CMU-CS-84-110

Executable Interface Definitions using Form-Based Interface Abstractions

Philip J. Hayes

March 1, 1984

Abstract

The integral bit-map displays and considerable computational power of the new generation of personal workstations offer the possibility of excellent user interfaces. Yet this potential is often unfulfilled because of the cost and complexity of building user interfaces that fully exploit the available resources. A solution to this problem is to define user interfaces through a language embodying appropriate interface abstractions. Such interface definitions can be interpreted by a central interface system to realize an interface that a user can interact with. If the interface abstractions employed are at a suitably high level, the task of constructing individual interfaces is much simplified, with the complexities of exploiting sophisticated interface hardware limited to the construction of the central interface system.

A specific set of interface abstractions is presented. The abstractions are oriented around a form-filling metaphor of communication between user and application program. They are suitable for defining command interfaces for many, but not all, applications. An attempt is made to delimit their range of applicability.

An interface system that runs on a Perq, a powerful personal workstation, is described. This interface system can interpret interface definitions expressed in a language embodying the interface abstractions just mentioned. The result of this interpretation is a graphical interface with many user-friendly features. An example of an interface definition and the interface that results from it is given.

To Appear in: *Advances in Human-Computer Interaction*, H. R. Hartson, Ed., Ablex, 1984.

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 F33615-81-K-1539. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction
2. Form-based abstractions for command interaction
    2.1. Details of the form-based abstractions
    2.2. Comparison with abstractions based on transition networks
3. COUSIN-SPICE
    3.1. Overview of COUSIN-SPICE
    3.2. An example COUSIN-SPICE interface
    3.3. The COUSIN-SPICE interface definition language
    3.4. Possible extensions to COUSIN-SPICE
4. Conclusion
Acknowledgements
References

List of Figures

Figure 1: External definition of a user interface
Figure 2: Form for simple print application
Figure 3: Form for interactive print application
Figure 4: COUSIN form for the dover print application
Figure 5: Asking for a menu for the `UnprintableCharacters` field

1. Introduction

Good human-computer interfaces have always been expensive and time-consuming to construct, and the more capable and sophisticated an interface is, the worse this problem becomes. The cost of good interfaces is due not only to their initial design and implementation, but also to the extensive refinement process of testing, evaluation, and subsequent modification always necessary to produce truly usable interfaces. Since it is not uncommon for a user interface of modest sophistication to take up the majority of its application system's code, these costs are high even when interaction is conducted through simple alphanumeric terminals. The stakes, however, have been raised dramatically by the present generation of powerful personal computers which can devote previously undreamt of physical and computational resources to support interaction with their users. The integral bit-map displays and pointing devices of these machines can provide menus, forms, and many other kinds of graphical interaction (e.g. [7, 18]). In addition, their often considerable computational power can be used for automatic correction of spelling and other errors (e.g. [2, 4]), highly responsive and integrated help systems (e.g. [3, 14]), and knowledge-based user modelling and support (e.g. [5, 8]). As the best of the interfaces running on such machines show, these capabilities can offer users facilities qualitatively superior to interfaces based on alphanumeric terminals. Yet the high cost of building and refining interfaces to exploit the advanced capabilities often means that they go unused or misused; it is, for instance, unfortunately all too common to see high-resolution bit-map displays used as "glass teletypes". In short, advanced interface features all too often turn into an embarrassment of riches that system builders cannot adequately exploit.

This paper advocates, both in general terms and by reference to a specific implementation, an approach to interface construction which can reduce the cost of developing good interfaces to acceptable levels. In essence, the approach is to spread the high cost of developing interfaces over a large number of applications by building a single central computer program, called an *interface system*, to provide user interfaces for all the different applications. The interface system finds the details of the interface required by each application in an external, declarative data base, called an *interface definition*[1] — one interface definition for each application. These interface definitions are *executable* in the sense that they can be interpreted by the interface system to produce the required interface behaviour for each different application. This results in the situation shown in Figure 1, where the user communicates with the application only indirectly through the interface system. This diagram also shows that the interface definition must, in fact, define two interfaces — the user interface, plus an *application interface* through which the interface system talks to the application.

---

[1] The terms *interface system* and *interface definition* do not have a widely accepted technical meaning, but throughout this paper, they will have the meanings specified above.

However, the latter interface is invisible to the user, and in what follows, we shall be concerned largely with the user interface and the language used for interface definitions.

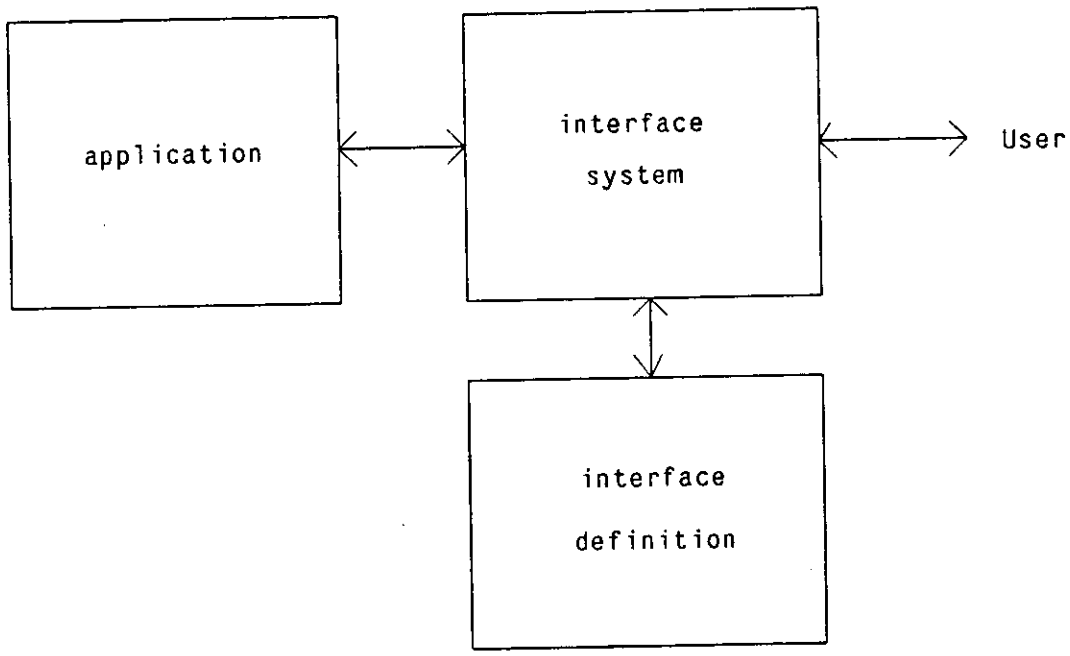

Figure 1: External definition of a user interface

For the approach just outlined to be effective, the interface definition language must be at a level much more abstract than the programming languages normally used for interface implementation. It should instead be built round abstractions of the kinds of communication required by the applications. When such abstractions are used, the approach has clear advantages as a method of interface construction and refinement:

- The interface definition will be much smaller and therefore easier and quicker to construct and modify than a conventionally implemented interface.

- Since the definition is external to the application, many modifications can be made without corresponding changes to the application itself. In other words, changes can be made to the interface definition that alter the user interface without also changing the application interface. This encourages experimentation with different interface characteristics and a more extensive, and presumably more effective, refinement phase.

In addition, there are several reasons to believe the approach will result in superior interfaces that need much less refinement than those produced on an individually tailored basis:

- Since all the details of interaction are managed by a single interface system, there will be a basic level of consistency across all applications using it.

- A significant part of refining an interface involves low-level details of interaction, such as

how best to interpret mouse-movement or keystrokes for menu selection. If the interface definition language is based on appropriate abstractions of the communication needs of the applications, these details will be abstracted out of the interface definitions and confined to the interface system interpreting the definitions. Thus, the details need be worked out only once and not for each new application, thereby allowing them to be evaluated far more carefully and refined far more thoroughly.

