes and communicate via some kind of interprocess communication facility.

3. COUSIN–Unix

In the preceding section, we discussed form fields being inserted or edited by the user, and fields being displayed to the user, but we did not say how this would take place. We omitted these details because COUSIN is being implemented for two quite different hardware configurations, and the details are correspondingly different in each case. On the one hand, COUSIN is being implemented on a VAX-11/780 using a standard (24-line) display terminal for communication with the user. This version of COUSIN will provide an alternative to the well known Unix shell, the top-level command interpreter for the Unix operating system (Ritchie & Thompson, 1974), and so is called COUSIN–Unix. On the other hand, COUSIN is also being implemented on the Perq, a powerful personal computer, equipped with bit-map graphics display and pointing device. This version of COUSIN will provide a command interface to the SPICE computing environment (Newell, Fahlman & Sproull, 1979) also under development at Carnegie-Mellon University, and so is known as COUSIN–SPICE. The more powerful interface hardware available for COUSIN–SPICE allows a much richer set of graphically-based interface techniques. Multiple windows, for instance, can be used to maintain several different applications and their forms simultaneously, or the user can employ pointing devices to select field values for alteration, menu selection to choose a field value restricted to one element of a small set, comprehensive screen editing techniques, etc.

At the time of writing (December 1982), an initial implementation of COUSIN–Unix has been completed and is in limited use, while COUSIN–SPICE is at an earlier stage of development. We will therefore base our account in this section of how the general COUSIN approach can be realized in practice on the present COUSIN–Unix implementation, discussing first the modifications its circumstances have required to the general approach, and then presenting some detailed examples of it in operation.

3.1. ADAPTING COUSIN TO UNIX

Our decision to implement the ideas of COUSIN in the form of an alternative Unix shell (top-level operating system command interpreter) was motivated in several ways.

We wished to evaluate the COUSIN approach through an interface with a sufficiently large potential user community to make the evaluation meaningful. Unix has an extremely large user community, over 100 of whom can be found within our own Department, and since a significant segment of that community (see Norman, 1981, for example) believes the standard Unix shell to be not very user-friendly, there are grounds to expect that many Unix users would be willing to try out a different command interface.

We wished to compare two interfaces to the same system, one using the COUSIN approach and the other more representative of generally available command interfaces. The standard Unix shell fulfills the role of representative command interface.

We wished to test the adaptability of the COUSIN form-based model of communication. The standard Unix shell has a style of interaction based on command lines which does not ostensibly follow the form-based model. This style of interaction can be very terse and efficient providing the user does not make errors or lack knowledge. This efficiency made it desirable to keep command line interaction available through

COUSIN–Unix, but to make it more user-friendly and to integrate it with the form-based model, thus testing the model's adaptability.

Given these motivations for choosing to try out the ideas behind COUSIN in the context of a Unix shell, we designed and implemented an interface with the following major components.

Flexible command line parser. Given that one of our motivations was to test the adaptability of COUSIN's form based model of communication to interaction through command lines, it was necessary to provide a command line interpreter with at least the functionality of the standard Unix shell, except for the shell programming features (conditionals, iteration, etc.). But since we also wished to maintain the user-friendly character of COUSIN through the adaptation, it was necessary that the parser be flexible in the face of input errors, rather than simply rejecting incorrect or incomplete commands as in standard Unix.† Accordingly, the parser for COUSIN–Unix tries to correct out of order arguments, and typos or other misspellings in command names and option and argument markers; the parser also allows such markers to be given in whole word format as well as in the single character style normal in standard Unix. The output from the parser is the form for the command specified by the first token of the command line with the appropriate fields filled by the parameters extracted from the remainder of the command line. Error detection on command parameters is handled through the standard COUSIN form correction mechanism in which COUSIN attempts to use the field types of invalid fields to correct them into valid ones to the extent to which this is possible. Clearly, no correction is possible when the type is "arbitrary string", but spelling correction is used on enumerated types, including dynamically defined enumerations (e.g. names of files and directories). A successful correction attempt may produce a unique correction for the user to confirm or several possible corrections for the user to choose among.

Interactive form editor. This component allows the user to correct incorrect fields, fill unfilled fields, or in general modify the value of any field of a form, and thus provides a structured way to correct command line errors without having to type the line over again, or indeed, to specify the parameters to a command without using a command line. The form editor also includes a command to cycle the user through the incorrect and unspecified fields, listing any corrections COUSIN has come up with, along with information about what should fill the field. The same facility can be used to make changes to previously saved forms, either to make corrections that were not possible earlier or to adapt an earlier correct command to a different task.

