A Dialogue Development System for the Design and Implementation of User Interfaces in Ada*

J. ROBINSON AND A. BURNS
Postgraduate School of Studies in Computing, University of Bradford, U.K.

The need for tools to aid the implementation of user interfaces is highlighted. Techniques for the production of good user interfaces are outlined and facilities to ease implementation of interfaces designed using these techniques are suggested. Emphasis is placed on the need for multi-level adaptable interfaces, and the need for a separate, high-level specification of an interface. A dialogue development system intended for use with interactive systems written in Ada is proposed.

1. INTRODUCTION

With the growing use of interactive computer systems there is an increasing awareness of the importance of the user interface. This awareness, once almost totally the preserve of the academic community, is now, thankfully, extending into industry.[1]

The need for an improvement in the quality of user interfaces is without doubt. Unfortunately, the problems associated with the production of 'good' interfaces are also apparent and act as a serious deterrent to programmers who wish to provide 'easy to use' interactive software. The Ada programming language, together with its programming support environments, now promises to enable systems to be developed which will ease these problems considerably. However, before discussing such environments, it is first necessary to review the facilities required by a user of an interactive system.

The design of user interfaces is a subject which has received much attention from academics, and design methodologies have been proposed to ease this design process.[2] Whilst this area still presents some difficulties, the increased appreciation of user needs has led to a greater understanding of the techniques required to improve the design and effectiveness of interfaces.

Some of these, however, impose a heavy workload on the applications programmer. It is essential that programmers are allowed to concentrate on the development of the applications software whilst still being able to guarantee a good user interface. In many cases the reason for a bad interface has been the difficulties presented by the implementation of a better alternative. This problem has been recognised and new languages developed[3, 4] and existing languages extended[5, 7] to ease the programmers' task.

Burns[5] has highlighted the shortcomings of i/o facilities in existing languages. Whilst the design of new programming languages has made great strides in recent years, i/o facilities have remained largely unchanged. The fact that the definitions of C and Algol 60 omit any form of i/o perhaps demonstrates better than anything else the low priority which language designers have associated with i/o. The typical read/write statements, originally designed for batch work, fail to exploit the facilities provided by even the simplest of interactive display terminals.

* Ada is a registered trademark of the U.S. Department of Defense.

This paper discusses facilities intended to ease implementation of user interfaces. Emphasis is placed on multi-level, adaptable interfaces and the need for a separate, high-level specification of an interface.

2. FACILITIES FOR PROGRAMMING INTERACTION

2.1 Simulation of a perfect user

Interactive computer systems must be robust to user errors. A system which 'crashes' due to a user error is no better than useless. Fortunately, this is universally accepted and systems will usually carry out input validation.

It is attractive for an applications programmer to assume a 'perfect user' who supplies inputs in exactly the form expcted, removing any need for error trapping. The provision of i/o facilities which emulate a perfect user have been provided in Simula[8, 9] and Pascal.[5] These solutions, however, leave the programmer to supply suitable feedback. Whilst the programmer must have the option of providing messages, the user interface should be able to construct default messages for use when no text is available. Edmonds[10, 11] suggests that an error message can be seen as a transformation of the original input message, just as a transformation of the original message is passed on to the main system, the only difference being the destination. The proposed system, therefore, must be able to handle all user errors and, unless already specified, generate appropriate feedback. This simplifies the applications software whilst guaranteeing the user a robust and helpful interface.

It must be emphasized that we are proposing automatic generation of user feedback as a means of guaranteeing existence of feedback. In most cases manually created feedback will be far better than that generated by a tool. However, there may be cases where text based on information on the data object (e.g. range, type, etc.) is required, in which case an automatically generated message may well be sufficient. Consider an integer data object 'size', range 0..10, in the following dialogue.

```
PLEASE TYPE IN SIZE >> 11
INPUT SHOULD BE IN RANGE 0..10
PLEASE TYPE IN SIZE >> 10
```

An interactive system may rely completely on these generated messages in the early part of its development cycle, messages which prove to be unsuitable being replaced later by manually generated text.

Frequently users will require a backtracking facility, i.e. the facility to re-enter input. This can usually only be provided in a limited form as eventually a point will be reached where an action has been carried out which simply cannot be 'undone'. The proposed system, however, will provide some form of backtracking without this introducing extra work for the programmer.