- The other major part of interface refinement involves working out the interplay and interference between various aspects of the interaction: for instance, finding good ways of combining command giving through both menu and command lines, or ensuring adequate prompting for missing command arguments. A sufficiently high level of abstraction in the interface definition language would result in a set of interface frameworks in which these details have already been worked out. This would reduce the refinement task largely to one of selecting between the various interface frameworks.

The interface definition language used in the implementation discussed later in this paper operates at the level of abstraction envisioned by the second rather than the third of the preceding itemized paragraphs.

The notion of specifying the interaction required by an application program at a level more abstract than the implementation language of the application is not a new one. There are three basic manifestations of the idea:

1. as a package of subroutines calls (e.g. [16]) that can be inserted into the code for the application to provide interface actions,

2. as a non-executable external description of an application's interface, (e.g. [9, 12]),

3. as an executable external definition of the interaction required by an application (e.g. [4, 6, 19, 20]).

The approach this paper advocates clearly falls into the third category. Let us examine briefly arguments against the first two.

Subroutine packages are a useful first step in sharing the effort of interface construction and refinement across several applications. Diligent use of a comprehensive package across applications can achieve much of the consistency of executable external interface definitions. However, modifying an interface implemented through subroutine calls involves changing (and therefore recompiling and relinking) the application source code. This makes it much less convenient to experiment with variations of an interface than if external interface definitions are used. Moreover, though low-level interaction details can be refined centrally using subroutine packages, the interplay and interference between different aspects of the interaction cannot be refined in the application-independent way envisaged above for executable external interface definitions. It is also difficult to enforce the use of subroutine packages and thus ensure consistency both within and across

applications. It is always tempting for an application builder to bypass the abstraction provided by the package to achieve some special effect, and this is easily done by copying and modifying the relevant subroutine from the package. Such modifications are impossible if the abstractions are implemented externally to the application.

Non-executable external descriptions of an application interface are used to analyze and compare interactions and to predict performance of the interface described, but are not used to produce a simulation of the final interaction. While such descriptions can be used to predict some aspects of the performance of the interface described, it is unlikely that such analysis can ever be as effective as one obtained from direct human factors evaluation of a working interface. A major problem relates to the level of abstraction used in such descriptions. If it is low enough to provide enough details for a complete description of the interaction, the task of constructing it is almost equivalent to that of implementing the interface in a more conventional language. On the other hand, evaluation of a description expressed in terms of higher-level abstractions will depend on assumptions about how the abstractions employed will be realized at the detailed level, and may thus not be valid for all possible realizations. Suppose, for instance, the high-level description specifies presentation of a menu at a certain point in the interaction, but does not (because of the degree of abstraction) specify: whether the menu is fixed or variable size, whether it is scrollable, whether it will obscure other relevant information, etc.. Evaluation of the description may depend on any of these details, thus forcing the person doing the evaluation to make assumptions about them. Since these assumptions may be invalid for some realizations of the description, the evaluation as a whole may also be invalid. By making an external interface description executable, and thus using it as the *definition* of the interface rather than merely a description, these problems are avoided. Execution or interpretation of the definition results in a working and therefore testable interface in which the abstractions used are inevitably made completely concrete. In this way, valid evaluations may be obtained for interfaces defined at an arbitrarily high level of abstraction. Of course, the evaluations are only valid for the specific interface system used to execute the definition, but this is not a problem if that interface system is the one that eventually runs the interface for the end user.

The remainder of this paper demonstrates by example how executable external interface definitions expressed at a high level of abstraction can be used to develop and implement user interfaces. It first presents a specific set of interface abstractions designed for a certain class of command interactions. It then describes an implemented interface system that interprets interface definitions expressed in a language embodying these abstractions. This interface system runs on a powerful personal computer and makes extensive of its powerful graphics capabilities. See [4] for an earlier interface system using similar interface abstractions in the context of an alphanumeric terminal.

2. Form-based abstractions for command interaction

The goal of providing application systems with user interfaces through executable definitions external to the applications themselves is critically dependent on finding suitable abstractions of the interactions required by the applications. This section presents the set of interface abstractions employed by the implemented interface system described in the next section, and compares them with the currently most commonly used type of interface abstractions — those based on recursive transition networks.

While appropriate for many interface situations, the abstractions presented below are not adequate for all forms of interaction (see [17] for some efforts in this direction). In particular:

- The abstractions cover a coarse-grained style of command interaction in which the user repeatedly specifies a command together with a set of dependent parameters subject to specific semantic restrictions.

- It is not possible using the abstractions to specify every way of realizing coarse-grained command interaction. The abstractions make strong assumptions about the style of interface as well as the style of interaction. In particular, they are not suitable for describing arbitrary existing interfaces.

Nevertheless, these abstractions are very useful since coarse-grained command interaction is so common. It is the typical style of interaction at the top level of an operating system interface through which application subsystems are initially invoked. It is also the natural style for many common applications (e.g. electronic mail systems, compilers, symbolic debuggers, etc.). It is different, however, from the finer-grained kind of command interaction that occurs, for instance, with a screen-oriented text editor or a graphics drawing package. This is not to say that fine-grained interaction will not arise in any realization of the abstractions for coarse-grained interaction, but only that the abstractions themselves are not suitable for defining it.

2.1. Details of the form-based abstractions

The abstractions presented here for coarse-grained command interaction are centered round a form-based metaphor of communication. They suppose that the user and the application he wants to use have certain specific 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. Using these abstractions, an interface definition for a given application specifies a *form* containing a *field* for each such piece of information the user and application need to exchange. Such a definition can be interpreted to realize a user interface in the following way:

- The interface system interpreting the definition keeps track of a current value for each field of the form.
- The user is presented with a graphical representation of the form which shows the current values for each field which he may modify as appropriate.
- The application can also access and update the current field values.

Figure 2 shows this for a hypothetical print application. The fields of the form correspond mostly to the input parameters of the application (the pieces of information that the user needs to give to the application) with an additional field for feedback and error messages from the application to the user (the information the application needs to give to the user). Using this arrangement, the user interface from the point of view of the application is very simple: it is just a set of named fields containing values. All the details of how the user actually sees the values in those fields or gets new values into them are taken care of by the interface system.

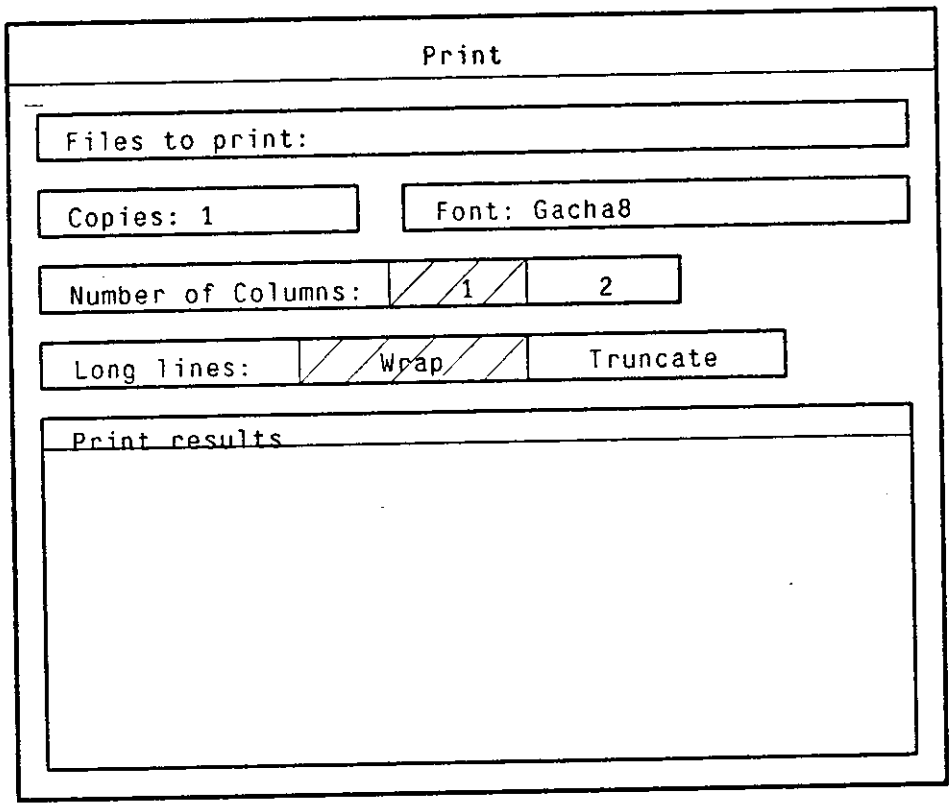

Figure 2: Form for simple print application

The general advantages of implementing user interfaces through interpretation of abstract external definitions of those interfaces have already been discussed. However, the high level of the specific interface abstractions presented here provide other advantages. In particular, form-based interface

abstractions form a good basis for the following kinds of user-friendly behaviour:

- **Correction of erroneous or abbreviated input:** Each form field can be given a type (e.g. positive integer for the "number of copies" field, readable file for the "files to print" field of our print application). Using these types, the interface system can detect and inform the user of any invalid values that he might place in an input field. When it is possible to generate the complete set of correct values for a given type (e.g. the dynamically determined set of available files for file types, or the enumerated set for enumeration types), the system can attempt to spelling-correct an incorrect value. A complete set of available values also makes it possible for the user to fill the fields through unique abbreviations and/or menu selection.

- **Interactive error resolution:** If it proves impossible to correct an incorrect field value or complete an abbreviated value uniquely, the interface system can use its knowledge of the field's type to resolve the problem through interaction with the user (e.g. the system can tell the user that the field is incorrect and present him with a menu of all possible values). The system can also base its interaction on an unsuccessful attempt at automatic correction (e.g. if there are several equally likely spelling corrections of an incorrect input, the user can be presented with a menu of the alternatives). Since fields can have defaults as well as types, 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.

- **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 to the user on request. More complete help can be provided in response to such requests if there is a way for the application builder to include a brief text description of the purpose of each field and the application as a whole in the application definition.

- **Automatically generated on-line documentation:** With the inclusion of text explanations of purpose mentioned in the last item, form definitions contain most of the information a user is likely to wish to know about individual commands. In addition to being displayed as a graphical form, this information can be formatted automatically into on-line documentation. The result is consistent and uniform in organization, and will always be up-to-date with changes in the application that are visible to the user since these changes will naturally be reflected in changes to the interface definition. See [3] for a detailed description of how this documentation was produced in an earlier interface system using form-based interface abstractions.

The form-based interface abstractions we have described so far are appropriate for non-interactive applications. They provide ways for applications to acquire their initial parameters (through input fields) and to report their intermediate and final results (through output fields), but they do not provide any way for the application to interact with the user in a command loop. Command loops in which an application repeatedly accepts, executes, and displays the results of commands from the user are a very common form of coarse-grained command interaction, so it is important to include support for

them in our interface abstractions. Three new concepts are involved:

- **Command fields**: command fields represent commands that the user can issue to the application in a command loop. They have an associated list of other fields in the same form which correspond to their parameters. The user issues a command to the application by changing the value of a command field. If the values in all the parameter fields of a command field are valid and up-to-date (see below), any change to that command field is reported immediately to the application.

- **Active fields**: active fields are like command fields in that changes to their values by the user are reported to the application immediately. The difference is that they have no associated parameters, so that the reporting is unconditional. Active fields can be used for fields representing parameters that are rarely changed, so that it would be inefficient for the application to retrieve their value for every command. Fields which are neither command nor active fields are called static fields. The values of static fields are reported to the application only on demand.

- **Confirmation fields**: confirmation fields have a *timeliness* attribute which may be set to up-to-date or out-of-date. If the attribute is set to out-of-date, the value will not be available to the application until the user has explicitly confirmed as correct the value currently in the field. Confirmation fields represent parameters to application commands that need to be confirmed by the user for each new command invocation.

Using these abstractions, a straightforward way to provide command loop interfaces is as follows:

- The form contains a command field for each command provided by the application. These command fields have binary On/Off values, and can be thought of by the user as 'buttons' that execute the corresponding command when pressed. The list of parameter fields associated with each such command field contains (unsurprisingly) the other fields in the form that provide the parameters to the command it represents.

- After initialization, the application waits to be informed that the user has pushed such a command button. Recall that the application is not informed of such a button push unless all the parameter fields are correct and up-to-date.

- When notification of a command button push arrives, the application retrieves the values of the parameter fields associated with the command, setting the ones that require confirmation to out-of-date (for the next time they are used). The application then executes the command, and waits for the next one.

From the point of view of the application, this arrangement is a straightforward parameters-first verb-last kind of command loop. However, as we shall see from the implementation described in the following section, the error correction and feedback provided by an interface system using these abstractions can make interaction appear much more friendly and supportive from the user's point of view.

Figure 3 shows how we could adapt the form of our hypothetical print command to this arrangement. Two button-style command fields are shown. The 'HardCopy' command actually

produces hard copy on the printer from the files named in 'Files to print'. The 'MakePressFile' command does not actually print the files, but translates them into a file in the binary format, called *press* format, required by the printer. The parameters for 'HardCopy' are all the fields in the form except 'MakePressFile' and 'Press file name' (which specifies the name to be given to the press file created by 'MakePressFile'), while those for 'MakePressFile' are all the fields except 'HardCopy'. Again, this level of abstraction allows various user-friendly features to be provided by the interface system that implements the abstractions, including highlighting all the parameter fields for a given command field on request, and highlighting any incorrect or out-of-date parameter fields when the user tries to execute a command.

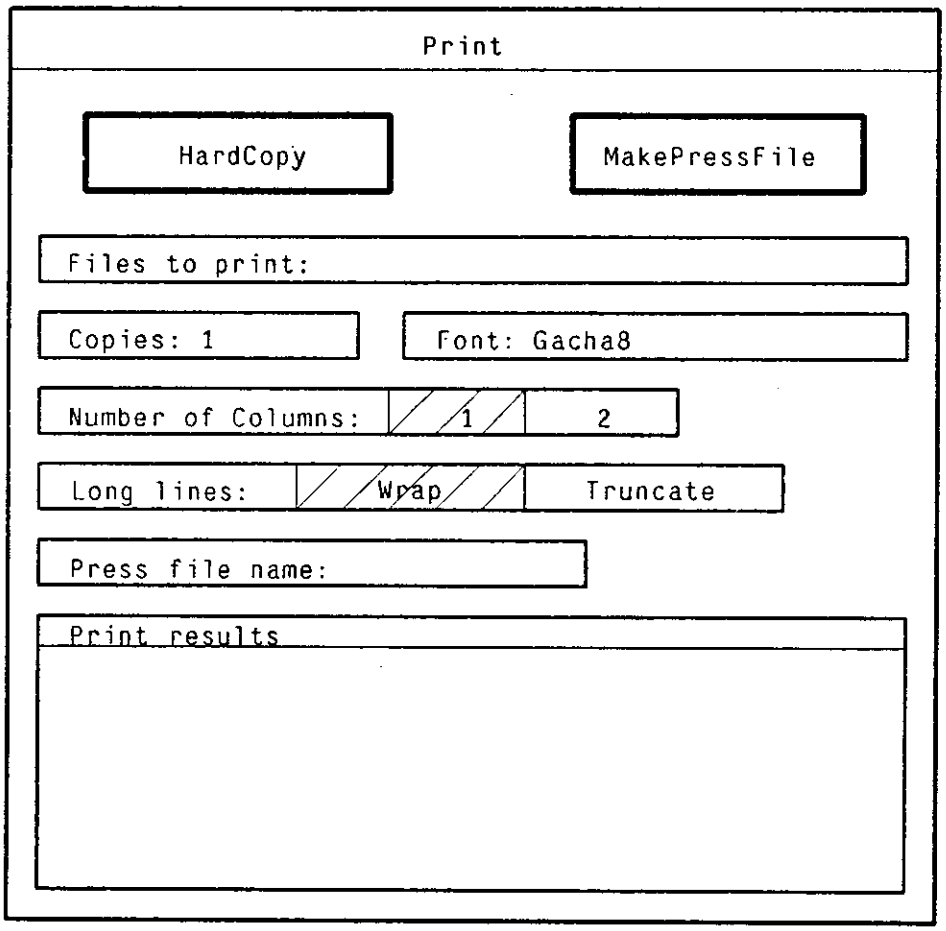

Figure 3: Form for interactive print application

Even within the restricted context of coarse-grained command loop interaction, different applications have differing interaction needs and this particular style of command loop interface may not suit all of them. However, there are several other possible methods of combining the abstractions we have outlined to obtain command loop interfaces of slightly different styles. Some application