Unix-like command loop. In order to obtain as direct a comparison as possible between COUSIN–Unix and the standard Unix shell, and also to minimize the start-up effort required for a Unix user to try COUSIN–Unix, we attempted to make the basic COUSIN-Unix command loop as similar to that of the standard Unix shell as possible for the interpretation of correct commands. It operates as follows:

1. The user types a command line in response to a prompt from COUSIN.

† Unix commands actually parse their own command lines, with the shell providing only some preprocessing, such as the file wildcard expansion, so different commands behave in different ways and it is hard to make general statements of this kind with complete accuracy, but the most common form of response to erroneous command lines is a one line usage summary followed by a return to the shell command level.

2. COUSIN identifies the application invoked by the command line, locates the blank COUSIN form for the application and preloads it with the appropriate defaults.
3. COUSIN parses any parameters to the application specified in the command line, using the flexible parser mentioned above and inserts the parameters into the appropriate form fields. The syntactic information required for this parsing is included with the blank form.
4. COUSIN checks the form for completeness (all fields have values), and correctness (all fields have values that satisfy their type restrictions).
5. If the form is correct and complete, COUSIN executes the command as specified by the form and loops to Step 1 by issuing another command prompt. Thus, if the user issues only correct and complete commands, the interaction will look just like interaction with the standard shell.
6. If the form is incorrect or incomplete, COUSIN enters the interactive form editor to help the user correct the errors. The user may also specify that he wishes to enter the form editor anyway, even if the form specified by the command line is complete and correct.
7. After the user has completed or corrected all empty or incorrect fields, and is satisfied with the values of all other fields, he may tell COUSIN to execute the form, and this execution happens in exactly the same way as it would if the user had specified the current form field values through a command line, with control returning to Step 1 and the issuing of a command prompt.
8. The user may also return to Step 1 by discarding the current form or by saving it for future reuse with all existing values maintained. Saving a form is useful for constructing personalized commands with parameter defaults different from the standard (the names of saved forms can be used just like command names in command lines), and for temporarily saving commands that cannot be executed because of some circumstances that cannot be remedied from within the form editor (e.g. the user does not have appropriate access rights for a file).

In terms of the classification of styles of communication with applications presented earlier, this command loop presumes all applications are non-interactive, at least as far as the services of COUSIN–Unix go, i.e. they expect all their parameters specified in advance and run to completion once started with no further interaction with the user through COUSIN–Unix. In the context of Unix, this is not an unreasonable assumption, since it is the only style of interaction of the three we discussed that does not require any direct communication between the application and COUSIN, and hence does not require any modification to pre-existing Unix application systems. COUSIN–Unix executes such pre-existing Unix commands by translating the form back into a command string and executing that. The difference being that after it has passed through COUSIN, it is known to be correct. If the application needs to communicate further with the user after it has started execution, COUSIN–Unix provides the same character stream oriented style of communication as the standard Unix shell. We anticipate that we will eventually modify some interactive applications to operate through their forms, thus extending the COUSIN services to those interactions as well.

Within-line input editor. All input by the user is through a single line screen-oriented editor which allows the user to insert and delete words and characters at positions other than the end of a given line. In conjuction with a facility by which the user can get back the input line he just typed, this editor provides the user an alternative and sometimes more convenient way of correcting command lines. The editor syntax is modeless and is derived from the Emacs whole screen editor (Stallman, 1981).

Screen management. While both the basic command loop and the form editor operate in a scrolling, line-oriented mode, the current version of COUSIN–Unix does provide a limited amount of screen management, maintaining a two-line mode window at the bottom of the screen in which information about the current state of the interaction and the interface's expectations for the user's next input is displayed. A pop-up window in the top half of the screen is also provided for the on-line help facility described below. A completely screen-oriented version of COUSIN–Unix, including screen-oriented form editing is currently under development as described in the concluding section of this paper.

On-line help and explanation facility. At any point in the interaction, the user may obtain on-line help either on a specified topic or, if no topic is specified, general information relevant to the current state of the interaction. This help is displayed in a window that springs into existence in the top half of the screen when the user asks for help, and disappears after the user exits help mode. To be useful, on-line help text must be available in chunks that are not too large (otherwise there is too much to read), and adequately cross-referenced and indexed (otherwise the relevant information cannot be found). To avoid these problems for COUSIN, we have adopted some ideas from the ZOG (Robertson, Newell & Ramakrishna, 1977) rapid menu-selection system. Like ZOG, the COUSIN help facility consists of text segments or *frames*, none larger than half a display screen, structured into a network by semantically motivated links, one or more leading from each frame to other frames containing related material. Traversing one of these links causes the current frame to be replaced with the frame pointed to by the link. Unlike ZOG, there are two types of frames.

