Exhibit 1017 – Part 3

they construct for achieving the goal and subsequent subgoals. The procedures represented at this level of the TKS provide a description of the rules task performers would expect to follow, and the alternative procedures they would follow under particular conditions. A task performer's plan at this level of representation is related to our notion of task structuring where certain task components precede, prime or follow one another. Task structuring determines the sequence or ordering of procedures necessary for successful task execution. This information provides the system designer with a view as to how people structure their tasks under certain circumstances: it also allows the system designer to decide how the user will expect to make use of the objects and actions (functions) and identifies the most frequent or preferred procedure for achieving a subgoal. This information can be used to set up default modes of operation in the program design.

Finally, at the object and action level, the taxonomic substructure identifies the representative actions and objects within the domain and the features of those task elements. The work on concept and object knowledge of Rosch and her colleagues (Rosch *et al.*, 1976) leads to the suggestion that if the designer chooses to support this task and provide a visible representation of the objects and the actions that can be carried out on those objects, then the taxonomic substructure provides an informative and detailed description of the features a person will expect to associate with those objects. Additionally, the degree of representativeness of task objects and/or actions, and the centrality of the procedures containing those components (actions and objects), provides the designer with an indication of which are deemed to be typical and necessary for successful task completion. The consequences of overlooking both central and representative task components in system and user-interface design are likely to have severe consequences for the ultimate usability of the system.

5.6.2 Empirical support for TKSs

Empirical evidence for the improvement in usability afforded by modelling TKSs is provided by Davis (1988) in a pilot study of graph and table drawing. The first part of the study identified representative objects and actions, central procedures and sequencing of task procedures for the above 2 tasks across a population of 12 subjects. An experiment was then carried out in which 3 further groups of subjects were required to undertake graph and table drawing tasks using 1 of 3 different interfaces with the same underlying functions.

One interface was structured so that it positively supported representative and central task components, and task sequencing identified by the TKS modelling stage. The second interface was unstructured; representative and central task components were supported but representative objects were not identified with their associated actions and the sequencing of task procedures was not supported. A further control group had an interface which contained neither central nor representative task components, and which had no explicit task structure.

The results of the pilot study showed that subjects found the structured interface easier to use, and also this interface had a higher preference value from subjects. Additionally, the structured interface resulted in quicker task execution, and the resulting graph and table drawings were better quality in that they were more complete. Also, the unstructured but central and representative interface design produced better performance than the control group interface, but less than the structured interface group. The findings thus support the theoretical view that TKSs provide important information about users which can be used to design improved user interfaces.

5.7 Relating KAT to design practices

Software systems design occurs in many different ways, resulting in a certain reluctance on the part of academics and industrialists alike to speak of an ideal design process. However, it is becoming increasingly clear that task analysis has a part to play in current and future design practices.

In this section, the contribution KAT might make to current design practices is briefly considered. It is not, however, the intention to argue for the use of KAT in specific design methodologies, structured or otherwise, since KAT is potentially appropriate to many current design methodologies and practices.

The traditional system development life cycle described in Chapter 1 typically involves the following stages. First, a feasibility study is carried out to establish whether it would be possible to build a system to support users' tasks, and if there is a market for such a product. After the feasibility study has been completed, a requirements specification would be prepared, followed by the design of the system. The system design, determined by the nature and the content of the requirements set out in the requirements definition, is then implemented, the implementation is tested and the system subsequently released. After a period of time in use by the customer the system is updated and maintained.

We can envisage a scenario where task knowledge requirements identified by KAT could augment existing user requirements in the software (and/or hardware) design life cycle. First, we would expect user requirements to be taken into account in any feasibility study. This would involve a small-scale task analysis. Using KAT at this stage identifies commonalities across tasks through within-role relations and also by the identification of generic task elements. At the requirements definition stage a full-scale task analysis using the KAT methodology would be carried out to establish and document user task requirements in terms of users' plans, goals, subgoals, strategies, procedures and representative and central actions and objects.

The results of a TKS model can be easily decomposed into general, specific and interface design models, as shown by Johnson *et al.*, (1988), where the KAT methodology was used to produce frame-based representations of a messaging system, the virtual interface to that system and the dialogue structure.

KAT may also play a role in usability and learnability evaluation, before and/or after the construction of a prototype or full implementation. The use of KAT in evaluation relates to whether aspects of a person's task knowledge, identified by KAT, have been carried over into the designed, prototyped or fully implemented system. This specifically involves finding out whether all appropriate tasks have been supported; whether generic task elements have been taken into account; whether representative and central task actions and objects have been represented; whether sequencing of task procedures have been supported rather than violated; and finally, whether defaults have been correctly specified and supported.

If these factors are taken into account the user will be expected to be in a position to transfer appropriate extant knowledge to the newly created environment and as a result the system designed will be easier to use and learn. Furthermore, predictions can be made as to where this transfer will be unsupported, whether interference is likely to occur, and in which areas training might be necessary.

5.8 Conclusions

In this chapter we have described a theory and method of modelling the knowledge people possess about tasks and roles in a given domain, known as knowledge analysis of tasks (KAT). The information contained within the task models constructed within the KAT methodology is very rich and can be used as an information source to which designers can be given access; it prevents the designer having to rely on his or her own intuitions about peoples' task knowledge. Empirical evidence suggests that the TKS models identify important features of knowledge that can influence the usability of systems when the design recommendations arising from a TKS are followed. Finally, the contribution made by KAT and TKS models to current design practices is considered.