2.2 Input/output format control

The use of text layout, font types, etc. is of considerable importance to any form of textual information exchange. Newspapers rely heavily on the use of different fonts and bold type to provide emphasis. These visual aids are used as a speaker would use gestures, facial expressions and changes in the tone of voice to provide emphasis and maintain interest;[33] the use of computer graphics in a similar manner is important. However, much work has been done in the development of user-friendly graphics[6] and it does not seem necessary to duplicate that work. However, the need for the easy specification of screen formats is clearly important.

Facilities of this type have been provided in existing tools and in extensions to existing languages,[7] illustrating the unsuitability of normal i/o for the implementation of modern interactive computer systems.

2.3 Interface specification

The user interface specification (UIS) should be completely separated from the applications software. Advantages of using this approach include:

(i) easy modification of the interface without unnecessarily affecting the applications software;

(ii) demonstration of the interface independently of this software;

(iii) the facility for the interface to be developed in isolation, perhaps by a user interface or domain specialist, provided the internal interface between the user interface and the applications software has been defined.

This complete separation is unusual. Traditionally, the UIS, if such a thing exists, will also contain information regarding the applications software. The need to tie the UIS with the application routines may influence the appearance of the interface, e.g. the grouping together of inputs to one routine, and will probably create more work if the interface, or the application routines, have to be changed.

The provision of a suitable user interface specification language (UISL) encourages this separation. The distinction between a specification language and an implementation language, and the ideal of collapsing these two languages into one, is made in ref. 12.

We hold the opinion that a UIS may also be used to implement an interface by providing an interpreter for the specification language, giving the language a dual role as a specification and programming language. A high-level specification language for the definition of forms for use in office information systems[15] included the use of the language as an implementation language.

It has been common in the past to consider the user interface of a system as the master and the system as the slave. Most tools have reflected this, typically by providing some standard interface on to which application routines are 'hung'.[16] This approach favours the use of state diagrams in the design phase,[13, 14] representing the system as a finite set of states which are traversed in response to user inputs. The specification of a state consists of a list of possible user inputs, possible paths from this state and application routines to be executed at that state.

We propose, however, that a user–computer dialogue is a 'mixed initiative' dialogue; i.e. the responsibility for control of the dialogue is distributed. With no permanent master/slave relationship, the user and system can be seen as two parallel processes interacting via the user interface.[17] Consider the following dialogue.

```
>> COPY
file 1 > FRED
file 2 > JOHN
file copied.
>>
```

When the prompt ' >> ' is given, control of the dialogue is transferred to the user, who may type any command available on the system. The path which the dialogue will take depends on this input. The user fails to specify the names of the files to be copied and the interface takes control to request the file names. Control is returned to the user when the ' >> ' prompt is issued.

To include dialogue paths in the UIS, all possible paths through the dialogue must be completely predictable at the design stage. If we wish to change the interface, e.g. moving a data object from one state to another, this prediction will be invalidated, possibly forcing extensive changes to the UIS. The UIS therefore should have no knowledge of possible paths through the dialogue. The UISL should have predefined rules for the movement of the dialogue from one state to another.

2.4 Virtual terminals

For the purposes of this discussion we are disregarding graphics terminals, and concentrating on the more common dumb and intelligent VDUs which are only capable of displaying normal text. These terminals have similar characteristics (e.g. reverse video, blink, etc.). However, the control character strings which operate these functions will invariably differ between terminals.

The UISL should make all references to the display device in terms of a virtual terminal. Several constraints must be made if this is to be successful, e.g. the virtual terminal must not impose unnecessary restrictions on terminals which may be connected to the system. Neither should the virtual terminal force a sophisticated terminal to have its capabilities under-used as a consequence of a restrictive set of 'virtual facilities'.

Since it would be impossible to produce a set of virtual facilities which would include all possible facilities in actual devices, it is necessary to have a virtual device which may be extended easily. These extensions must be achieved without affecting existing interfaces using the virtual terminal.

Criticisms of the use of virtual terminals can be found in ref. 18. We believe, however, that the concept of an extensible virtual device provides a powerful tool for user interface production.