Static frames constitute the vast bulk of the frames and describe aspects of the system being interfaced to that do not normally change within the course of a single interactive session. These include the commands available, the parameters they take, the objects they manipulate, and the syntax used to describe these things. These frames are presented in response to requests for help on specific topics.

Dynamic frames are constructed on the fly in response to non-specific requests for help, and describe the current state of the system, how it came to be in that state, what COUSIN expects the user to do next, what the user's options for action are, etc. These dynamic frames also contain links to frames in the static network that contain descriptions relevant to the current command context.

A more complete description of the COUSIN help system can be found in Hayes (1982), along with an account of how most of the static network can be generated automatically from the information about command line syntax and parameter types and defaults in the blank forms that COUSIN already needs to perform its flexible parsing and interactive error resolution functions.

History mechanism. This component keeps a record of commands already executed, can print out a list of them on demand, and can retrieve specified commands for re-execution either exactly as before, or if the user employs the within-line editor, in a modified form.

Transcript facility. To facilitate its evaluation, COUSIN–Unix can record a complete time-stamped transcript of any interactive session conducted through it. The example interactions with the system given later in this paper were recorded via this facility. (The time-stamps have been edited out.)

To make it clearer how these various components of COUSIN–Unix benefit the user, some examples of actual interactions with COUSIN–Unix are appropriate. But first, a short digression to describe the command language handled by the flexible command parser of COUSIN–Unix is in order.

3.2. COMMAND LANGUAGE FOR COUSIN–UNIX

The command language for COUSIN–Unix is the same as that used by the standard Unix shell (Ritchie & Thompson, 1974), minus the constructions at a level higher than single commands, but supplemented by other language features that make it easier for the user to specify commands. Speaking approximately to avoid complication irrelevant to the present purpose, the standard Unix format for command lines is the command name, followed by a sequence of option flags and markers (single characters preceded by dashes), followed by a fixed-order sequence of non-optional arguments. An example is:

```
dover -r -c 3 -l foo.txt fum.doc
```

This is a call to dover,[†] an application program local to Carnegie–Mellon, which prints files on a Xerox Dover laser printer. Three options are specified: "-r", print 90 degrees rotated, i.e. with lines parallel to the long side of the paper; "-c", print the number of copies specified by the immediately following input token, in this case three; and "-l", lineprinter mode—no headings and 66 lines per page. The options are followed by dover's single non-optional argument, a list of files to be printed, in this case foo.txt and fum.doc. When a command has more than one non-optional argument, which input tokens are assigned to which argument is specified strictly by the position of the tokens in the input line. An example is the command, "cp", which copies a list of files, its first argument, into a directory, its second argument, as in:

```
cp file1 file2 dir
```

The standard Unix conventions are extended by COUSIN in two major ways: the addition of explicit markers for command arguments as a supplement to the present system of purely positional specification, and the addition of full word flags and markers for options as a supplement to the present system of single characters preceded by dashes. So the above examples could be written for instance as:

```
dover foo.txt fum.doc rotated copies 2 lineprintermode
cp onto dir from file1 file2
```

[†] For reasons too obscure to relate here, the command is actually called "cz", but we have named the form "doverprint", and through the command synonym facility of COUSIN–Unix may refer to it by any of several names including "cz" and "dover", the latter being treated as an abbreviation of "doverprint".

Note that when whole-word markers are used order can be relaxed, and that when only one argument remains unmarked it may appear anywhere in the command line.

The language recognized by COUSIN–Unix is also extended implicitly by the flexible, error-correcting parsing techniques employed. The following deviations are handled:

out of order arguments—to avoid ambiguity, the arguments must be distinguished either explicitly by markers or implicitly by type;

garbled arguments or spurious interjections—these are saved on a CouldNotRecognize list, and

misspellings of command names, option and argument markers and flags, and as far as possible the actual values of arguments and options. In the case of argument and option values, correction is based on the type of the form field in which the parameter is to be inserted, and is currently implemented only for enumerated types, including file and directory names which are considered dynamically defined enumerations.