Acknowledgement

The research described in this chapter was developed as part of an ICL-funded University Research Contract research project on 'The development of task analysis as a design tool'.

References

Annett, J. and K. D. Duncan (1967) 'Task analysis and training design', *Journal of Occupational Psychology*, **41**, 211–221.

Card, S. K., T. P. Moran and A. Newell (1983) '*The Psychology of Human Computer Interaction*', Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Davis, S. (1988) 'Knowledge structures in the human computer interface'. Unpublished manuscript, Queen Mary College, University of London.

Diaper, D. and P. Johnson (1989) 'Task analysis for knowledge descriptions: theory and application in training', in *Cognitive Ergonomics*, J. B. Long and A. Whitefield (eds), Cambridge University Press, Cambridge.

Fleishman E. A. and M. K. Quaintance (1984). *Taxonomies of Human Performance*, Academic Press, New York.

Galambos, J. A. (1986) 'Knowledge structures for common activities', in *Knowledge Structures*, J. A. Galambos, R. P. Abelson and J. B. Black (eds), Lawrence Erlbaum Associates, Hillsdale, NJ.

Graesser, A. C. and L. F. Clark (1985) *Structures and Procedures of Implicit Knowledge*, Ablex Publishing, Norwood, NJ.

Johnson, P. (1985) 'Towards a task model of messaging', in *People and Computers; Designing the User Interface*, P. Johnson and S. Cook (eds.), Cambridge University Press, Cambridge.

Johnson, P., D. Diaper and J. Long (1984) 'Tasks, skill and knowledge; task analysis for knowledge based descriptions', in *Human–Computer Interaction—INTERACT '84*, B. Shackel (ed.) North-Holland, London.

Johnson, P., H. Johnson and F. Russell (1988) 'Collecting and generalizing knowledge descriptions from task analysis data', *ICL Technical Journal*, **6**, 137–155.

Johnson, P., J. Johnson, R. Waddington and A. Shouls (1988) 'Task related knowledge structures: analysis, modelling and application', in *People and Computers: from Research to Implementation*, D. M. Jones and R. Winder (eds), Cambridge University Press, Cambridge.

Keane, M. and Johnson, P. (1987) 'Preliminary analysis for design' in *People and Computers*, D. Diaper and R. Winder (eds), Cambridge University Press, Cambridge.

Kieras, D. and P. Polson (1985) 'An approach to the formal analysis of user complexity', *International Journal of Man-Machine Studies*, **22**, 365–394.

Kelly, G. A. (1955) *The Psychology of Personal Constructs*, Norton, New York.

Leddo, J. and R. P. Abelson (1986) 'The nature of explanations', in *Knowledge Structures*, J. A. Galambos, R. P. Abelson and J. B. Black (eds), Lawrence Erlbaum Associates, Hillsdale, NJ.

Olson, J. R. (1987) 'Cognitive analysis of people's use of software', in *Interfacing Thought: Cognitive Aspects of HCI*, J. M. Carroll (ed.), MIT Press; Cambridge Mass.

Payne, S. J. and T. R. G. Green (1986) 'Task-action grammars: a model of the mental representation of task languages', *Human Computer Interaction*, **2**, 93–133.

Rosch, E. (1978) 'Principles of categorization', *Cognition and Categorization*, E. Rosch and B. Lloyd (eds), Lawrence Erlbaum Associates, Hillsdale, NJ.

Rosch, E. (1985) 'Prototype classification and logical classification: the two systems', in *New Trends in Conceptual Representation: Challenges to Piaget's Theory?*, E. K. Scholnick (ed.), Lawrence Erlbaum Associates, Hillsdale, NJ.

Rosch, E., C. Mervis, W. Gray, D. Johnson and P. Boyes-Braem (1976) 'Basic objects in natural categories', *Cognitive Psychology*, **8**, 382–439.

Schank, R. C. (1982) *Dynamic memory: A Theory of Reminding and Learning in Computers and People*, Cambridge University Press, New York.

Welbank, M. (1983) *A Review of Knowledge Acquisition Techniques for Expert Systems*, Martlesham Consultancy Services, British Telecom Research Laboratories, Ipswich.

7 Dialogue delivery systems: example research systems

PETER JONES

7.1 Introduction to dialogue delivery systems

In Chapter 4, part of the discussion was based on the idea of classifying dialogue styles. The main styles identified were *command language*, *menu selection*, *form-filling*, *natural language* and *direct manipulation*. In Chapter 6, an abstract model for a dialogue system was developed. Systems that implement the model are referred to as *user interface management systems* (*UIMS*). This chapter examines several dialogue delivery systems chosen to illustrate important aspects of these styles, which in reality may overlap.

In addition to being exemplars of the styles, the systems were also chosen to demonstrate a variety of approaches to the specification of the dialogue component of interaction. Command language is covered only briefly as it is assumed that most readers will be familiar with such interfaces. However, some recent work on extending command language interfaces is described. Next the *ZOG* menu- and frame-based system is described; then a form-filling metaphor is illustrated with *COUSIN*. The use of transition networks to specify a dialogue is shown in *Rapid/USE* and *CONNECT*. Natural language is introduced through some early work by Weizenbaum on the application of natural language to man–machine communication. Then its use both in medical interviewing and the *Mycin* expert system is described, leading on to the natural-language help system used in the *UNIX Consultant*.