commands, for instance, have a small number of parameters that are usually specified differently for each invocation of the command, and a much larger number that are usually defaulted. The 'HardCopy' command above is an example with one non-defaulted parameter, 'Files to print', and four other usually defaulted parameters. In such cases, it is inefficient and unnecessary for the application to retrieve each parameter each time the command is invoked. A solution is to specify the usually defaulted parameters as active fields and have the application note their values whenever the user changes them. Only parameters that are not usually defaulted are actually listed as parameters to the command in the form. Note that this style of interaction requires the application to maintain an internal data structure duplicating the values of the defaulted parameters which are already stored once in the form. In the basic style outlined above, no such parallel structure was necessary.

Another variation in interface style is obtained through command fields which take on one of several application commands as values instead of just being binary-valued buttons. In the case of the print application, this would mean a field, say 'PrintCommand', that could take two values: 'HardCopy' and 'MakePressFile'. A restriction of this arrangement is that both application commands then have to have the same list of parameters. It is also awkwardness for the user to issue the same command twice in a row.

Yet another possibility is to let command fields take a parameter of the command as their value. For instance, the 'MakePressFile' field might be required to have a writable file as its value, which would be interpreted as the file into which to write the press formatted output. This arrangement is most effective when, as in this case, there is exactly one non-defaultable parameter which is used by that command and no others.

The several command styles mentioned here are initial attempts at larger scale interface abstractions. As mentioned in the introductory section, such larger scale abstractions would allow the interplay and interference between different aspects of the interaction to be worked out once and for all and made available to application systems within a single abstraction. We have not yet reached the stage of incorporating abstractions at this level into the interface definition language explicitly, but are still experimenting with them as styles of use of the lower-level abstractions already described. Exactly what the higher-level abstractions turn out to be will depend on how the lower-level abstractions finally end up being used in the context of the implemented interface system described in the next section. This interface system interprets interface definitions expressed in the lower-level abstractions. We are currently constructing and refining interfaces to real applications using these facilities.

2.2. Comparison with abstractions based on transition networks

Before going on to the implementation, it may be worthwhile to contrast the form-based interface abstractions described above with the type of interface abstraction most frequently proposed or used as a basis for command interface definitions or descriptions. This other approach to interface abstraction is based on the recursive transition network (e.g. [1, 6, 19, 20]), or its formal equivalent, the context-free grammar (e.g. [12, 15]). There is a good deal of variation between the different attempts to use transition networks or context-free grammars to define interfaces (see [6, 19, 20] in this volume for instance). However, to a first approximation (using the network rather than the grammar formulation), they all operate along the following lines. Nodes of the network correspond to different states or modes of the interface. Arcs linking nodes have an input and/or output event associated with them. The network is interpreted by starting at some designated initial node and repeatedly traversing arcs to respond to input events and/or to generate output events.

The main abstraction here is the notion of interface modes between which transitions occur as a result of user or system actions. This interface model is, therefore, well suited to scrolled teletype-style interfaces in which the user and system take turns in typing on the same linear typescript. Such interfaces typically have several distinct states, transitions between which are directly determined by user input. Since the only possible interface events are typing on the typescript by either system or user, the transition net model can provide a complete and accurate interface definition for such interfaces.

The suitability of transition net models to other types of interface, particularly graphical interfaces, is, however, less clear. Many graphical interfaces are modeless (or close to it), and often involve interface events, particularly feedback events, that are not connected with mode changes. The lowest levels of graphical feedback, such as cursor tracking, can be handled uniformly at the lexical level of the transition net models, in a way similar to character echoing or long-line wrapping for teletype-style interfaces. However, other kinds of feedback are more specialized and would be quite awkward to handle at the lexical level. A graphical interface using the form in Figure 3, for instance, might want to provide additional feedback by changing the cursor to one shape when the user moved it over a command field whose parameters were all correct and up-to-date, and to a different shape when it was moved over a command field with incorrect or out-of-date parameters. This cursor changing is a state transition of a kind, but the corresponding state transition diagram would be quite large and unwieldy. It is also hard for a transition net scheme to account for interface events that might be going on strictly in parallel. The system might be printing a message out in one of the fields, for instance, at the same time as the user is moving the cursor. Or the user might be typing characters (with one hand) at the same time as he was moving the cursor, with the destination of the

characters on the screen possibly being determined by the position of the cursor. The form-based interface abstractions presented above thus (perhaps not too surprisingly) appear better able to describe graphical form-based interfaces than the recursive transition net or context-free language models.

3. COUSIN-SPICE

The previous section discussed in general terms an approach to executable interface definition centered round form-based interface abstractions. In this section, we will examine how this approach has been put into practice in the COUSIN-SPICE interface system.

3.1. Overview of COUSIN-SPICE

The COUSIN-SPICE interface system operates in the context of the SPICE project [10] of Carnegie-Mellon University Computer Science Department. The goal of SPICE is to provide an individual researcher with an excellent computational environment based on a powerful personal computer linked together in a high-bandwidth network with mainframes and other personal machines. The current target machine for SPICE is the Perq [11], a personal workstation with an integral high-resolution bit-map display and pointing device, 1 MByte of main memory, and a 1 MIPS processor.

User interfaces are, naturally, a major concern of the SPICE project, but the richness of the hardware resources available means that all the problems outlined in the introduction apply to the construction of interfaces for application systems. Most application builders do not have the resources of time and effort necessary to develop interfaces that exploit the hardware adequately. Consequently, scrolled teletype-style interfaces have been the norm. As we have argued, this is a situation in which significant practical benefits are likely to result from providing interfaces through an interface system operating from external interface descriptions.

COUSIN-SPICE interprets external interface definitions to provide a variety of SPICE applications with mutually consistent, user-friendly interfaces that fully exploit the available hardware resources. The interface definitions used by COUSIN-SPICE are expressed in the form-based interface abstractions introduced in the previous section. Since it is relatively simple to modify interfaces produced in this way, we hope that interfaces implemented through COUSIN (sometimes we will omit the '-SPICE') will be subjected to an extensive refinement process. This refinement process should not only result in truly excellent interfaces, but, when done for a sufficient number of applications, should also help us to formulate the higher-level interface abstractions discussed in the previous section. While all interface evaluation and refinement done so far for COUSIN interfaces has been based on informal

observations of the interfaces in use, we hope eventually to incorporate more thorough human factors evaluation procedures into the refinement loop.

To use the facilities provided by COUSIN-SPICE, an application builder must do two things:

1. Write an interface definition in the language interpreted by COUSIN-SPICE. This defines a form-based interface of the kind we have been examining.

2. Write the application in such a way that it does not communicate directly with the user, but instead communicates with the COUSIN-SPICE interface system. COUSIN communicates with each application in terms of the values of the fields in the form defined for that application.

Once the application builder has done this, COUSIN can control all interaction between the user and the form defined by the interface definition, thus providing a cooperative, graphically-oriented interface for the application. While this arrangement allows the application builder to produce powerful interfaces with minimal effort, it also imposes certain constraints which should be made explicit here:

- COUSIN can only provide interfaces to those applications suited to form-based interaction. However, since COUSIN forms are appropriate for most coarse-grained command interaction (though see Section 3.4), the range of applicability of COUSIN is still very wide.