In addition to the syntax for individual commands described above, the standard Unix shell also supports syntax for combinations of commands, including pipelines, conditional execution, and iteration. Of these, COUSIN-Unix currently supports only pipelining (including input/output redirection). File wildcarding is also supported exactly as in the regular shell, but we have not yet tried to combine wildcarding with spelling correction.

3.3. EXAMPLE INTERACTIONS WITH COUSIN-UNIX

As mentioned earlier, COUSIN–Unix operates through a standard (24-line) display terminal. The type of terminal actually used also allows some simple screen management. In particular, the current implementation of COUSIN–Unix divides the screen into the three independent windows shown in Fig. 2.

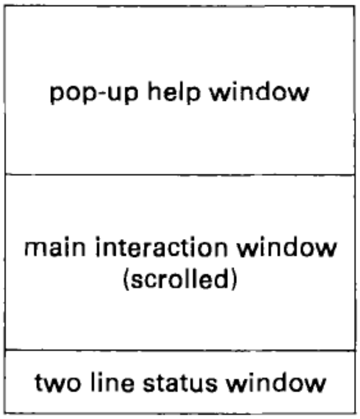

FIG. 2. COUSIN–Unix screen organization.

Main interaction window. A scrolled window through which all command line and form editor interaction takes place; it occupies those parts of the screen not occupied by the other two windows.

Mode window. A two-line window at the bottom of the screen, containing continually updated information about what mode COUSIN–Unix is in, what it is expecting the user to do next, how to abandon what is being done, and how to ask for more extensive help.

Help window. A window that springs into existence in the upper part of the screen when help is requested. It displays nodes from the highly connected network of finely-chunked text frames that constitutes COUSIN's on-line help facility. If no topic is specified, a frame containing pointers to information likely to be helpful in the current state of interaction is generated.

In the example that follows, we will be concentrating our attention on the main interaction window, with occasional comments about the other two windows as appropriate.

When COUSIN–Unix starts up, the user is presented with a command prompt "1": in the main interaction window, and a message in the status window informing him that the system is ready to accept a Unix command. The number in the prompt is related to the history mechanism provided by COUSIN–Unix, of which more later. If the user types a correct command, it will be executed as in:

```
   1: ls       ["ls" is the Unix command to print the contents of the current directory]
differences     humfact.aux     ijmms.trans     save/           umist.mss
foo.press       humfact.mss     outline         umist.aux       umist.press
   2: dover rota -c 2 humfact.mss
[3 pages * 2 copies⇒7 sheets] [output from the dover program]
   3:
```

Here, user input is in bold italics, and comments in ordinary italics. Note the mixing of Unix style markers with the extended COUSIN–Unix syntax, and the initial abbreviation of the keyword, "rotated", in the second command. The status line remains unchanged during the entering and execution of both commands. In terms of COUSIN's underlying operation, performing the second (dover) command involved finding the blank form for dover, filling in whatever defaults were specified, filling the appropriate fields in the form from the command line using the associated syntatic information, checking that the form thus obtained is correct and complete, and finally executing the form in the way described earlier.

Suppose now that the user had made a couple of errors in the last interaction, misspelling the filename and the abbreviation for "rotated".

```
3: dov roat -c 2 humfat.mss
roat⇒rotated
Editing form for doverprint
incorrect field(s):
   filestoprint: humfat.mss (not a readable file) (⇒ humfact.mss)
with the above corrections the form can be executed.
```

```
FormEd [go]: go
[3 pages * 2 copies⇒7 sheets]
  4:
```

For both of these errors, COUSIN is able to make unique corrections, of which it informs the user ("filestoprint" is the name of the field in dover's form that specifies what files are to be printed). The spelling corrector used by COUSIN is a simple one that is capable of correcting exactly one error (character transposition, insertion, substitution, or deletion) per word.† In this example, the corrections it is able to make result in a correct and complete form for dover which could be executed in just the same way as the one derived from the previous totally correct command. However, since what the user typed has been altered, COUSIN does not go ahead with the execution, but switches to form editor mode to allow the user to make further changes or abort the command. The mode shift is indicated by the "FormEd:" prompt, and also in the mode window. Anticipating the most likely response, COUSIN also gives "go" as the default, indicated by brackets, associated with that prompt. "Go" is one of the standard form editor commands, and it means that COUSIN should initiate execution of the form currently being edited. At this point, if the user types a carriage-return, the default will be entered, the go command will be interpreted, and the dover form executed. On the other hand, the user can type any other form editor command to display the current form, reject the correction, alter any other fields he wishes to, save the form for future reuse, etc. In the event, he is happy with the correction, so he types a carriage-return, COUSIN echoes the default (note the second "go" is not in bold face), executes the form, and when it is finished, switches back to command mode with appropriate changes to the mode window.