The systems, which come mainly from the research environment, have been chosen to illustrate how actual implementations differ from the abstract model presented in Chapter 6. In general, the discussion is not detailed, but instead concentrates on bringing out what is seen as the important ideas embodied in the implementation. However, further details can be found in the references.

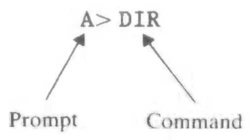


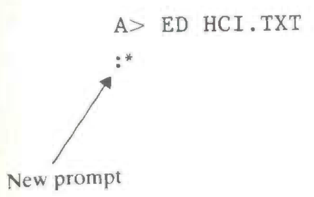
Figure 7.1 A CP/M prompt with a command

7.2 Command languages

Command languages are exemplified by the user interface to most operating systems (Beech, 1986). In the early days of computing, designers of such systems knew that the users were likely to be highly motivated experts and therefore concentrated on delivering the maximum functionality. Many early operating systems operated purely in batch mode, where the user presented a complete task with commands and data. Any feedback to the user was provided much later in the form of a hard-copy listing showing the progress of the commands, the data used and the output produced. Later operating systems allowed for more interaction with the user in order to control and monitor what was going on.

7.2.1 Digital Research CP/M

With the advent of the minicomputer in the mid-1960s and the microcomputer in the 1970s, more users began using computers interactively. From the user's point of view, the CP/M operating system from Digital Research was of great importance as the first general-purpose operating system to be widely used on microcomputers, and hence by the novice user. Its technical success stemmed from the fact that it was designed to be largely independent of the particular hardware on which it ran, and thus it established a large base of users and application software. Nevertheless, it had many awkward features (in common with many other command language systems): the prompts were rather cryptic, error messages were barely intelligible, there was an almost complete lack of help, and a crude command syntax with position-dependent arguments were used. For example, the prompt merely shows the currently selected disk drive, (Figure 7.1), while on switching to a subsystem, for example a text editor, the prompt changes, but with very little other feedback to the user (Figure 7.2).



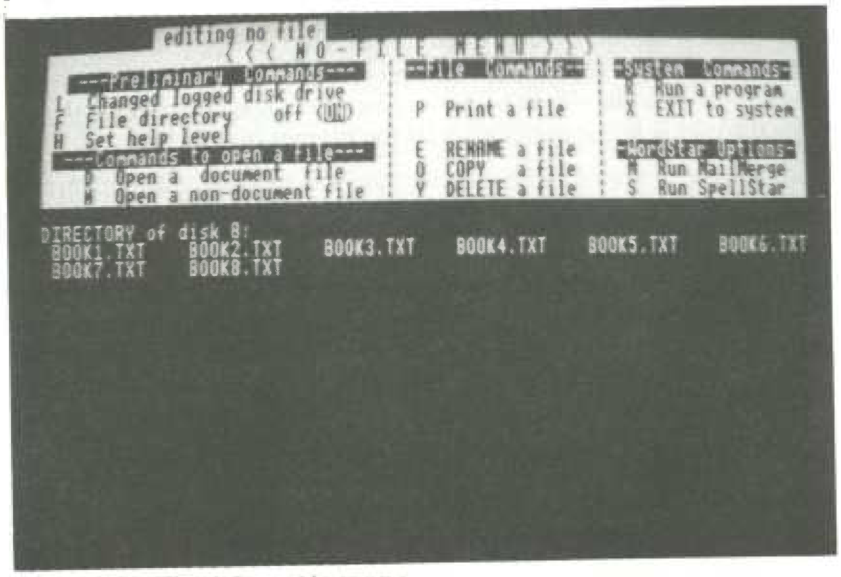
```
A> ED HCI.TXT
:*
```

Figure 7.2 Change of prompt in the editor

This style of interface was very poorly suited to the new and inexperienced computer users who were attracted to microcomputer systems by their low cost compared with previous generations of mini- and mainframe computers. Fortunately, the support provided for CP/M application programmers in the form of system calls and access to the file system made it possible for applications to provide their own, completely independent, support for common operating system functions. Thus successful end-user CP/M applications such as *Wordstar* (Figure 7.3) entirely replaced the basic CP/M command language with menu-based command selection and a form-filling mechanism for specifying filenames.

7.2.2 Dialogue Development System (DDS)

More recently, other approaches have been taken to increase flexibility in command language dialogues. At the University of Bradford, UK, a multilevel adaptable system, *Dialogue Development System* (*DDS*), was proposed for use with Ada (Robinson and Burns, 1985). As well as



```
     editing no file
            < < <  N O - F I L E   M E N U  > > >
 ---Preliminary Commands---   |  --File Commands--   |  -System Commands-
L  Changed logged disk drive  |                      |  R  Run a program
F  File directory     off (ON)|  P  Print a file     |  X  EXIT to system
H  Set help level             |                      |
---Commands to open a file--- |  E  RENAME  a file   |  -WordStar Options-
   D  Open a  document  file  |  O  COPY    a file   |  M  Run MailMerge
   N  Open a non-document file|  Y  DELETE  a file   |  S  Run SpellStar

DIRECTORY of disk B:
 BOOK1.TXT   BOOK2.TXT   BOOK3.TXT   BOOK4.TXT   BOOK5.TXT   BOOK6.TXT
 BOOK7.TXT   BOOK8.TXT
```