- Since COUSIN's interface abstractions are based on forms, there may be styles of interaction that would be suitable for a given application that cannot be implemented through COUSIN. On the other hand, the uniformity of COUSIN interfaces across a variety of applications may be more important than getting the interface "exactly" right for any given application.

- COUSIN cannot be used directly with existing applications. The applications must be modified to communicate directly with COUSIN according to COUSIN's pre-established protocol.

An example of a COUSIN interface definition, and the user interface that COUSIN provides from it will provide a more concrete picture of COUSIN.

3.2. An example COUSIN-SPICE interface

In COUSIN-SPICE, an interface definition consists of a form name definition followed by a sequence of field definitions. Each field definition consists of a set of attributes with each attribute defining some aspect of the appearance or behaviour of that field as it appears to the user in the form. For instance, here is a definition for a slightly extended version of the print application we have been using for examples.

```
[
  FormName: print
]


[
  FieldName : "Files to print"
  MaxNumber : 25
  MinNumber : 1
  Purpose : "Files to be printed"
]


[
  FieldName : HardCopy
  ValueType : Button
  InteractionMode : PushButton
  ChangeResponse : Command
  Parameters :
    (
      "Files to print"
    )
  Purpose : "Produce hardcopy of the files to print"
]


[
  FieldName : MakePressFile
  DefaultSource : NoDefault
  ValueType : Button
  InteractionMode : PushButton
  ChangeResponse : Command
  Parameters :
    (
      "Files to print"
      PressFileName
    )
  Purpose : "Produce a press formatted version of the files to print,
               but do not produce hard copy"
]


[
  FieldName : Quit
  ValueType : Button
  InteractionMode : PushButton
  ChangeResponse : Command
  Purpose : "Exit the file printing program"
]
```

```
[
  FieldName : Copies
  ValueType : Integer
  LowerBound : 1
  UpperBound : 200
  DefaultValue : 1
  ChangeResponse : InformApplication
  Purpose : "The number of copies to print of each file"
]

[
  FieldName : Font
  DefaultValue : Gacha8
  ChangeResponse : InformApplication
  Purpose : "The font in which to print the files"
]

[
  FieldName : Orientation
  EnumeratedValues :
    (
      Landscape
      Portrait
    )
  DefaultValue : Portrait
  ChangeResponse : InformApplication
  Purpose : "Printing orientation: Landscape is across length of page,
               Portrait across width"
]

[
  FieldName : LongLines
  EnumeratedValues :
    (
      Wrap
      Truncate
    )
  DefaultValue : Wrap
  ChangeResponse : InformApplication
  Purpose : "What to do with lines that are too wide"
]
```

```
[
  FieldName : "UnprintableCharacters"
  EnumeratedValues :
    (
      OctalRepresentation
      CaretConvention
      Omit
    )
  DefaultValue : OctalRepresentation
  ChangeResponse : InformApplication
  Purpose : "How to deal with unprintable characters:
               use their three-digit octal representation (\nnn),
               use the caret convention (↑C = control-c),
               or omit them."
]

[
  FieldName : NumberOfColumns
  ValueType : Integer
  LowerBound : 1
  UpperBound : 2
  DefaultValue : 1
  ChangeResponse : InformApplication
  Purpose : "The number of columns per page for non-press files"
]

[
  FieldName : PressFileName
  DefaultValue : ""
  Purpose : "Name for press formatted file produced by MakePressFile command"
]

[
  FieldName : "Standard Typescript"
  ValueType : Port
  InteractionMode : TypescriptField
]
```

A drawing of the user's view of the interface produced by COUSIN-SPICE using this definition[2] is shown in Figure 4.[3] Complete details of how the various elements of the form definition influence the

---

[2] A simplified version of the definition for the dover printing application that runs on a Perq and uses COUSIN-SPICE to provide its interface.

[3] Since the form is formatted automatically by COUSIN to fit into a window on the Perq screen whose size and shape can be altered by the user, the alignment of the fields can vary.

appearance and behaviour of the form are presented in Section 3.3. For now, it is sufficient to note that each field definition is a property list of attribute/value pairs with one of the pairs specifying the name of the field. Each field definition results in a field of the given name in the form, and the appearance and behaviour of each form field are controlled by the other attribute/value pairs in the field definition. **Hardcopy**, **MakePressFile**, and **Quit**, for instance, have a speckled background (not reproduced here), and are grouped on a line together because they have **Command** and **Pushbutton** attributes. Note that the set of attributes given vary from one field to another. In fact, all fields have all attributes, but in many cases, straightforward defaults are available (see Section 3.3). Figure 4 also shows two unnamed fields in the form. A scrolled window is provided automatically at the top of every form for help and other messages from COUSIN to the user. Our example form also contains a typescript field at the bottom (corresponding to the **Standard Typescript** field in the form definition) that the print application uses for messages to the user. Already this is a lot of interface for a small amount of effort on the part of the application builder, and some example interactions with the user will show how complete an interface it provides.

Suppose the user, with the form in the initial state shown in Figure 4, presses the **HardCopy** button.[4] The image of the button changes to inverse video (white on black) to acknowledge the push and to indicate that the command cannot be executed because some of its parameters are incorrect. The problem lies with the **Files to print** field which has no value. So this field also changes to inverse video to indicate that it is a parameter, but is incorrect. The user then types the name of a file into the **Files to print** field. This causes the image of that field to change to a grey background, indicating it is a correct parameter of the pending command. The **HardCopy** button also changes to a grey background, indicating that all its parameters are correct — it only has one — and that it is, therefore, ready to execute. Finally, the user presses **HardCopy** again, the file that he named is queued for printing, and **HardCopy** and **Files to print** revert to their initial backgrounds.

This is the user level view. To see the contribution of COUSIN to this interaction, it is necessary to dig below the surface. When the user first presses **HardCopy**, COUSIN notes from the interface definition presented above that the **InteractionMode** of **HardCopy** is **PushButton**, and so immediately changes **HardCopy** to have the value 'on', providing feedback that this has happened by changing the field to a grey background. In addition, since the **ChangeResponse** of **HardCopy** is **Command**, COUSIN assumes that the user is attempting to execute a command and starts to check the parameters listed for the field. In this case, there is just one parameter, **Files to print**, which

[4] By moving the mouse-controlled cursor over the image of the button on the screen and clicking the physical button on the mouse.