Now for an example where a unique correction is impossible.

```
4: dover align sideways /urs/ppjh/papes/umf
  Editing form for doverprint
  incorrect field(s):
    alignment: sideways (invalid selection)
    filestoprint:/urs/ppjh/papes/umf (not a readable file)
                                        ⇒ /usr/pjh/papers/humfact.mss | . . .)
FormEd [correct]: correct
  Correcting field alignment
    alignment: sideways (invalid selection)
    selections:
      vertical       horizontal
alignment [vertical]: sideways        [initial string]
alignment [vertical]: h
    alignment: horizontal
  Correcting field filestoprint
    filestoprint: /urs/ppjh/papes/umf (not a readable file)
        Possible correction(s): /usr/pjh/papers/humfact.mss |
```

† Experiments with a very similar spelling corrector (Durham, Lamb & Saxe, 1983) show that this level of correction is sufficient for the vast majority of spelling errors, and a corrector capable of correcting more complex errors might incur unacceptable performance penalties.

```
        /usr/pjh/papers umist.press | /usr/pjh/papers/umist.mss |
        /usr/pjh/papers/umist.aux | /usr/pjh/papers/humfact.aux
filestoprint: /usr/pjh/papers/          [initial string]
filestoprint: /usr/pjh/papers/h
     filestoprint: /usr/pjh/papers/h (not a readable file)
          Possible correction(s): /usr/pjh/papers/humfact.mss|
        /usr/pjh/papers/humfact.aux
filestoprint: /usr/pjh/papers/humfact.          [initial string]
filestroprint: /usr/pjh/papers/humfact.m
     filestoprint: /usr/pjh/papers/humfact.mss
FormEd [go]: go
[3 pages * 2 copies⇒7 sheets]
  5:
```

Here the user has incorrectly used the word "sideways" instead of "horizontal" for the alignment of printing on the page, and in trying to use an absolute Unix file specification, rather than as before one relative to the current directory (which is /usr/pjh/papers), has misspelt the name in a way that does not have a unique correction.† COUSIN informs the user of these problems, and places him in form editor mode with a default command of "correct" rather than "go"; "go" would not work here even if the user typed it explicitly. The default "correct" command, which the user accepts, cycles through any incorrect or empty fields in the current form, and helps the user to correct each problem individually. In this case, it tackles the alignment field first. Because it is an enumerated type, COUSIN lists out the possible fillers, then prompts the user with the name of the field, giving the default value of the field as the default input, and the value entered as an initial string that the user can edit if he wishes, using the within-line character editor through which all input to COUSIN takes place; The rationale for giving the user the initial string to edit is that the user might have misspelt the value in a way that the spelling corrector cannot deal with, and may find it easier to line edit it rather than retype it; if he wishes to start again as in this example, a single keystroke empties the line. In the event, the user simply cancels the initial string and types "h" followed by a carriage return.‡ Since "h" is a unique initial substring of one of the values for "alignment", that value is inserted in the field and "correct" goes onto the second problem. Here there is no fixed set of possible values for the field, but COUSIN has found several spelling corrections for the value entered and these are listed; note that there is an error in each element of the full file specification, but that the spelling correction can still cope because it resolves each element separately. In this case, the initial string provided by COUSIN is the common initial substring of the several corrections; the user extends it by one character, cutting down the number of possibilities to two, at which point another character is enough to resolve the ambiguity uniquely. A variation on the "correct"

† The Unix file system is structured as a tree of directories, and a full file specification involves the names of the directories on the path from the root directory to the one containing the file in question in addition to the name of the file, the names of the directories and file are separated by "/" and an initial "/" indicates an absolute specification.

‡ In the actual interaction, this happens on one line rather than two; the repetition is provided by the transcript generator to show the before and after state of any input line for which COUSIN provides an initial string to edit.

command, not illustrated here, arises with fields, such as "filestoprint", which can have more than one filler. If such a field has several fillers and not all are correct, the "correct" command cycles through each incorrect value individually, allowing the user to correct each one independently of the others. Alternatively, the user can line-edit the value of such fields as a single string which is then reinterpreted into a set of separate fillers.