Figure 7.3 Wordstar main menu

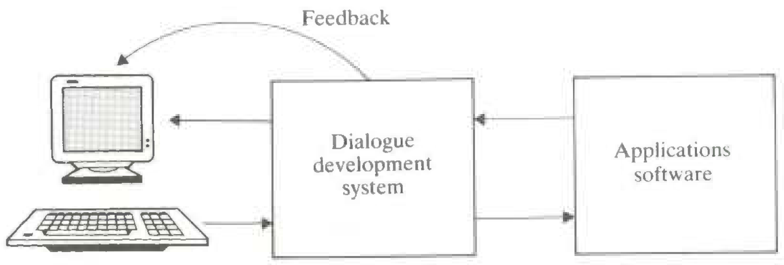


Figure 7.4 Proposed DDS system

emphasizing the adaptiveness required, this provided a separate high-level specification of the interface in order to relieve the application programmer's task of dealing with the interaction. A *user interface specification language* (UISL) was proposed to encourage this separation. Another key objective was to automate aspects of providing user feedback: for example, the type of an object could automatically provide for user feedback on erroneous input.

DDS is a range of tools that includes a *dialogue manager* (DM), which interprets the UISL, provides feedback, and controls the adaptation. A *terminal database* provides a virtual terminal that is device- and machine-independent. A *validator* is interposed between the application software and the DM to check the interface. As can be seen from Figure 7.4, the user interface system is run as a separate Ada task communicating with the other tasks forming the application. A *screen formatter* is provided to allow the construction of non-textual dialogues such as menus. Finally, a *system monitor* is used to provide a constant analysis of the system's performance and how this affects users.

Although a command-style interface is used, DDS provides for a mixed initiative interaction, for example prompting when arguments are missing. An example would be the issuing of a 'file copy' command when the user is in control (shown by the prompt of '>>'):-

```
>> COPY
```

DDS then realizes that the arguments are missing and takes control to ask for them,

```
fromfile> FRED
```

and then,

```
tofile> JOHN
```

with the prompt of '>' to show that control is now with DDS.

```
file copied.
>>
```

and control is returned to the user.

7.3 Menu selection: ZOG

Menu systems are straightforward to implement. All that is needed is to list the options available and then to ask the user to choose from among them. For example, several commercially available communication packages for connecting personal computers have a scripting language that can readily be used to generate menu systems without the need for any expertise in programming. It is more interesting, however, to investigate systems that provide a generalized notation for the specification of menu systems.

ZOG (Robertson *et al.*, 1981) is a rapid-response, large-network, menu selection system. Work on ZOG began originally in 1972 and restarted in 1975 at Carnegie-Mellon University, Pittsburgh, USA. The later work was inspired by the interface style used in the PROMIS (Walton *et al.*, 1979) medical information system at the University of Vermont. The name ZOG was chosen as a short arbitrary name and is not an acronym.

In the period 1975–80 ZOG was developed on Digital Equipment Corporation's PDP-10s and Vaxes. Subsequent development moved ZOG to a personal workstation, a PERQ, with high-resolution graphics and a pointing device. In the early 1980s the PERQ version was used as a computer-assisted management system on the aircraft carrier USS *Carl Vinson* (Akscyn and McCracken, 1984). Most recently, development has resulted in a distributed hypermedia system called *knowledge management systems* (*KMS*) (Akscyn *et al.*, 1988b), which is a commercial version of ZOG available on Sun and Apollo workstations.

ZOG has been used as an interface for a command language system, a database retrieval system, a CAI system, a guidance system, an interrogation system and a question-answering system. It is based on a hierarchy of subnets. Each subnet is a tree of frames in the form of a database. The system displays the frames to the user; a self-descriptive version is shown in Figure 7.5.

The frame has a title and a unique number. Then comes some context information, which could, for example, indicate how the user came to

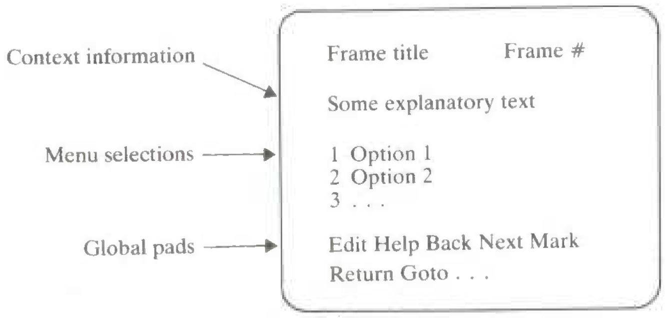


Figure 7.5 An outline of a ZOG frame

this particular frame. Below that can be several lines of descriptive text followed by columns of menu selections, shown as options in Figure 7.5. At the bottom is a line of global pads, or menu choices, that appear on all frames. For example the user could choose *goto* and proceed directly to a known frame.

The user traverses these frames by making selections. Additionally, a selection can evoke an action to accomplish a task. The selection can be made either by using touch-sensitive pads overlaying the selection or by entering a single character from the keyboard. The user therefore navigates through a structured set of subnetworks of interconnected frames, gathering information on the way. As an aid to navigation the user can see a list of frames visited, or a list of frames marked (i.e., anchor points) and can search for a particular frame.