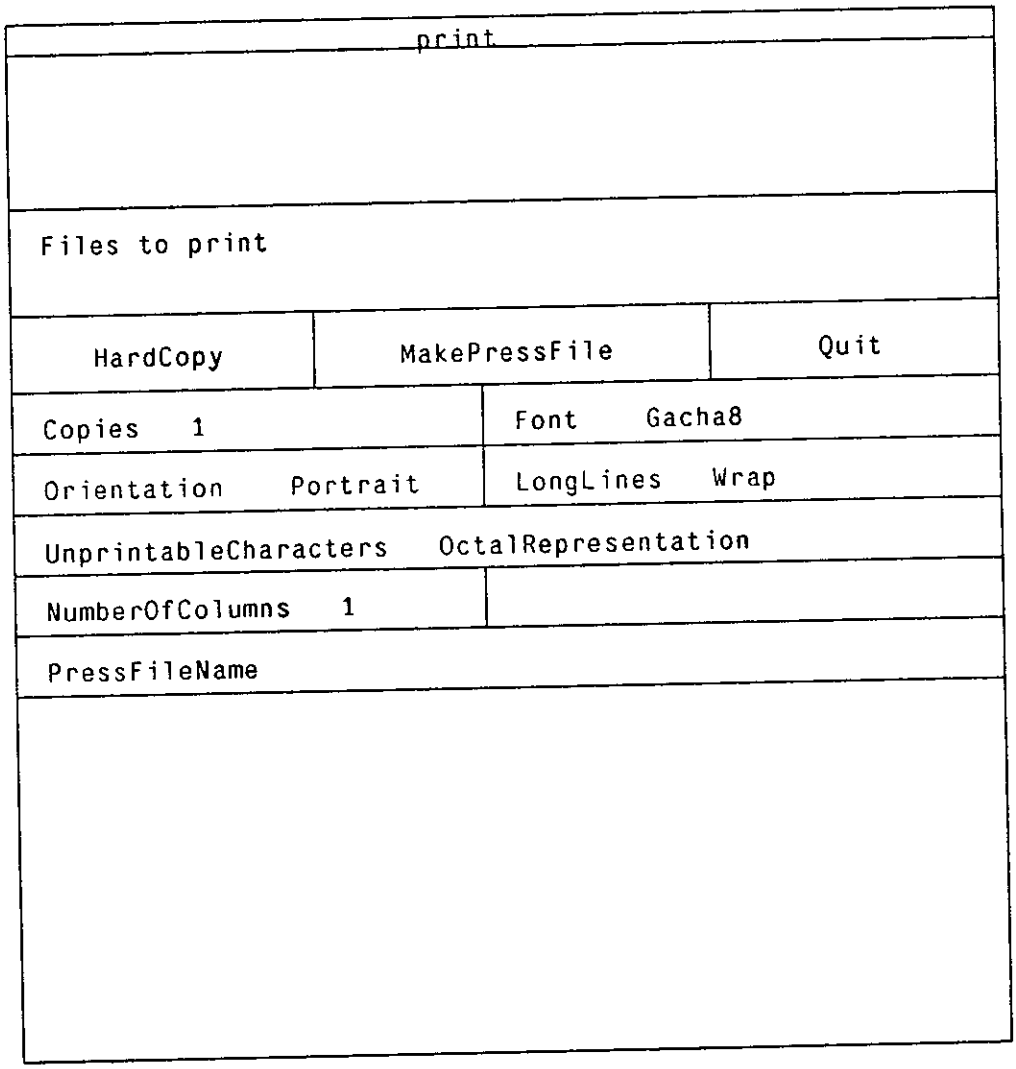

Figure 4: COUSIN form for the dover print application

according to the form is required to contain between one and twenty five readable files. It actually doesn't contain anything, so the field is incorrect, and COUSIN indicates this to the user by the inverted display. As soon as a parameter is found to be incorrect, COUSIN also changes the command field to have an inverted display. Normally, this happens fast enough that the user is not aware of the intermediate change to grey.

At this stage, COUSIN has established that the user is trying to issue a command, but has not specified all the parameters of the command correctly. So, after giving the feedback just mentioned, it simply waits for the user to take corrective action (or do whatever else he wants to: change another field, ask for help, press a different command button, or even interact with an entirely different application). As it happens, the user takes corrective action by inserting the name of a readable file

into **Files to print**. Since the constraints specified for this field are now satisfied, COUSIN removes the error feedback. Also, since the field is a parameter to the pending command, COUSIN replaces the error feedback with feedback indicating this fact (the grey background). Also, since all parameters to that command are now correct (there was only one), the background of **HardCopy** is also changed to grey indicate that the command is ready to execute. Note that up to this point, the application has not been involved in the interaction at all. It is only when the user presses **HardCopy** again that COUSIN sends a message[5] to the application saying that the HardCopy command is being issued and that all its parameters are complete and correct. The application then retrieves the parameter it needs (by asking COUSIN for the value of **Files to print**), tells COUSIN it has finished retrieving values from the form,[6] queues the file for printing, and waits for another command notification from COUSIN. The application's part in this fairly involved interaction is thus quite small.

This example shows how the user can get COUSIN to prompt him for the parameters to commands. He is, of course, free also to specify the parameters in advance of the command. For instance, by typing the name of a writable file into **PressFileName**, replacing **Files to print** with the name of another readable file, and pressing the **MakePressFile** button, he would create a ready-to-print version of the file named in **Files to print** on the file named in **PressFileName**. The action of COUSIN is similar to the previous case, the main difference being that when it comes to check the parameters to the command being issued, it finds them both correct and thus transmits the command to the application immediately. Note that if the user had not replaced the value of **Files to print**, COUSIN would not have issued the command immediately because **Files to print** would have been marked out of date from its previous use in the **HardCopy** command. However, if the user really wanted to use the same file again, he would only need to confirm the value and the command could proceed.

So far, the user has interacted only with the two commands of the form and their required parameters. The remainder of the form consists of optional parameters which users sometimes want to change but which they usually are satisfied to leave with the default values. Suppose the user is dissatisfied with the default handling of unprintable characters which the form lists as **OctalRepresentation**. Before issuing his next print command, he can change the values in one of several ways. By pointing at the field with the mouse-controlled cursor and clicking a mouse button,

---

[5] COUSIN and the applications it serves run as separate processes in the Accent operating system on the Perq and all communication between them is handled by the interprocess communication facility of that operating system.

[6] To avoid race conditions the user is not allowed to change the form after a command has been issued to the application until this notification is received.

he can change the field to another of its possible values, **CaretConvention**. Repeated clicking will cycle through the remaining values (in this case there is only one, **Omit**), and then go back to the first and start again. COUSIN supports value cycling on all fields like **UnprintableCharacters** whose possible values are defined explicitly through an **EnumeratedValues** attribute. Alternatively, should the user want to select from a menu of all the possible values of the field, a different keystroke will cause a pop-up menu of the values to appear. If the user selects an element of this menu, the corresponding value will be inserted in the field. Along with the menu, COUSIN prints the **Purpose** attribute of the field in the message area at the top of the form, so the situation immediately after requesting the menu for **UnprintableCharacters** is as shown in Figure 5. In addition to facilitating form-filling, the pop-up menus and help text comprise a simple, but uniform and consistent, help system.