2.5 Adaptable interfaces

Interactive systems attract a diverse range of users possessing a wide variety of computing experience. To provide an interface suited to the abilities of users, one must provide an interface less rigid than those currently in common use.

A dialogue between two people allows for the differing knowledge levels of the communicating partners. For example, a car mechanic involved in a discussion with an experienced colleague will use a terse dialogue using terms and phrases which would prove unintelligible to a new apprentice. When talking with a 'naïve' colleague, the mechanic will adapt to make allowances for inexperience.

Also, the human element of a system is not static and will adapt to a system or environment as experience is gained. As the apprentice mechanic gains more knowledge about the work, colleagues will adapt to allow for this increase in experience. If this adaptation did not occur the dialogue would become a source of irritation and inefficiency.

It is the inability of many interactive systems to adapt which causes user discontent. Whilst 'serious' systems have largely ignored this requirement, it is ironic that they should have found widespread use in far more trivial applications. In any amusement arcade you will see video games adapting to the abilities of their players. Some allow the user to choose a level of difficulty whilst others automatically adapt as the game progresses; the longer the game lasts, the better the player must be, and hence the harder the game gets.

It is recognized that a system should provide a number of different interfaces.[19, 20, 21, 22] There are several different user types, categorized on their past experience, and although there is currently no agreement on the number of dialogue levels, it is certain that a multi-level interface has a much better chance of pleasing its users.

An interface may implement a different type of interaction at each level or a different version of the same interface. Mozeico's graphics system[22] uses a five-level interface, three in a command language style, one a menu type question–answer dialogue and the other a tutorial frame-driven dialogue. Luker's Modeller[21] on the other hand retains a similar structure on each level, with the verbosity of the dialogue varying between levels.

Some users may be expected to progress through levels as their experience improves whilst others may be happier to stay with a known level. Research[23] has demonstrated that some users simply learn to use the system without understanding its operation. A user who shows an increase in understanding of the system will be likely to benefit from a move to a higher dialogue level, whilst a user who is learning by rote would suffer considerably by a change.

It is also common for a user to become an expert at part of the system whilst remaining a naïve user of other parts, hence requiring a mixed-mode interaction. Clearly users must be able to change levels as they wish.

It has been suggested[24] that an adaptable user interface must leave the control of the adaptation with the user. The user has a conceptual model of the system with which he/she is working and this could be disturbed by an interface which is dynamically adapting to the user. An obvious source of confusion would be change from a command-driven mode to a menu-driven mode when a user made mistakes at the command level. Modeller[21] forces a change in dialogue level when an error occurs at the 'expert' level. This change does not result in a change in mode since both levels use a similar style, and it seems to work well in practice.

The implementation of such a sophisticated user interface using conventional languages would impose an unbearable workload on the programmer, e.g. since the user will be allowed to swop between levels of interaction, it is important that the levels will not get out of step, i.e. a change of level could not result in a change in the state of the dialogue.

Often the users of a system will require a more subtle adaptation. A good example is 'parallel–sequential tradeoff'.[24] The user may enter commands singularly or several in sequence, allowing an experienced user to speed their interaction with the system, as they become accustomed to the required sequence of commands.

A widely used technique is to allow a mnemonic form of a command; e.g. a system may allow LIST, LIS, LI or L to refer to the same command, provided no ambiguity arose.

2.6 Help facility

Despite widespread acceptance of the importance of a help facility there are still interactive systems in existence which provide no such facility, and many more which provide one which is of little practical use.

It has been proposed in ref. 24 that the help facility should be adaptive. The 'query-in-depth' technique proposed by Gaines and Facey allows the user to expand on the information given by repeating the call for help. This allows an inexperienced user to gain access to a long and detailed help message whilst not forcing this verbose text on an experienced user. A disadvantage is that a naïve user must work through the terse messages before receiving the verbose text.

The help information issued should vary according to the current state of the dialogue. In a system with user levels this information should differ between levels. Given that different levels could implement different modes of interaction, these differences could be considerable. This system could also include a 'query-in-depth' style of adaptation within each level.

It is difficult to imagine the user view of the system and hence what kind of user feedback is required if the users are not allowed to influence system design. This involvement is made simpler if the help information may be demonstrated to the user.

2.7 Rapid re-configuration of the interface

The need for user involvement in the design process, and the easy adaptation of the system to changing user requirements throughout its life cycle, means a user interface must be quickly and easily modified.