The frames are built using a frame editor, ZED. This can be used at any time by the developer, including when using the system. A user might also use the editor for a limited amount of personalization.

The internal structure of a frame contains not only the visible information as seen by the user but also additional features. For example, there is a maintenance field that contains, among other items, the name of the owner of the frame and access privileges indicating whether it may be modified or viewed by others. The frame builder has the responsibility of providing guidance, on-line documentation and help.

Selections and frames may include action text. This text is sent to the communications multiplexer, which arranges for it to be sent to the correct destination. The frames also have an external format. This supports portability to other ZOG implementations as well as allowing for external maintenance manipulation.

ZOG is really a hierarchical menu system together with a generalized mechanism for the display of information and triggering of actions. As with any menu hierarchy, users can get lost, but are provided with some significant features that aid navigation. The limited display area places a heavy load on short-term memory; even so, evaluation revealed evidence that users failed to read the information in frames. The researchers also investigated the impact of different response times and provide some evidence for it to be less than 0.5 seconds. The later KMS system overcomes many of these problems, for example by always showing two frames so that the user can see the previous frame too.

7.4 Form-filling: COUSIN

COoperative USer INterface was designed by Phil Hayes at Carnegie-Mellon University (Hayes and Szekely, 1982). It is aimed at typical command-level interaction, for example interacting with an operating system, and not the more fine-grained interactions within, say,

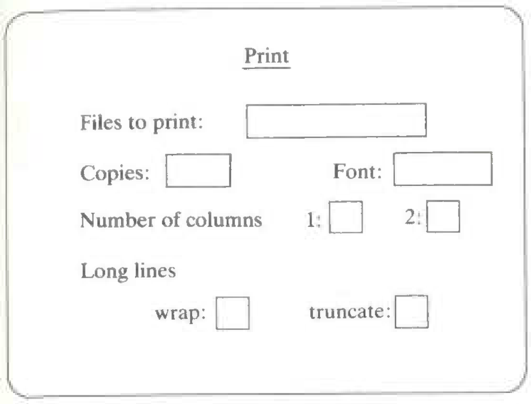


Figure 7.6 A COUSIN form

an editor. COUSIN aimed to present a consistent interface for all applications and to provide the benefit of reduced implementation time.

COUSIN uses a form-filling metaphor. This provides a single interface to several applications. The form-filling has intelligent support, mainly from the type information held in each field.

The (simplified) example in Figure 7.6 shows a command (print) and a set of arguments which must be supplied. Fields for the arguments can have default values that may be overtyped by the user. Each field has a data type and an attribute indicating whether the field is optional.

Three modes of operation are supported. A *non-interactive mode* is used for batch applications: COUSIN ensures correct command arguments are supplied before calling the application. In the *interactive mode* the application is started and can then prompt the user, when needed, for further information. Finally, in the *command loop mode* the user is in control and can issue commands with arguments and observe the feedback from the system.

Several variants of COUSIN were produced, but the main application reported is an interface to the UNIX operating system. In this application, the user typed a command, then COUSIN loaded an appropriate form and assisted the user in filling it in. Once completed, the command and arguments were parsed with COUSIN helping the user to remove errors. The user could save the form for later use, for example, the form could be partly filled in with the user's particular default settings.

The form-filling approach makes good use of bit-map displays, allows arguments to be filled in out of order and permits fields to have defaults. However, applications need modifying before they can be used with COUSIN. Although providing COUSIN as a front-end interface halved the speed of UNIX, this did not seem to be a problem with the experimental users.

7.5 Transition networks

7.5.1 Rapid/USE

Rapid/USE (Wasserman *et al.*, 1986) has been developed at the Medical Information Science Centre at the University of California since 1975. It is based on graphical specification of the required dialogue using families of state-transition nets and subnets and can rapidly produce either a menu or a form-filling dialogue. Diagrams are created using a graphical *Transition Diagram Editor* (TDE) on Sun workstations, or textually by using a special language entered with an ordinary text editor.

The diagram is a representation of a transition network. Each diagram is given a *name*, a *start node* and an *exit node* together with a network of *interior nodes* and *arcs*. A *node definition* is a description of what to do with the screen, for example screen control, display of text and contents of variables. An *arc definition* describes the structure of the diagram. It is labelled with the transition conditions and can have actions, including links to a relational database (TROLL).

Input can be a single key or a fixed-length reply, and can include a default to handle errors. Additionally a time-out can be included. The actions can be routines written in C, FORTRAN or Pascal or can be commands to operate on the relational database. A simple method for calling a single procedure with an integer parameter is also provided. Figure 7.7 shows a typical transition diagram created using the TDE.

A text file such as that shown can then be compiled from the transition diagram, or alternatively the text file can be created directly using a conventional text editor.