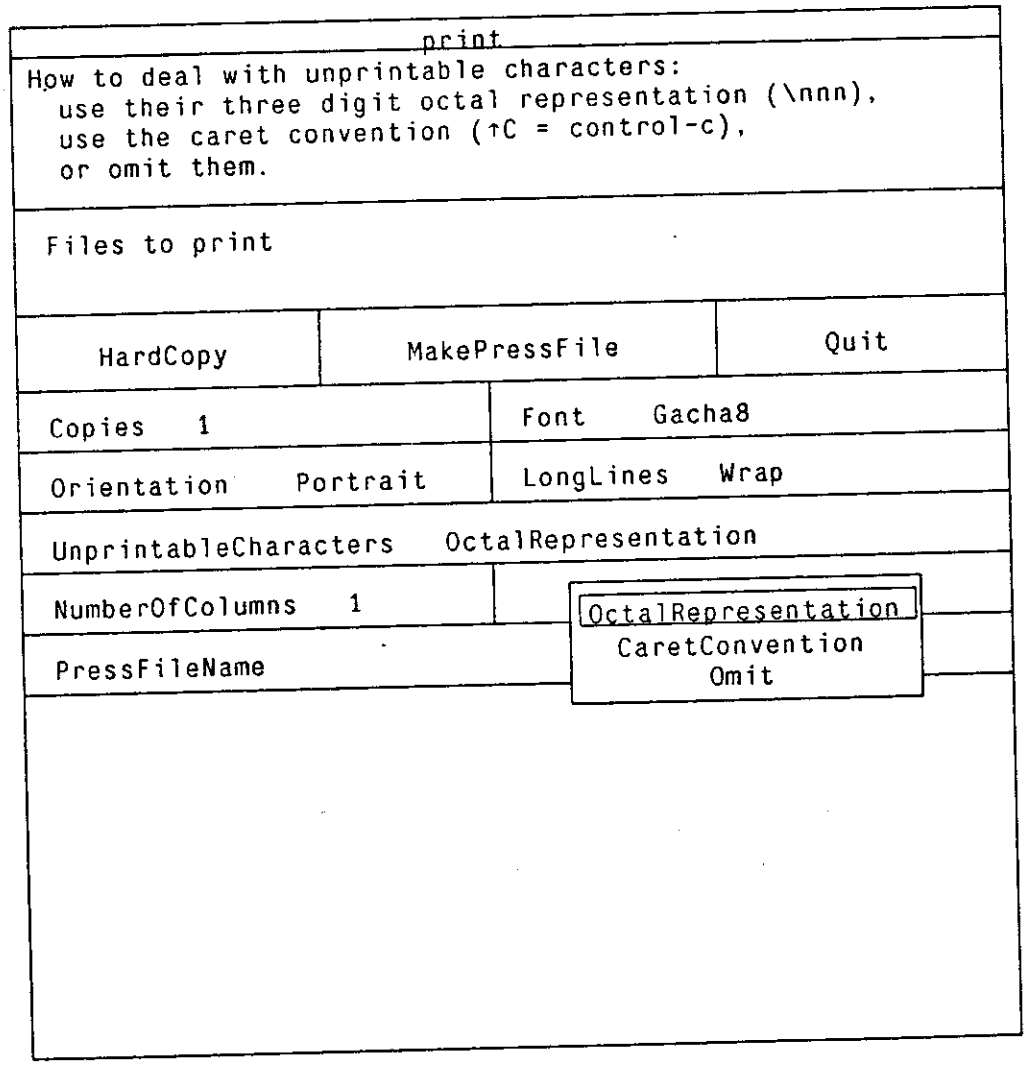

Figure 5: Asking for a menu for the **UnprintableCharacters** field

Since the **ChangeResponse** for **UnprintableCharacters** is **InformApplication**, COUSIN immediately informs the application when the user inserts a new, correct value into the field. All the optional parameters for the print application are handled this way so that their values do not need to be retrieved by the application for each **HardCopy** or **MakePressFile** command that the user issues.

A final form-filling aid provided by COUSIN is error-correction. If the user enters an incorrect value into a field, COUSIN immediately informs the user in the way described earlier that it is an error. In addition, COUSIN also attempts to spelling-correct the bad value against the set of possible correct values if these are known, as they are for enumerated or file types.[7] If a unique correction is available, the incorrect value is overwritten and the field is treated as though it were out of date so that the user will be required to confirm the correction. If there are several equally likely corrections, they are offered to the user in the form of a menu just like the menus produced in response to explicit user requests, except that this menu contains only the potential corrections and not all the values that could go in the field.

3.3. The COUSIN-SPICE interface definition language

As already described, the interface abstractions provided by COUSIN are oriented round a form made up of named, value-containing fields which are used for communication between user and application. We have seen that there are different kinds of fields that behave differently from both the user's and the application's points of view. This section presents the details of the language that the application builder uses to define a COUSIN form, and thus the details of the interface abstractions that COUSIN supports.

A COUSIN form definition consists of a declaration of the name of the form, plus a sequence of field definitions. Each field definition consists of a list of the following attributes (curly brackets ({}) indicate a choice of one element from the set, angle brackets (<>) indicate an attribute of that type, round brackets indicate lists):

[7] Checking and correction of file types is not implemented yet.

```
[
Name: <String>
ValueType: {Integer, Boolean, String, Button, ReadableFile, WritableFile, Port}
MaxNumber: <Integer>
MinNumber: <Integer>
EnumeratedValues: (value1 value2 ...)
LowerBound: <Integer>
UpperBound: <Integer>
DefaultSource: {NoDefault, ExplicitDefault, ApplicationDefault}
DefaultValue: <String>
ChangeResponse: {Passive, InformApplication, Command}
Parameters: (FieldName1 FieldName2 ...)
InteractionMode: {EditIn, PushButton, CycleButton,
          DisplayOnly, Invisible, Typescript, Canvas}
]
```

These attributes taken together determine the appearance of the field to the user and the behaviour of the field for both the application and the user. Most of the attributes have defaults which are used if the attribute is not mentioned in a particular field definition, as is true in many cases in the example form definition in the previous section. The meaning and default values of the attributes are:

Name: the name used to label the field on the screen for the user and to identify the field to the application.

ValueType: The kind of object that is supposed to go in the field (default *String*). The last two values require explanation:

- Button: *{On Off}*; required when InteractionMode is PushButton.
- Port: For use when InteractionMode is Typescript or Canvas.

MaxNumber, MinNumber:
the minimum and maximum number of values permitted in the field (both default to 1).

EnumeratedValues:
an initial list of possible values that the field is restricted to.

LowerBound, UpperBound:
on the value if the field has an integer value.

DefaultSource:

- *NoDefault*: the field has no default (the default).
- *ExplicitDefault*: the default is listed explicitly in the application description.
- *ApplicationDefault*: the field has a default which will be determined dynamically by the application, so the user can leave this field empty and yet it is still counted as having a correct value.

DefaultValue: the default value of the field; required when DefaultSource is ExplicitDefault.

ChangeResponse:

Passive:	only check for validity when a new value is entered; do not tell the application (the default).
InformApplication:	tell the application as well as checking for validity.
Command:	first check the values of the fields listed as parameters, and inform the application only if all these values are correct.

Parameters: the list of parameter fields for use when ChangeResponse is Command.

InteractionMode:

the way the field is presented to the user and the way he is allowed to interact with it.

EditIn:	the user types the value into the field (the default).
PushButton:	the field looks like a button, can be pushed, and has two values (ButtonOn and ButtonOff); requires ValueType to be Button.
CycleButton:	the field name does not appear, only the value; useful with binary-valued enumerated value fields if the value names are well chosen.
DisplayOnly:	can only be modified by the application.
Invisible:	the value is not displayed (for use with passwords).
Typescript:	for typescript interaction between the application and user directly, or for scrolled output from the application.[8]
Canvas:	for any kind of free-form graphical output or direct user interaction that the application builder wishes.

In summary, COUSIN-SPICE supports interface abstractions organized round the notion of a form consisting of a set of named, value-containing fields that are used for communication between application and user. The fields can be restricted to contain a specified number of values of a specified type, can have a default value, and can be displayed to the user in a variety of ways. The application builder can specify whether a user's changes to a field's value will be transmitted immediately to the application or only upon demand. There is also the notion of a command field, changes to whose values are transmitted immediately to the application providing that the values of other (parameter) fields are correct.

[8] Currently, only one typescript field is allowed per form and it is always placed at the bottom of the form, using whatever space remains in the form's window.

3.4. Possible extensions to COUSIN-SPICE

The present version of COUSIN-SPICE supports the definition of interactive user interfaces at a high level of abstraction. Preliminary use of the system suggests that the set of interface abstractions it provides are sufficient for many, though not all (see below for an example), common applications requiring coarse-grained command interaction. This section presents some of the deficiencies we believe exist in the current version of COUSIN-SPICE, both in terms of the interface abstractions it provides and the way it realizes them. That we can make this distinction and work on the two issues independently underlines the importance of interface abstraction. It gives us the ability to work on a major part of the interface refinement task (the part involving the way the abstractions are provided) without changing any of the application systems that use the abstractions or their interface definitions.

Areas where we plan to concentrate future efforts include:

- **More comprehensive interface abstractions**: The interface abstractions supported by the current version of COUSIN-SPICE largely concern communication between user and application through a collection of independent fields. While the fields are grouped in a dynamically generated, graphically displayed form, and the notion of some fields being parameters to other command fields is supported, this falls well short of a comprehensive abstraction of the complete set of interactive services required by even a simple application. Missing concepts include a complete command listing for the application, a distinction between optional and required parameters, a standard packaged command format including both commands and parameters for transmitting commands to applications, and no doubt others. Moreover, we think it unlikely that one comprehensive interface abstraction will be adequate for the diverse communication needs found in application systems. Therefore, we plan to develop and experiment with several abstractions that are more complete.