COUSIN–Unix in general, and its form editor in particular, also provide support for pipes and the redirection of standard input and output.† The command line syntax for these features is exactly the same as for the standard shell. In terms of forms, the features are supported by giving each form two extra fields called "StandardInput" and "StandardOutput" which may be filled by file names (for I/O redirection) or by pointers to other forms (for pipes), a pipeline being represented by a sequence of forms in which adjacent elements point at each other through these fields. In cases of error or any other use of the form editor, the user edits one form at a time in the manner illustrated above, and may switch between the forms in a pipeline by means of the "next", "previous", "first", and "last" commands built into the form editor. The "correct" command switches forms automatically to get to the next error in the pipeline.

Sometimes, the form-orientated method of error correction is quite incovenient, and COUSIN–Unix provides an alternative line-oriented method as shown in the next example.

```
   5: dover rotatedfor campbell outl
      Editing form for doverprint
      incorrect field(s):
         filestoprint: outline
                       rotatedfor (not a readable file)
                       campbell (not a readable file)
FormEd [correct]: lined
   5: dover rotatedfor campbell outl          [initial string]
   5: dover rotated for campbell outl
   6:
```

Here the user has missed out the space between two words and instead of "campbell" going into the "recipient" field, it goes into "filestoprint", along with the incorrectly tokenized "rotatedfor". Clearly, to change this in a form-orientated way would be quite complicated, involving the alteration of three elements in two fields, so the user gives the "lined" command instead of taking the default. "Lined" causes the original command line to be printed out again by COUSIN as a string to be edited by the user through the usual within-line editor. When the user finishes this line edit, the result will be interpreted as a new command at the top command level. In the event, he moves the cursor to the appropriate place, inserts a single space, and types carriage-return to execute the now correct command.

† For those readers unfamiliar with Unix, this is a useful feature supported by the regular shell by which the input and output of commands can be connected to the output and input (respectively) of other commands or to named files.

COUSIN–Unix also provides a way to retrieve and reuse previously executed commands through its history mechanism.

```
   6: hist
      1 = ls
      2 = doverprint -r -c 2 humfact.mss
      3 = doverprint -r -c 2 humfact.mss
      4 = doverprint -r /usr/pjh/papers/humfact.mss
      5 = doverprint -r -n campbell outline
   6: redo 3
   6: doverprint -r -c 2 humfact.mss          [initial string]
   6: doverprint -r -c 3 for campbell humfact.mss
[3 pages * 3 copies ⇒ 10 sheets]
   7:
```

The top-level 'historydisplay" command, uniquely abbreviated here, displays the commands previously executed translated from their form representations (hence the order rearrangements and replacement of whole word markers by Unix style dash markers). The top-level "redo" command allows the user to obtain one of the previous commands as the initial line to the next command prompt. He can then just type carriage return to re-execute the command or line edit if first and execute the edited version as in this example.

An alternative method for reusing old commands is to save the forms derived from them. The form editor provides a "save" command which allows the user to save a form under a name of his choice. Both incorrect and incomplete forms can be saved in exactly the same way as correct and complete ones; thus by saving and then re-editing an incorrect form, the user can fix up problems that cannot be corrected through the form editor (e.g. files having the wrong access permissions). Saved forms are recovered by using their name instead of a regular command name at the start of a top-level command line, whereupon COUSIN–Unix places the user in the form editor editing the saved form, just as though he had typed a command line that parsed into the form, so that if, in particular, the saved form is correct and complete, the user can immediately execute it through the "go" command of the form editor. Any parameters on the command line after the name of the saved form are parsed as though they were parameters to the command from which the form was originally derived, overwriting any conflicting field values from the saved form. Thus saved forms also provide a simple and uniform method to save personalized versions of commands with non-standard parameter defaults.

So far, we have said little about the help component of COUSIN–Unix, and space does not permit a comprehensive set of examples here. To summarize briefly, at any point in any of the above interactions, the user could make a non-specific request for help, and get a summary of the current situation, and his options for action, together with pointers to information relevant to the current context. Suppose, for instance, he had typed "help" or "ESC-?" at the point in the example sequence above where he was correcting the alignment field in the form resulting from his "***dover align sideways /urs/ppjh/papes/umf***" command at prompt "4", i.e. where COUSIN had just prompted him "alignment [vertical]:" and given him the initial string of "sideways" to line edit. COUSIN would display the following help frame.

Correct mode

Before requesting help, you were in Correct mode (type ↑X to return) In Correct mode, the system expects you to provide a value for a form field, by either correcting the present value, or cancelling it (↑C) and typing a new one. The field currently being corrected is alignment of the doverprint form. Its filler must be one of the listed selections.