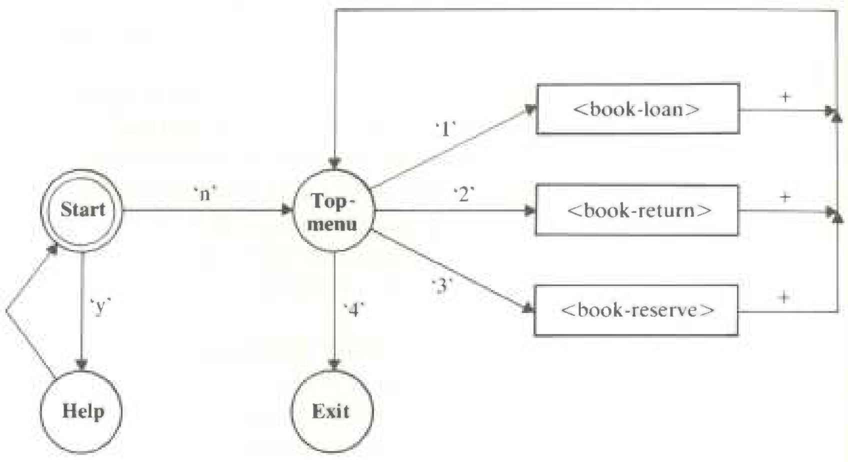


Figure 7.7 A Rapid/USE transition diagram

```
diagram library entry Start exit Exit

node Start
  cs,r1,c_ 'University Library System'
  r+2,c10 'Do you need help (y/n)?'
. . .
arc Start
  on 'n' to Topmenu
  on 'y' to Help
. . .
arc Topmenu
  on '1' to <book-loan>
  on '2' to <book-return>
. . .
arc <book-loan> skip to Topmenu
```

A *transition diagram interpreter* (TDI) can be used immediately on this text file in order to demonstrate and evaluate the menu system. As an aid to evaluation, Rapid/USE can also record two logs of the raw input and the transitions occuring within the system (time, diagram, node, action, input).

7.5.2 The CONNECT system

The CONNECT system (Alty, 1984a) was developed at the University of Strathclyde from 1983 onwards. It is a front-end to the CP/M and MS-DOS operating systems and is based on transition networks together with a production rule system. This latter feature enables the network connectivity to be altered to provide an adaptable interface.

At any time the network is dealing with one node. It then determines which node to move to in response to input from the user. A node can be either a *connector* to communicate with the user, a *task* to invoke a task, a *subnet* for structuring or an *assistor* to help routing as well as playing a role in adaptivity.

Transitions along arcs are determined by parsing the user's input, and can occur on detection of identical text, text somewhere in input, or numerically equal. Actions that are possible when the transition is chosen are *null*, *do a task*, *show screen*, *assign to a global*, *call a subnet*, or *prompt the user*. Display updates work on parts of the screen in the form of non-overlapping panes. Global variables are used to provide communication between the application and the user.

Production rule system

For every network the designer can provide a production rule system as a set of statements of the form:

```
if ... then <action-rules>
```

Each time a node is reached the production rule system is invoked. The *if* parts of the rules access a database and the actions can then modify global variables. Exit arcs from nodes that are labelled with these variables can then have their action modified in the light of the production rules. This provides a level of adaptability, which allows different interfaces to be presented according to the different network paths traversed. The designer or the user (by using function keys) can adapt the dialogue according to user characteristics.

CONNECT comes with a family of tools to help build the system:

- BUILDNET, an interactive network constructor;
- BUILDSCR, to construct screens;
- VIEWNET, to examine nets and screens;
- EXECNET, to execute the net;
- BUILDTSK, to construct tasks;
- TESTSCR, to test a single screen;
- PERFNET, to examine net statistics.

A support system based on Path Algebras (Alty, 1984b) has also been developed. This allows the designer to analyze the net for consistency and to examine the net behaviour. Using this system, it would, for example, be possible to determine what would be on the screen at any particular point in the dialogue.

7.6 Grammar-based systems: SYNICS

Grammar-based notations such as Backus–Naur Form (BNF) are also often referred to as 'production rule' systems in that the application of the rules can be viewed as 'producing sentences (programs) in the language'. For use as a notation for a dialogue system the grammar defines the input language and sometimes the output too (Shneiderman, 1981). The terminals of the grammar correspond to the user's input tokens (mouse clicks, keyboard characters) and the non-terminals are related to the higher-level structure of the dialogue.

The user interface is thus (as far as input is concerned) a parser for this grammar. In order to act on the user input, actions are attached to rules of the grammar. These actions may be calls to procedures within the application or prompts to the user. This is the approach taken in the design of the *yacc* compiler–compiler available under UNIX. The main advantage of using context-free grammars is that a wealth of related work exists from their use in programming languages. However, they have proved awkward to use, particularly in controlling when actions are performed. Additionally, they are only well suited to parsing linear textual input and not to more graphically based interactions.

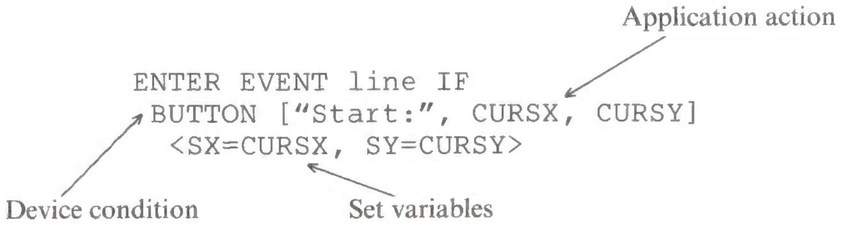
An example of the form of the notation is shown below in a command from a screen-based editor. The designer defines non-terminal symbols (shown with < and >) as a series of rules defined (shown as ::=) in terms of a series of terminal symbols (shown as UPPER-CASE or as single-quoted characters) and other non-terminal symbols. Choices are separated with a vertical bar and repetition indicated with an asterisk. Terminal symbols must be matched with user input, while non-terminal symbols are for structuring purposes.

```
<locate> ::= <get line> <move to line>
<get line> ::= <find command> | <scroll>*
<move to line> ::= <cursor movement>*
<find command> ::= '/' <search string> CR
<scroll> ::= '^D' | '^U'
<cursor movement> ::= '⇐' | '⇑' | '⇒' | '⇓'
```