It has been recognized that user involvement in the design and support of interactive systems is essential to their success.[11, 26] Eason *et al.*[25] highlighted the lack of user involvement. Users were split into three groups; managers, specialists and clerks. The survey showed that 45% of managers, 32% of specialists and 25% of clerks had been involved in the design of computer systems, yet over 70% of all users wished to become involved in the design of future systems. Proposals have been put forward (e.g. ref. 26) for design and support techniques to encourage user participation, and problems which may arise have been documented. Prospective users must have the opportunity of seeing a prototype of a proposed system, or at least its user interface, before agreeing to its introduction. Similarly, any changes to an existing system, or user interface, should be demonstrated.

Changes made to an existing system will typically be proposed by the users and will often be vague and/or impracticable,[27] with the users often having no idea of how these changes will affect the system. The ability to change an interface rapidly allows a specialist to demonstrate the effects of changes and to demonstrate alternatives. The effects of this are far-reaching: users are more likely to get an interface which they like and accept, and will be more likely to accept the introduction of a new system or the alteration of an existing one.

This leads us to two requirements.

(1) The user interface should be completely separated from the software behind it. Typically a user interface in an interactive system will be distributed throughout the systems software, making changes to the interface difficult and the provision of several different interfaces almost impossible.

(2) Where possible, changes to the interface must be carried out without re-compilation of software. The user interface will be closely linked with the applications software and some changes, e.g. range of an input variable, will also force changes to this software. Despite this there are many changes, e.g. screen formats, which may be carried out without these being propagated into the software. This problem is reduced by implementing the system in an interactive language.[4] It is unfortunate, however, that in many applications the use of an interactive language is either not possible, or simply not attractive. In this case the provision of tools to help speed the modification of the software when necessary will help considerably.

The advantages of constructing an interactive system in such a way have been sufficient to encourage the development of appropriate system construction tools.[34]

3. THE PROPOSED SYSTEM

The rest of this paper is devoted to proposing a dialogue development system (DDS), for use with the Ada language,[28] which is to be developed with the above comments in mind.

The general structure of an interactive computer system constructed using this DDS is shown in Fig. 1. The following sections outline the purpose of each component of the DDS and the way in which these components interact with each other.

3.1 The dialogue development system (DDS)

The DDS comprises a range of tools for the production and maintenance of user interfaces. The applications software will interact with different components of the DDS at different stages of its life cycle. These components will include the following:

3.1.1 Dialogue manager (DM). This is the central core of the DDS, and it is this component with which the applications software will interact during the time that the system is 'live'. The dialogue manager supervises the dialogue and implements the interface specified in the UIS, essentially acting as an interpeter for the UISL. This interpreter is, however, very sophisticated and handles

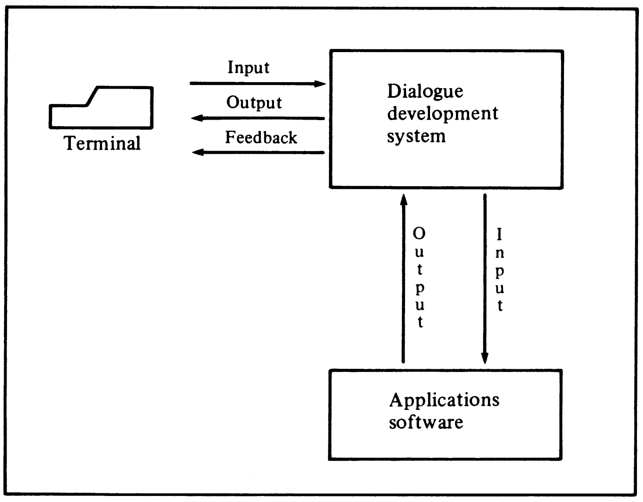

Figure 1

work, such as backtracking, input data validation, etc., which would normally have to be carried out by the applications programmer.

Our model of a user–computer dialogue treats the two partners in the dialogue as two concurrent processes. Since the DDS is intended for use with Ada, the dialogue manager will be implemented as a task in an Ada library and will run in parallel with the applications software using it. Hence the actual software will reflect the conceptual model of the system it implements.

