R

4

1981a

In Proc. 6th ACM European Conf. London, 1981.

PROVIDING THE USER WITH A TAILOR-MADE INTERFACE

I. Sommerville

Dept. of Computer Science, University of Strathclyde, Glasgow, Scotland

This paper describes a system for generating user interfaces to computing systems and an associated architecture which can support many different interfaces running simultaneously. By using a pattern matching system, complex and/or informal user interfaces, tailored towards different classes of user can easily be generated. The proposed architecture utilises programmable terminals and has the side effect of offering extra system security features.

INTRODUCTION

As computer hardware becomes cheaper, it is inevitable that computer applications will become more diverse and that more and more people will directly interact with a computer system as a normal part of their working life.

In some cases, these interactions will be with personal computer systems, in others, the interaction will be with a dedicated transaction processing system and in yet others, the interaction will be with a general purpose operating system or application system, offering a range of facilities to many diverse users. It is to this latter group of users, that is, those interacting with a general purpose system, that our work is addressed.

Computer users interact with the computer via a human-computer interface which we term the user interface. In operating system terms, this is the job control language but, more generally, the user interface is made up of those commands which allow the user to access and control system facilities irrespective of whether the system is an operating system, a database management system, an information retrieval system or any other application system.

Existing computer interfaces which involve the user issuing system commands can be classified into two broad categories:-

(1) Those interfaces tailored towards the user who has some computing background and knowledge of computing terminology. An example of such an interface is the interface to a time-sharing system such as Unix[1].

(2) Transaction processing interfaces where the system is dedicated to a single application. The system users are trained in that application and in use of the computer. An example of such a system might be an airline reservation system. Such transaction processing systems may run as subsystems under a time sharing system - in this case the user does not interact directly with the system but with the transaction processing program.

Neither of these interfaces cater for the casual user who has no specialised training yet it is almost certain that, as applications diversify, there will be more and more users of this type. The computer system will become an everyday tool in the same way as the pocket calculator has done in the past decade. We believe that existing general purpose interfaces cannot satisfy the needs of the majority of casual users because of their terseness, their specialised terminology and their unsympathetic reaction to user error.

The range of users and their applications is so diverse that we do not believe that any single user interface can satisfy all classes of user. An interface designed to be verbose and satisfying to secretarial staff say, is irritating to programming staff. A terse "computer oriented" interface is confusing and frightening to secretarial staff yet "natural" for a programmer. Each user ought to be provided with a system interface couched in familiar terminology rather than in machine terminology. For example, a secretarial system should know about letters, memos, reports and other documents commonly handled in the office. The secretaries should not be forced to think in terms of workfiles, directories, file identifiers and other terms derived from computing jargon.

Eason et al[2] have studied man-computer interfaces and have identified user reactions to an unsatisfactory interface with the computer. Depending on the status of the user, these reactions can be classified as follows:-

(1) The user refuses to make use of the computer system. After initial "testing", he deems it inadequate for his needs.

(2) The user learns a few commands "parrot fashion" and always uses these when interacting with the computer system. Valuable facilities are not used because they are not understood.

(3) The user interacts with the computer system via an intermediary. This mode of interaction was the only one possible with batch systems but it hardly utilises the potential of interactive systems if individuals cannot use the system directly to satisfy their requirements.

We believe that the only way to make computer systems acceptable to many diverse users is to provide each user or class of user with their own system interface, reflecting the work which they normally do. The provision of such a system with many different user interfaces requires careful analysis of the needs of each class of computer user and the implementation of a tailored interface suited to those needs. If this analysis is not carried out, most users will react as described above.

We have not tackled the problem of interface design. Rather our work has involved the construction of software tools to aid interface implementation. In the next section of this paper we describe a system which allows a large class of tailor-made interfaces to be constructed at reasonable cost. This is followed by a description of a systems architecture geared to supporting multiple interfaces and finally, we draw some general conclusions about our system and associated architecture.

CONSTRUCTING TAILOR-MADE USER INTERFACES

A number of distinct types of user interface with a computer system can be identified. These include interfaces built around terminals with special function keys such as those used on many contemporary word processors, graphical interfaces using a light pen or tablet, such as those used in CAD systems, and imperative interfaces where the user types commands. These commands are interpreted by the computer system. Our work is geared towards providing a system which will allow new imperative interfaces to be generated for existing or proposed systems.

We make the assumption that there exists some general purpose for some particular system which allows access to the full range of system facilities. Typically, this interface allows access to facilities without safeguards, it is terse, oriented towards the computer specialist, and frequently unreadable. Error messages are couched in system terms.

To use a programming language analogy, forcing users to work with this general purpose interface is comparable to forcing programmers to work exclusively in assembly code. Our proposal is for "high-level" user interfaces which conceal much of the details of the system and allow the user to express his demands easily. Each different user interface is comparable to a high-level language and it is translated into the low level interface before being interpreted.