SYNICS (Edmonds, 1981) was developed in 1978 at Leicester Polytechnic and also described by Guest (1982). The name SYNICS is taken from SYN tax and semantICS. The initial idea was to use BNF to specify the dialogue. The system produced a table-driven, top-down recognizer that accepts strings of text from the user.

Subsequently this was augmented with a transition network front end (dialogue definition language, DDL). In adding the transition net front-end the BNF is now associated with arc conditions and therefore combines the expressive power of the string matching of BNF with the more attractive features of transition networks.

Later work has produced SYNICS2 (Edmonds and Guest, 1984). This uses *dialogue events*, which contain GKS output and control, then take input from the application or user and are followed by any required application action and setting of variables. For example:


```
ENTER EVENT line IF
 BUTTON ["Start:", CURSX, CURSY]
  <SX=CURSX, SY=CURSY>
```

Thus the device condition is tested for whether the button is pressed, and, if so, the variables are set to the (x, y) coordinates of the pointer locations and the event is sent to the application, indicating that this is the start of a line with parameters set from the variables containing the (x, y) coordinates.

7.7 Natural language

Natural-language communication has long been viewed as a Holy Grail in human interaction with computers (Rich, 1984). Among other objectives, this is one of the aims within the man–machine communications component of the Japanese Fifth Generation Project (Simons, 1983). However, the use of natural language has its dangers as was shown by early work on natural language communication at MIT (Weizenbaum, 1966). Weizenbaum developed a pseudonatural language program named ELIZA (after Eliza Doolittle in Shaw's play *Pygmalion*), which used keyword and template matching and would then trigger off output that was some simple transformation of the input. A variety of other stratagems such as standard stock phrases were also used, and in addition ELIZA could refer back to earlier inputs when there were lulls in the user response. That it was so successful at creating the illusion of being able to carry on a conversation, and yet had virtually no understanding, is a warning that the use of natural language may cause users to imbue the interface or application with more intelligence than is the case.

In subsequent research into natural language dialogues it has generally been found that the more successful applications have either limited its use to output only, or have operated within a narrow knowledge domain.

At the UK National Physical Laboratory (NPL), Dr Chris Evans developed a medical interviewing system (Bevan and Pobgee, 1981) that showed a successful use of natural language for textual output. The output of Evans's system consisted of questions for the patient to answer, and also reassuring phrases. The user input was confined to selecting one of three possible answers using buttons labelled 'yes', 'no' and 'don't know'. The system also split the logic of controlling the dialogue from the interview text itself, thus modularizing the system so that non-programmers could prepare the interview text. The NPL work always made it clear that the computer was simply repeating text and collecting simple replies.

Work on expert systems has shown that the NPL approach can be successfully extended to provide full language output with a terse input of a restricted form of natural language. The work on Mycin (Shortliffe, 1976), for example, developed a medical diagnosis expert system for infections. An abbreviated example from Mycin is shown below:

```
[5] What is the infection?
** PRIMARY-BATEREMIA
[6] Please give the date and time where...
** MAY 5, 1975
...
```

Eventually the Mycin system offers a diagnosis and a suggested therapy.

With less constrained natural language input, a more restricted domain is generally required. Work at MIT by Dr Terry Winograd on his SHRDLU (Winograd, 1972) project demonstrated quite impressive understanding in the constrained world of model blocks. However, it is not clear how this could be extended to larger domains.

7.7.1 UNIX Consultant (UC)

UNIX Consultant (Wilensky *et al.*, 1984) was developed at the University of California, Berkeley in 1984 and acts as an advisor to a user of the UNIX operating system. It is a natural language interface that comprises an input analyzer, an output generator, a context and memory model, a common-sense planner, extensible knowledge bases for UNIX and English and a goal analysis system for acquiring new knowledge. Users can query UC on actions to be performed, ask about command names and formats, receive on-line definitions and get debugging help. If this fails to help, then the user is at least still able to proceed, as UC sits to one side of the user interacting with UNIX.

An example interaction (taken from the Wilensky *et al.*, 1984 paper) is as follows:

```
User: How do I delete a file?
UC:   Typing 'rm filename' will remove the
file with filename from your current
directory.
User: A directory?
UC:   Typing 'rm -r directoryname' will
recursively delete the directoryname and all
the files and directories contained in the
directory.
User: How do I print a file on the
lineprinter?
UC:   To print a file on the lineprinter type
'lpr filename'
...
  some intervening interactions
...
User: Has the file foo been printed yet?
UC:   The file foo is in the lineprinter
queue.
User: How can I cancel it?
UC:   To remove files from the line printer
queue, type 'lprm username'
```

Some problems can arise if the form of help, while being correct, results in the user making an error. For example, the following, imaginary, conversation would result in the user removing all his or her files!

```
User:  How do I get more disk space?
Naive: Use the command 'rm *' to get more
disk space.
```

It is reported that in experimental use the response times were very slow—up to 15 seconds on a VAX 11/780.