To the applications software, the DM appears to be a pre-defined package containing a task to which calls are made for transferring data between itself and the user. The DM is transparent to the user. Subcomponents of the DM will include the following.

(1) *Feedback generator*. This generates appropriate user feedback, and initially will simply consult a set of standard messages which will be customized to suit the feedback required.

(2) *Adaptive interface handler*. This component controls adaptation of the dialogue. It will simply check user input and, when a request for change of level is received, will inform the other components of the DM that the level has been changed. Effectively this component will act as a filter in the input stream.

(3) *Buffered i/o handler*. This allows parallel–sequential tradeoff. If the data required has already been input the DM reads this directly from an input buffer. If this buffer is empty then the user is prompted for the input.

The initial versions of these components will be as simple as possible in order to allow a prototype of the system to be constructed. They do however open up interesting avenues for further development once the system has been shown to be feasible. For example, both the adaptive interface handler and the feedback generator could be replaced by expert systems.

3.1.2 Specification languages translator. This is essentially the front end of the DM and will convert a UIS written in the UISL into an intermediate form. The prototype system will be built around a subset of the UISL for which a translator has been constructed.

3.1.3 Virtual terminal. The virtual terminal is a device- and machine-independent Ada task which transforms device-independent instructions produced by the dialogue manager into device-dependent strings of control characters.

The virtual terminal software has no knowledge about the terminals with which it communicates, but uses a database of information about the types of terminals which may be connected to the system. Adding a new terminal type should require nothing more than an extension of the database with information regarding this terminal's characteristics. This divorcing of device information from the virtual terminal software is to aid the adaptability of the virtual device to new terminals which are added to the system.

This software can be looked on as the database handler for the terminal description database, with the extra function of checking all tokens sent from the dialogue manager, and deciding whether these are passed directly on to the real terminal or, in the case of control information, replaced by control character strings before passing on to the actual terminal.

3.1.4 Validator/system configurator. The validator is responsible for checking the interface between the applications software and the UIS. This is not checked by the UISL translator since this internal interface can become quite complex and as such is better managed by a separate component.

There is a need for a tool that will configure a 'production' system from its constituent parts. This tool may include the validator as a subcomponent.

3.1.5 Screen formatter. An interactive screen formatter will be available to enable the construction and modification of screen formats to be achieved easily and quickly.

Whilst the specification language is undoubtedly the best way to approach the specification of textual dialogues, the specification of menus, etc. is not so straightforward. It is in this situation that the screen formatter should be able to interact with the UISL translator. The screen formatter will have an extensible database of screen formats which are 'instantiated' by the formatter. These formats are stored in the database in the form of 'stencils', i.e. a basic outline of a screen which will require parameters to be supplied to fill in the gaps left in the stencil specification. The formatter will have an interactive front end for the specification of these stencils.

The instantiation of screen formats will essentially be a macro call from the UIS, i.e. the format instantiation mechanism is not actually part of the UISL, and hence not part of the functions of the UISL interpreter, but will be replaced by UISL statements.

3.1.6 System monitor. Once a system has gone live, there is a need for constant analysis of the systems performance and how this affects the performance of its users. The system monitor will record appropriate data, e.g. the location of points in the interface where users encounter problems with the dialogue. These data may then be analysed to assess the systems performance, and changes and enhancements can be suggested based on these results.

3.2 Applications software

The applications software is defined for the purpose of this document as that software which implements the functions of a system.

All communication between this software and the user is made via the dialogue manager. The interface between the applications software and the DM is strictly defined in the UIS to enable changes in this internal interface to be carried out easily and with as little delay as possible. The applications software deals only in the basic units of i/o, e.g. integers, characters, strings, etc., and the basic operations 'get' and 'put', which transfer these objects to and from the dialogue manager.

No validation of user inputs needs to be carried out by the applications software, this responsibility being carried by the dialogue manager. The applications software also has no knowledge of the interface presented to the user. The type of interface, prompts, user feedback, etc. are transparent, thereby separating the applications software from the user interface and rendering it almost immune from interface reconfigurations. The only changes which will affect the applications software, such as changes to the range of an input or output object, can be detected at the interface between the dialogue manager and applications software and the necessary alterations made.

The applications software which may use the DDS proposed here will be written in Ada and will run in parallel with the dialogue manager.