Implementing the user interface as a "high-level language" offers the same advantages as do high level languages over assembly code. Systems are easier to use, more understandable and more portable. If the underlying system changes, the user interface compiler can be rewritten so that the user is not presented with a new set of commands to learn. Obviously this is only true if comparable systems have comparable facilities and we see a need for low-level interface standardisation for each class of system such as relational data base systems, operating systems, reservation systems etc etc.

Clearly each user interface requires its own separate "compiler" and we have developed a software tool which is effectively a "compiler-compiler" permitting low-cost generation of user interface translators. This interface generator allows the user to specify a user interface along with associated system commands. The casual user can then use this interface and a general purpose translation program converts his commands to the appropriate system commands. As well as this, the interface translator can convert system replies to a form which is more sympathetic to the user.

Briefly, our system operates as follows. The system designer specifies the user interface in the formal notation illustrated, by example, below. This interface specification is input to a table generator program which outputs a set of tables representing the user commands along with associated system commands and replies. These tables are input to a generalised terminal processor program which accepts user commands, looks up the appropriate table and generates the associated system command. This is immediately passed to the operating system or application program. The reply generated from the user commands can also be detected by the terminal processor, looked up in a table of possible replies and translated into a form understandable to the user.

The distinguishing feature of our system which makes it possible to specify complex and informal user interfaces is the incorporation of a pattern matching system which is used to identify user commands[3]. This pattern matching system has been implemented on PDP-11, VAX and Z80 computers and offers significantly more power than that available via regular expression matching or even SNOBOL4[4], a widely used pattern matching language. The pattern matching facilities make it easy to specify flexible user interfaces where synonomous commands can be defined as having exactly the same meaning. We can also build interfaces where the meaning of a word within a command is dependent on its context - there is no requirement for continual uniqueness - a frequently aggravating feature of computer systems. These facilities are of immense value to the casual user who, if the interface is properly designed, can type anything sensible into the system and it will respond with a sensible reply. There is no need for memorisation of exact command syntax or formats.

The interface definition is specified as a sequence of patterns, one pattern for each user command. These patterns are translated into the low level form used by the pattern matcher and output as a set of tables. The terminal processor reads in these tables and, when a user command is input, calls the pattern matcher to identify the command in the table of patterns. If the user command is found, the appropriate system command is then selected from a corresponding table and issued.

Similarly, when the terminal processor receives a reply from the system, that reply can be looked up in a table of possible replies and the user reply detected and issued.

In a paper of this length, it is not appropriate to provide a full formal specification of the notation used to define the user interface. Rather, we present a number of examples which illustrate the power provided by the pattern definition facilities.

An interface definition has three distinct parts:-

(1) The definition of pattern names and specifications. This is accomplished using a let declaration.

(2) The definition of action procedures. These are procedures, coded in the language of the interface translator which carry out run-time actions in the interface translation. Within these procedures, reference may be made to pattern names defined using a let declaration. Action procedures may return a pattern, and this pattern can be associated with a name defined in a let declaration.

(3) The definition of the user commands, the associated system commands, and system replies and associated user replies. These commands are defined using a define declaration and within this declaration reference may be made to pattern names and action procedures.

An example of a let command might be the definition of a pattern representing a file name:-

```
let file = [a-z]#0-13
```

This specifies a pattern which must start with a letter and which may consist of a letter plus any number of characters up to 13. The pattern [a-z] matches any single character in the range a to z, the pattern #0-13 matches any string of characters from length 0 to 13, naturally always matching the longest possible string. Valid file names therefore might be x, myfile, interface.fil, and this.document.

Of course, the patterns declared in a let declaration may simply be character strings:-

```
let directory = /users/secretaries
```

This definition associates the name directory with the string "/users/secretaries".

If pattern names are used in the definition of other patterns, that is on the right hand side of let declarations, the name must be enclosed in angle brackets to inform the translator that this is a pattern name rather than a simple string:-

```
let fullname = <directory>/<name>
```

Essentially the let declaration normally defines a macro name and the table generator program substitutes the "macro body" for that name each time it is encountered on the right side of a declaration. Therefore the above declaration becomes:-

```
let fullname = /users/secretaries/[a-z]#0-13
```

The power of the pattern matcher is such that the user can define patterns as a sequence of alternatives or as "anything up to some delimiter". For example:-

```
let edit command = edit|editor|change|modify|correct
```

This specifies that the name edit command should match the strings "edit", "editor", "change", "modify", "correct".

The let declaration:-

```
        let up.to.by = (by)@
```

defines a pattern which will match anything up to but not including the string "by". The special symbol $ stands for "end of command" so

```
        let the.rest = ($)@
```

specifies that "the.rest" should match everything up to the end of the current command.