- **Structured forms**: Another deficiency of COUSIN's current abstractions is the single-level unstructured nature of forms — an ability to structure fields into groups and have forms within forms is needed. Consider, for example, an electronic mail system. One likely component of a form-based interface to such a system would be a field containing a set of summaries of current messages. Each summary would have an inherent structure, comprising, say, the sender, subject, and date of the corresponding message. The interface designer might also wish to incorporate other elements of structure, say command buttons for each summary to erase the corresponding message or display it more fully. Since more than one summary may appear in the summaries field, a subordinate level of structure would be needed for the application builder to do this.

- **Display and layout issues**: From preliminary experience, it seems unlikely that the present automatic approach to form layout can provide forms that meet sufficiently high standards of graphic design — at least not with the information provided through the current set of interface abstractions. An obvious solution is to allow the application builder to lay out a form himself through a graphical editor, but we would like to avoid this approach if at all possible because the present implementation operates in the context of

a window management system through which the user may change the size and shape of windows at any time. A scheme by which the application builder could specify constraints on the grouping and alignment of fields would be preferable. With the development of the more comprehensive application abstractions mentioned above, it might even be possible to make a default set of such constraints implicit in the more basic aspects of an interface definition. Note also that once a suitable method for describing form layout is established, there is no reason to restrict its use to the application builder. Personalizable forms could be supported by allowing the end user to modify the layout definition dynamically. We are currently working on a scheme along these lines in which the form seen by the application is treated as an "abstract" form which can be mapped onto what the user sees on the screen in various ways, the mapping itself being defined by another form which the user or application builder will be able to view and modify dynamically.

- **Programming with applications**: One of the most attractive features of the Unix operating system [13] is that several applications can be joined together in *pipelines* to accomplish a larger scale task. Communication along these pipelines is via character streams so that it can be made transparent to an application whether it is communicating with another application via pipeline or with the user via a terminal. It would be nice to retain the power of pipelines for applications that communicate with the user via forms, but clearly the character stream mode of communication is no longer feasible. We are currently looking at ways to format the input and output of applications in terms of structures that could be generated either by other applications or by an interface system like COUSIN. The issues here relate both to the way that commands are issued to an application and to the way an application gives the "outside world" feedback about what it is doing in response. The design of such communication is also highly related to the more comprehensive interface abstractions mentioned above. Finally, given a composite application comprised by a set of applications joined together via pipelines or more complicated iterative control structures, it should be possible to define a form interface for that composite application in just the same way as it is possible to define forms for individual applications.

- **Integration into a complete system**: The present COUSIN-SPICE system is concerned only with the interface between a user and an individual application program. Applications, however, do not exist in isolation — a large set of more or less related applications is normally offered to the user as part of an overall interactive computing environment. Exactly which applications are offered depends on the purpose of the environment; an office information environment would offer a quite different set from a scientific computing environment such as SPICE. The only contribution the present COUSIN system would make to this notion of an overall environment is to ensure the individual interfaces are consistent across the various applications. More support is needed for the overall environment, perhaps by including information about function and relation to the rest of the environment in the interface definition of each application. Using this information, on-line help indices could be constructed automatically. We hope to experiment with this kind of integration of individual applications into an overall environment as COUSIN becomes more widely used for interfaces within the SPICE environment.

4. Conclusion

As interface hardware becomes increasingly capable and as interfaces show a corresponding increase in sophistication and complexity, the need to share the burden of interface development across application systems will become more and more critical. This paper has argued that the way to share this burden is to provide an abstract language for interface definition and an interface system to provide application interfaces by interpreting definitions in that language. A suitable level of abstraction in the interface language not only reduces the amount of effort required by the application builder to construct an interface, but also is likely to result in higher quality interfaces than if application interfaces were constructed individually. The reasons for this are:

- the details of the interaction required to implement the various interface abstractions need only be worked out once, and therefore a large amount of time and effort can be devoted to their evaluation and refinement;

- in addition to facilitating construction of the initial interface, the abstracted interface definition language makes it easy to change the interface and thus permits and encourages rapid interface prototyping, evaluation, and refinement;

- interfaces are consistent across all the applications employing the interface system.

These arguments were supported and illustrated by a description of such an interface definition language and an implemented interface system which supports it in the context of a powerful personal computer. The abstractions provided by the language were oriented round a form-based metaphor of communication. Interface definitions using them can be interpreted by the implemented system to provide end user interfaces with the appearance of interactive graphical forms to a variety of application systems. Many issues remain to be resolved in the abstractions provided by this language and in the realization of these abstractions. Nevertheless, we believe that interface technology is developing so rapidly that it will soon become impractical to develop interfaces for a set of applications comprising a computing environment without using the basic approach that this paper has advocated — interface definition via a language containing a high level of interface abstraction.

Acknowledgements

The detailed design and implementation of the COUSIN-SPICE system are due largely to Rick Lerner and Pedro Szekely.

The content and presentation of this paper have benefitted substantially from reviews by authors of other papers in the collection in which it is to appear (see title page), and from comments by Pedro Szekely, Howard Gayle, and Cynthia Hibbard.

References

1. Bleser, T. and Foley, J. D. Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces. Proceedings of Conference on Human Factors in Computer Systems, Gaithersburg, Maryland, March, 1982, pp. 309-314.

2. Durham, I., Lamb, D. D., and Saxe, J. B. "Spelling Correction in User Interfaces." *Comm. ACM 26* (1983).

3. Hayes, P. J. Uniform Help Facilities for a Cooperative User Interface. Proc. National Computer Conference, AFIPS, Houston, June, 1982.

4. Hayes, P. J. and Szekely, P. A. "Graceful interaction through the COUSIN command interface." *International Journal of Man-Machine Studies 19*, 3 (September 1983), 285-305.

5. Huff, K. E. and Lesser, V. R. Knowledge-Based Command Understanding: An Example for the Software Development Environment. Computer and Information Sciences, University of Amherst, Massachusetts, 1982.

6. Jacob, R. J. K. An Executable Specification Technique for Describing Human-Computer Interaction. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed.,Ablex, New Jersey, 1984.

7. Kaczmarek, T., Lipkis, T., Mark, W., Robins, G., Sondheimer, N., Swartout, W., and Wilczynski, D. The Consul/CUE Interface: An Integrated Interactive Environment. CHI '83 Conference: Human Factors in Computing Systems, Boston, December, 1983.

8. Mark, W. Representation and Inference in the Consul System. Proc. Seventh Int. Jt. Conf. on Artificial Intelligence, Vancouver, August, 1981, pp. 375-381.

9. Moran, T. P. "The Command Language Grammar: a representation for the user interface of interactive computer systems." *International Journal of Man-Machine Studies 15* (1981), 3-50.

10. 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.

11. Perq. Three Rivers Computer Corp., 160 N. Craig St., Pittsburgh, PA 15213. .

12. Reisner, P. "Formal grammar and human factors design of an interactive graphics system." *IEEE Transactions on Software Engineering 7* (1981), 229-240.

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

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. Schneiderman, B. "Multi-Party Grammars and Related Features for Defining Interactive Systems." *IEEE Transactions on Systems, Man, and Cybernetics* (March 1982).

16. Shafer, S. A. CI Command Interpreter. In *Unix Programmer's Manual (CMU additions)*, Bell Labs, 1983, ch. 3.

17. Shaw, M., Borison, E., Horowitz, M., Lane, T., Nichols, D., and Pausch, R. "Descartes: A Programming-Language Approach to Interactive Display Interfaces." *ACM Sigplan Notices 18*, 6 (June 1983), 100-111.

18. Teitelman, W. A Display Oriented Programmer's Assistant. Proc. Fifth Int. Jt. Conf. on Artificial Intelligence, MIT, August, 1977, pp. 905-915.

19. Wasserman, A. I. and Shewmake, D. T. The Role of Prototypes in the User Software Engineering (USE) Methodology. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed.,Ablex, New Jersey, 1984.

20. Yunten, T. and Hartson, H. R. A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed.,Ablex, New Jersey, 1984.