*⇒form editor commands
*⇒doverprint—further information
*⇒alignment—further information
*⇒general information on Cousin (the interface you are currently talking to)
*⇒line editor commands (all input to Cousin is through a within-line editor)
*⇒how to use the Cousin help system (type "how" to obtain this information)

The characters "*⇒" indicate links to other pre-stored help frames, which may be followed by typing an appropriate initial substring of the following word, so if the user was, for instance, confused about the meaning of the "alignment" parameter, he might type "alig", and causing the first help frame to be replaced by:

Details on the alignment parameter of doverprint

Purpose:	determines whether the printing will be in standarard orientation on the page (vertical), or rotated 90 degrees (horizontal)
Parameter type:	optional
Filler type:	one of: {vertical horizontal}
Default:	vertical
Syntactic Markers:	layout, alignment (followed by explicit value) -r, rotated, landscape (imply horizontal) portrait (implies vertical)
	*⇒ meaning of fields in this frame

These two frames are part of a large network of help frames, containing details of all commands that have blank forms, including frames for each of their parameters like the example above. The frames describing individual commands are generated automatically off-line from the blank forms. They are tied together by hand-written frames for such things as command indices and file system descriptions. Both these kinds of statically defined frames are supplemented by frames dynamically generated from pre-stored templates to satisfy contexually-dependent requests for help. The first frame above is an example of a dynamically generated frame. In all cases, the help is displayed in a separate window at the top of the screen without overwriting or displacing the immediate context that prompted the request for help. As the user follows links from one frame to another, the frames successively overlay each other. Some additional information on the help system was given earlier in this paper in the section on adapting the COUSIN model to Unix , and a much more detailed account is given by Hayes (1982).

4. Conclusion

At the time of writing (December 1982), the version of COUSIN–Unix we have described has been available for two months on several Vaxes in our Department for use by people outside the COUSIN project. We have set up a data collection mechanism, whereby (unless the user specifies otherwise) all sessions with this experimental version are automatically transcripted, using the built-in transcript facilities. The transcripts generated are automatically collected on a daily basis onto a single machine via a file transfer program on our local area network. A comment facility through which COUSIN–Unix users can mail comments directly to the project personnel is also provided. The system operates at about half the speed of the standard Unix shell on correct commands; performance cannot be compared for incorrect commands since the functionality is quite different, but correction by COUSIN–Unix can take up to two or three times as long as the processing of correct commands. Overall, speed does not seem to be a significant problem in using COUSIN–Unix, at least for lightly loaded machines.

Initially, we have encouraged use only by a relatively small set of (about 10) "sympathetic" users, assuming that enough detailed, but practically important problems would emerge from their experience to make a larger, more controlled, evaluation unnecessary and unprofitable before those more obvious problems were corrected. Our assumptions proved to be accurate and we are now engaged in tuning of the implementation to iron out many of the small, but practically important problems that came out in this experimental use, as well as engaging in some slightly longer-term and more extensive revisions also suggested by this small experiment (see below, under screen-oriented COUSIN–Unix). Overall, however, the results of the experiments were strongly positive, with many users expressing enthusiasm for the error-correction and on-line features built into COUSIN–Unix, and deriving from the basic COUSIN form-based model of communication presented in this paper.

To close, we will describe our plans for work on COUSIN in the near future.

Screen-oriented COUSIN–Unix. By far the most common serious complaint about COUSIN–Unix from our group of experimental users was that it did not make the form metaphor of communication very immediate or real to the user. The line-oriented form editor, in particular, only dealt with one field at a time, had no means of keeping the entire context of the present form clearly in the user's mind, and thus failed to convey the notion that the user was editing a form. Its relatively conservative efforts to keep the user informed about the currrent field being edited also made it appear rather verbose. The solution to these problems seems to be a screen-oriented version of COUSIN–Unix in which the form editor operates on a two dimensional image of the current form which is continuously displayed and kept up with the user's changes. A revised version of COUSIN–Unix along these lines is currently being implemented.

COUSIN–SPICE. High on our list of priorities is to complete the implementation of COUSIN on the Perq personal computers, and have it used as one of the command interfaces for the emerging SPICE personal computing environment. We believe that the kinds of service that COUSIN provides will be particularly attractive when they are coupled with the kind of display management and multi-media input only possible with a bit-map display and pointing device. Some of the possibilities for this kind of hardware were sketched out earlier in the paper.