Ada was developed primarily as a language for embedded systems and its facilities naturally reflect this. Due to the widespread publicity it has received, however, the language has attracted attention from many other areas of computing and has shown itself to be suitable for application to fields other than real-time systems (e.g. ref. 29).

The choice of Ada, not only as the language to be used with the DDS, but also as the implementation language of the DDS itself, was made due to several factors. The language provides an excellent medium in which to work, supporting as it does state-of-the-art programming language features. In particular, the facilities for packages, tasks and separate compilation make the language eminently suitable for this work. Perhaps the most important advantage of developing the DDS for, and in, Ada is the proposals for the development of Ada Programming Support Environments.[30] Clearly, the DDS would form the central hub of an APSE for dialogue and interactive system development. It is with this in mind that the DDS will be designed and implemented in a manner complying with the requirements for APSE tools as laid out in ref. 30.

3.3 Relationships between tools

The relationships between components of an APSE are clearly important and are often complex.[35] For example the relationship between the Applications Software and the UIS, given that the requirement of independence of the interface and software is to be fulfilled, is far more complex than would at first appear. This relationship also involves the DM, which is responsible for managing communication between the applications software and UIS.

Tool interfaces are usually specified in terms of Ada package specifications. However, the relationship between tools often cannot be fully specified using this facility alone and extra information, in the form of data formally specified and contained in files, will be required.

Moreover, in some cases two tools will carry out all communication via a data file. The DM and UISL interpreter, for example, communicate via the intermediate code of the UISL, putting the file containing this intermediate code in the position of defining the interface between these two tools. The following sections outline some of the files which will be present in the DDS and which will be involved in interface specifications.

3.3.1 Dialogue specification file. The dialogue specification file contains the specification of the user interface. This specification will, of course, be converted to some intermediate code by the UISL translator and it is this form which will, in effect, specify the interface between the DM and UISL. In fact the information contained in this file may not be sufficient to specify this interface, and work is being carried out to develop a method of completely defining this interface. More complicated however is the role which this file plays in the relationship between the applications software and the DM; this is however beyond the scope of this paper.

3.3.2 Terminal description database. The terminal description database contains the characteristics of each type of terminal used on the system. The database associates the device-independent terms of the dialogue manager with the actual control character strings which will implement that function on the particular terminal in use, e.g. change to reverse video.

A similar, if rather limited, database is already available on the UNIX* operating system[31] and similar database may exist on some other operating systems, although certainly not all operating systems have such a facility.

This file does not specify a relationship between two tools but between a tool and an outside entity i.e. external to the APSE, in this case a VDU.

4. CONCLUSION

We have highlighted the need for an improvement in user–computer interfaces, and the requirement for the provision of tools both to aid the applications programmer in the development of such interfaces, and to encourage the use of techniques for the provision of user-friendly and adaptable user interfaces.

A dialogue development system intended for use with the Ada programming language has been proposed. This will be designed and implemented at the University of Bradford on a DEC VAX 11/750 running the UNIX operating system.[32]

Acknowledgements

The authors gratefully acknowledge the financial support of the Science and Engineering Research Council and the University of York, U.K. for the provision of an Ada compiler.

* UNIX is a registered trademark of Bell Laboratories.