The let declarations can introduce patterns which are context sensitive:=

```
        let subject = <noun>!<verb>!
```

In a system where <noun> and <verb> ar defined as patterns, the pattern "subject" matches <noun> only if it is followed by <verb>. Conversely, we can specify that a pattern should only match when it does not appear in some particular context:-

```
        let object = <noun>!~<verb>!
```

Therefore if we had simple commands made up of nouns and verbs we could identify the appropriate nouns representing the subject and object of the command.

Action procedures are specified by the interface definer in the programming language of the interface translator. In this implementation, this is a programming language called S-algol[5]. They are exactly as any other procedures in that language except that reference may be made within these procedures to the pattern names defined in a let declaration. A preprocessor scans the procedure code, replaces references to pattern names by strings representing the pattern and once, the procedure has been completely preprocessed calls the S-algol compiler to check the syntax of the procedure. If an action procedure returns a value, it must be returned as a string, and the returned string is considered to be a pattern definition.

The example below illustrates a simple action procedure which asks the user for his name. The name is returned as a string and the action procedure may be called in a let declaration to assign that string to some name.

```
        action procedure get.name(->string)
        begin
             write "hello, please type your name'n"
             read.a.line(s.i)
        end !getname
```

This action procedure would be called as follows:-

```
        let user.name = action get.name
```

The string returned by get.name would be named by user.name. When a let declaration includes the word action, the system associates a "don't yet know" value with the string user.name. On the first reference to user name in a define statement, a call to the action procedure is filled in to find the actual user name.

The user interface being defined is specified using define declarations. A define declaration has three parts - the user command, the associated system command and an optional reply part which defines the system replies and the corresponding user translations of these replies. For example:-

```
          define <edit command> as
                ed workfile
          end
```

Again, if pattern names are used, they must be enclosed in angle brackets. This declares that <edit command> is translated into "ed workfile". However <edit command> matches a number of strings so that the user commands "edit", "editor", "correct", "change" and "modify" are synonomous and all have the same effect.

Naturally user commands may be translated into a sequence of system commands. For example:-

```
          define get <file> as
                cat <directory>/<file> >workfile
                cat workfile
          end
```

This defines a get command which makes the named file the current workfile and lists it on the user's terminal. Notice that in the case, the user types a simple file name and it is automatically converted to the full name of the file.

A reply part may be associated with a define statement. In the reply part, the possible system responses are listed along with their translations. For example:-

```
          define <edit command> <file> as
                ed <directory><file>
          and reply:
          "no file"     : " I am sorry, I can't seem to find your document
                           Could you please check that the name
                              is correct"
          "no permission"       : " I am sorry this is a private document and
                                     you may not modify it"

          end
```

In each case, the terse system reply is translated into a form less likely to intimidate the casual user.

If an action is to be associated with a reply, the action procedure must be provided and a procedure call included in the define declaration. This permits a more sophisticated response to certain replies than simple message translation. For example:-

```
          define <edit command> <file> as
                ed <directory>/<file>
          and reply:
          "no file"     : "I am sorry, I can't seem to find your document";
                        action findnearest(<directory>,<file>)
          end
```

if "no file" is returned, a message is output and the routine findnearest called. This routine takes as parameters the directory and the file and perhaps may search the directory for the closest match to the specified file and suggest to the user that this file might be the file intended.

These examples illustrate the potential of the user interface generator and we are presently evaluating the system by generating a secretarial interface to a Unix time-sharing system. This interface will allow access to word processing, information retrieval and electronic mail programs available under Unix.

The present s
communicating
providing inte
system is not
we envisage th
overhead invol
following sect
multi-interfac
the existing i

SYSTEM ARCHITE

As micro com
appropriate i
interface tra
dedicated to
requirements.

We assume tha
operating or
which transla

Not only does
hence increa
developed fo
Further, beca
circumstances
reduced user

Providing lo
system securi

(1) Th
hi
bu
pr
fa

(2) T
lo
ac
in
la

(3) I
de
fa
n

(4) T
t
v
t

Of course,
provided b
interprete
terminals.

(1)

The present system involves implementing the interface translator as a process communicating with both the operating system and the user terminal. The overhead of providing interface translation is acceptable in our existing system. If the system is not heavily loaded, the user response time is roughly doubled. However, we envisage that if we were to try and support many different user interfaces, the overhead involved in a conventional implementation would be unacceptable. In the following section, we describe an alternative system architecture for supporting a multi-interface system which we consider offers many advantages when compared with the existing implementation technique.