Natural language functionality. Much of the attractiveness of natural language as an interaction medium stems not from its surface forms, which tend to be baroque and redundant, but rather from the elliptical and anaphoric forms which allow people to miss out much information that can be filled in by their listeners. We intend to give users of COUSIN similar opportunities for economy in communication, letting them refer cryptically to objects that are currently being manipulated, and leaving out details if they want the standard thing done. Many interfaces provide defaults, and some keep track of a single current object, but we intend COUSIN to include a mechanism that provides something much closer to the functionality of human ellipses and anaphora. True natural language capabilities are still too poorly understood, and require too much deep cognitive modelling to be included in a practical interface like COUSIN. However, preliminary work by Hayes (1981) suggests that it is possible to devise mechanisms that provide much of the functionality and convenience of natural language, including anaphora and ellipsis, but without deep cognitive modelling, relying instead on the limited semantics of command interaction, and simple adaptation by the user. We expect to incorporate such mechanisms into COUSIN, and determine their utility in practical situations.

Personalization. Just as human conversational participants adapt to the needs of their conversational partners, so should interactive interfaces be sensitive to the differing needs and idiosyncracies of individual users. Some of the areas for adaptation we intend to explore through COUSIN in the short and near-term future include: common typing errors, special vocabulary, parameter defaults, and frequency executed macro commands. Initially, we intend COUSIN to work from explicit descriptions of these individual characteristics, but eventually we hope to devise methods for the interface to personalize itself through observation of the user.

Attractive as these extensions and additional features appear to us at the moment, we view their ultimate disposition as an empirical matter. The real test of user-friendliness is a reduction of frustration and an increase in producitivity for the end user, and the usefulness of any specific interface feature cannot be determined until it comes to be used on a daily basis by people other than the implementors. We look forward with interest to the results of such experiments on COUSIN–Unix and COUSIN–SPICE and on the approach to man–machine communication that they embody.

Eugene Ball and Raj Reddy are co-originators with the primary author of the COUSIN approach to user interface design. Significant contributors to the development of the present COUSIN–Unix interface were also made by Sandeep Johar, George Mouradian, and Mike Rychener.

This research was sponsored by the Defence Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

BALL, J. E. & HAYES, P. J. (1982). A test-bed for user interface designs. *Proceedings. Conference on Human Factors in Computer Systems*, Gaithersburg, Maryland.

DURHAM, I., LAMB, D. D. & SAXE, J. B. (1983). Spelling correction in user inferfaces. *Communications of the Association for Computing Machinery*, **26**.

HANSEN, W. J. (1971). User engineering principles for interactive systems. *Fall Joint Computer Conference, AFIPS*, pp. 523–532.

HAYES, P. J. (1981). Anaphora for limited domain systems. *Proceedings. Seventh International Joint Conference on Artificial Intelligence*, Vancouver, pp. 416–422.

HAYES, P. J. (1982). Uniform help facilities for a cooperative user interface. *Proceedings. National Computer Conference, AFIPS*, Houston.

HAYES, P. J. & REDDY, D. R. (1983). Steps toward graceful interaction in spoken and written man-machine communication. *International Journal of Man–Machine Studies*, **19** (3), 231–284.

HAYES, P. J., BALL, J. E. & REDDY, R. (1981). Breaking the man–machine communication barrier. *Computer*, **14** (3).

LANTZ, K. A. (1980). Uniform interfaces for distributed systems. *Ph.D. Thesis*, Computer Science Department, University of Rochester.

MILLER, R. B. (1968). Response time in man–computer conversational transactions. *AFIPS Proceedings. Fall Joint Computer Conference*, Washington, D.C., pp. 267–277.

NEWELL, A., FAHLMAN, S. & SPROULL, R. F. (1979). Proposal for a joint effort in personal scientific computing. *Technical Report*, Computer Science Department, Carnegie–Mellon University.

NORMAN, D. A. (1981). The trouble with Unix. *Datamation*, **27** (11) 139–150.

RITCHIE, D. M. & THOMPSON, K. (1974). The UNIX time-sharing system. *Communications of the Association for Computing Machinery*, **17** (7), 365–375.

ROBERTSON, G., NEWELL, A. & RAMAKRISHNA, K. (1977). ZOG: A man–machine communication philosophy. *Technical Report*, Carnegie–Mellon University, Computer Science Department.

STALLMAN, R. M. (1981). EMACS: The extensible customizable self-documenting display editor. *Proceedings. ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, Oregon, pp. 147–156.