REFERENCES

1. *Computing*'s user friendly competition. *Computing* **10**, 44, 30–39 and 15 (November 1982).
2. B. Schneiderman, Human factors experiments in designing interactive systems. *Computer*, **12**, 12, (1979).
3. A. I. Wasserman, Design Goals for Plain. *Proceedings of the 11th Hawaii Conference on Systems Science* **1**, 60–70 (1978).
4. B. R. Gaines and P. V. Facey, Basys – a language for processing interaction. *Proceedings of the Conference on Computer Systems and Technology, IERE 36*, pp. 251–262, (1977).
5. A. Burns, Enhanced input/output on Pascal. *ACM Sigplan Notices, Dec.*, (1983).
6. W. M. Newman and R. F. Sproull, *Principles of Interactive Graphics*. McGraw-Hill Computer Science Series (1979).
7. J. M. Lafuente and D. Gries, Language facilities for programming user-computer dialogues. *IBM Journal of Research and Development*, **22**, 2 (1978).
8. M. Ohlin, Safe conversational Simula programs using class, SAFEIO. Association of SIMULA Users, *Third SIMULA Users' Conference and Educational Workshop, Brighton, 1975.*
9. R. J. Orgass and R. E. Porter, Dialog. A Simula class for writing interactive programs. Virginia Polytechnic Institute and State University technical memorandum no. 79–3a, (1979).
10. E. Edmonds, Adaptable man/machine interfaces for complex dialogues. *Proceedings Eurocomp 1978.*
11. E. A. Edmonds, Adaptive man–computer interfaces. In *Computing Skills and the User Interface* (ed. M. J. Coombs and J. L. Alty).
12. Specification of dialogue and interactive programs. In *Methodology of Interaction* (ed. R. J. Guedj *et al.* 1980).
13. E. Denert, Specification and design of dialogue systems with state diagrams. *International Computing Symposium, Liège, Belgium* (ed. E. Morlet and D. Ribbens), (1977).
14. D. L. Parnas, On the use of transition diagrams in the design of a user interface for an interactive display system. *Proceedings of the National ACM Conference*, pp. 379–385, (1969).
15. N. H. Gehani, High level form definition in office information systems. *The Computer Journal*, **26**, 1 (1983).
16. B. Negus, M. J. Hunt and J. A. Prentice, Dialog: a scheme for the quick and effective production of interactive applications software. *Software, Practice and Experience* **11**, 205–224 (1981).
17. W. M. Newman, Languages for describing interactions. In *Display Use for Man–Machine Dialogue* (ed. W. Handler and J. Weizenbaum) (1971).
18. Portability and device independence. In *Methodology of Interaction*, (ed. R. A. Guedj *et al.*) (1980).
19. I. A. Newman, Personalised user interfaces to computer systems. *Proceedings Eurocomp 1978.*
20. R. L. Wexelblat, Design of systems for interaction between humans and computers. In *BCS 81, Information Technology for the Eighties* (ed. R. D. Parslow) (1981).
21. P. A. Luker, Computer-Assisted modelling of continuous systems. Ph.D. thesis, University of Bradford, U.K. (1981).
22. H. Mozeico, Human/computer interface to accommodate user learning stages. *Communications of the ACM*, **25**, (No. 2) (1982).
23. T. C. S. Kennedy, Some behavioural factors affecting the training of naïve users of an interactive computer system, *International Journal of Man–Machine Studies*, pp. 817–834 (1975).
24. B. R. Gaines, The technology of interaction – dialogue programming rules. *International Journal of Man–Machine Studies* **14**, 1, 133–150 (1981).

25. K. D. Eason, L. Damodaran and T. F. M. Stewart, *A Survey of Man–Computer Interaction in Commercial Applications*. SSRC report HR 1844/1 (1974).
26. L. Damodaran and K. D. Eason, Design procedures for user involvement and user support. In *Computing Skills and the User Interface* (ed. M. J. Coombs and J. L. Alty) (1981).
27. D. J. Cairns and J. J. Florentin, Human factors and program flexibility. *BCS'81, Information Technology for the Eighties* (ed. R. D. Parslow).
28. J. D. Ichbiah *et al.*, *Reference Manual for the Ada Programming Language*. United States Department of Defense (1983).
29. J. A. Kirkham, A. Burns and R. J. Thomas, *The Use of Structured Systems Analysis in the Rapid Creation of Information Management System Prototypes Written in Ada. Ada Letters*, **4**, 1 (1984).
30. D. A. Fisher, *Requirements for Ada Programming Support Environments – Stoneman*. United States Department of Defense (1980).
31. UNIX Programmer's Manual (seventh edition, Virtual VAX-11 Version) (June 1981).
32. M. Banahan and A. Rutter, *UNIX – The Book*. Sigma Technical Press (1983).
33. T. C. S. Kennedy, The design of interactive procedures for man–machine communication. *International Journal Man–Machine Studies*, **6**, 309–334 (1974).
34. G. Ringland, Introduction to developing interactive systems for the small machine environment. *Proceedings of the Conference on Computer Systems and Technology*, IERE 36, pp. 241–250 (1977).
35. A. Burns and J. Robinson, Tool requirements for the implementation of user interfaces in Ada. *Ada U.K. News* (Jan. 1984).