SYSTEM ARCHITECTURE

As micro computer hardware is now relatively cheap, we believe that a more appropriate implementation of a multi-interface system is to distribute the user interface translation either directly to the user's terminal or to a microcomputer dedicated to a number of terminals used by a group who have similar interface requirements.

We assume that a general purpose "computing oriented" interface is provided by the operating or application system. The users terminal contains a preprocessor program which translates the user command to the appropriate sequence of system commands.

Not only does this approach reduce the load on the central computer system, and hence increase overall system performance, it also means that interfaces can be developed for existing systems without disrupting these systems in any way. Further, because some intelligence is distributed to the terminal, under certain circumstances, such as data entry, it may be possible to provide some kind of reduced user service if the central system is unavailable.

Providing local interface processing has a number of important implications for system security and data privacy:-

(1) The user may be permitted access to those system facilities relevant to his application and no others. This can be accomplished simply by only building knowledge of appropriate system facilities into the interface processor. It should be impossible to generate commands using forbidden facilities.

(2) The terminal processor used to handle a particular interface can be located in an office where only those permitted to use that interface have access. Any unauthorised user can easily be detected by the individuals in that office and the system can be lock up when the office is unused. A locked door is a potent weapon for deterring snoopers.

(3) If an unauthorised individual gains access to the system, he will be deterred from accessing private information simply because he he is not familiar with the user interface. There is no need for users who do not need to use particular facilities to know how to access these facilities.

(4) The user validation system may be distributed to the terminal interface so that there is no need to keep a centralised on line record of user validations. Therefore, it is impossible for unauthorised users to crack the centralised validation system from any terminal.

Of course, under existing time sharing systems, a multi interface system can be provided by implementing different user command interpreters and running these interpreters as processes serving the appropriate user terminal or group of terminals. It appears to us that this approach has two important disadvantages:-

(1) It imposes an extra load on the central system because extra processes

have to be supported and serviced.

(2) It is expensive to modify existing operating systems or applications systems to provide a number of new interfaces.

As well as these absolute disadvantages, implementing the system under an existing operating system does not offer the security and privacy advantages provided by a distributed system.

To sum up therefore, we believe that the best way to implement a multi-interface system is to provide local facilities for translating particular interfaces to more general system calls. This can be accomplished either by using a programmable terminal or by using a microcomputer as a controller for a group of terminals.

CONCLUSIONS

In the course of development, it has become clear to us that defining suitable well-tailored interfaces is a very difficult task indeed, simply because of the communication problems which exist between computer specialists and those unversed in computing terminology. We believe therefore that the most formidable obstacle to the development of tailor-made interfaces is neither hardware nor software capability but simply the human problems involved in ascertaining what constitutes a well designed interface.

Our system makes no attempt to solve these problems. Rather, we have implemented an interface translator generator which can significantly reduce the work involved in developing imperative user interfaces for existing or proposed systems. This reduction is particularly marked where the user interface is complex and/or fairly informal.

The present implementation of our system appears to work reasonably well but it has become clear to us that further development is required. Our system relies on user errors being detected by the underlying system and an error message returned. This message is then translated to a form understandable by the user. Whilst this technique is frequently successful, there are situations where it breaks down, particularly if one user command generates many system commands. Identifying which command is associated with which error message is a difficult task. Further developments must therefore include more error checking and perhaps correction by the translator.

We believe that one of the most important results of our work is the identification of the advantages which accrue from distributing the user interface to the terminal itself. We intend to transport our system to a Z80 based system in the near future and use this Z80 as a terminal controller. With this system, we intend to investigate the extent of the privacy and security advantages offered by a distributed interface system.

ACKNOWLEDGEMENTS

Des Ward of Strathclyde University has done much of the coding of the system, Douglas Macgregor provided constructive criticism of this paper and Ron Morrison of St Andrews University provided and extended the programming language used in the project.

REFERENCES

1 Ritchie D.M. and Thomson K. 'The Unix Time-Sharing System', Comm. ACM 17,7,pp 365-375(1974)

2 Eason K.D.,Damodaran L.,and Stewart T.F.M. 'Interface Problems in Man-Computer

Interaction'. in
91-105. North-Ho

3 Sommerville
of Strathclyde,

4 Griswold R.E
Prentice-Hall, N

5 Morrison R
University of St

Interaction'. in E.Mumford andH.Sackman(eds), 'Human Choice and Computers', pp 91-105. North-Holland, Amsterdam, 1975.

3 Sommerville I. 'A Pattern Matching System' Dept of Computer Science, University of Strathclyde, Research Report 4/80.

4 Griswold R.E., Poage J.F., and Polonsky I.P. 'The SNOBOL4 Programming Language. Prentice-Hall, New Jersey, 1971.

5 Morrison R. 'S-algol Reference Manual', Dept. of Computational Science, University of St Andrews, Scotland. June 1980.