In summary, as indicated above, there are problems with users forming incorrect conclusions from the use of natural language interfaces. Natural language is also ambiguous (Hill, 1983) and requires large programs even for narrow domains. Typing complete sentences as natural language input is far from an ideal interaction style, and even speech recognition, where natural language input would seem most appropriate, may not be the answer given the current state of the art.

7.8 Adaptive dialogues

We understand human behaviour much less than we understand computers. With the interface as a separate module the design can be modified in the light of user experience. It must therefore be easy to change. The aim is to minimize the cognitive load and to compensate for the weaknesses of the human. Hence the first need is to identify the weaknesses and strengths of particular users.

7.8.1 Agents for adaptation

Adaptive interfaces are concerned with ways in which interfaces can be built so that *while in use* they can be adapted. Edmonds (1981) suggested a variety of possible adaptation mechanisms: adaptation by the system automatically; adaptation by the system in cooperation with the user; adaptation by exploiting an expert intermediary; and adaptation by the end-user himself or herself.

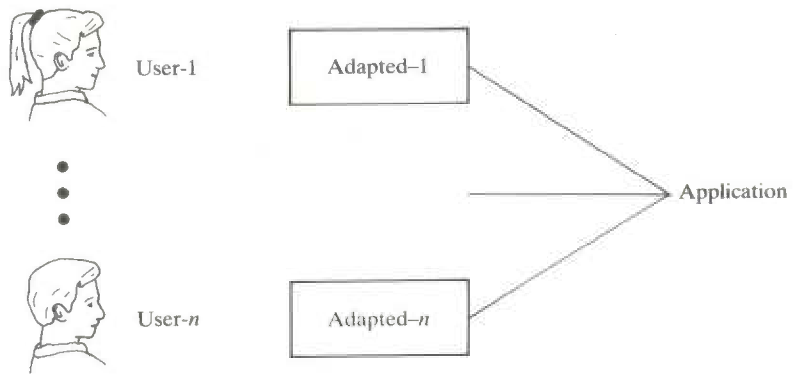


Figure 7.8 One application with several users' adapted interfaces

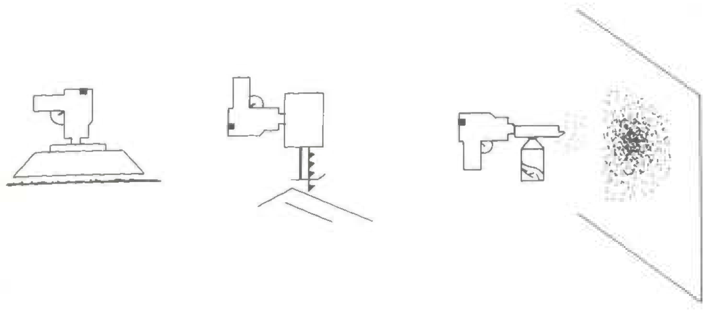


Figure 7.9 A common tool adapted to different tasks

In traditional linear system development (see Chapter 1), the system as a whole often fails because system interface needs cannot be anticipated and met. Of course, users are themselves adaptable, but in many situations they may not want to adapt to an inflexible system. In any case, users' needs are diverse and constantly changing. As shown in Figure 7.8, the application need have only one user interface if that could be adapted to the needs of different users and can have differently adapted parts of the system for an individual user over time.

7.8.2 Forms of adaptation

Some possible forms of adaptation are:

- dialogue style,
  e.g., yes/no, menus, forms, command language, natural language;
- current task context,
  e.g., contents of a directory or mail box,
  macros for frequent commands,
  context-sensitive help and error measages;
- the task,
  e.g., setting an appropriate environment for this task.

Figure 7.9 illustrates the concept of the tool being adapted to different tasks yet presenting the same interface to the user. If this approach is compared with the alternative of having a separate sanding machine, jigsaw and paint-spray, the interfaces may well be quite different. In human–computer interaction, the two alternatives are readily illustrated by comparing the Macintosh 'universal' interface style with the wide range of interfaces presented by different application packages for the IBM PC. A further consideration is how well the two approaches allow the user to complete a task, for example in using the saw to create a hole in wood or metal.

7.8.3 Timing of adaptation

Choices must also be made of *when* to adapt. After each interaction? After a context change? After task completion? Between sessions? Whatever time quanta are chosen, the system will need feedback, i.e., *an evaluation of the user's performance with the current adaptation.* This is problematical, since both the user and the system are adapting simultaneously.

7.8.4 Adaptive intelligent dialogues (AID)

The phase 1 exemplar of the AID project was an adaptive interface to the electronic mail system Telecom Gold. Its principal adaptation dimensions were the level of guidance, the use of context, recognizing commands from analogous systems, allowing user tailoring, and informing the user about additional unused functionality. Under guidance from the user model, the system could vary the level of prompting, the amount of feedback and the help level.

Though superficially appealing, the design of a complete adaptive dialogue system proved to be problematical, and the two later phases of the AID project therefore attempted to break down the issues involved in adaptation into more tractable components (see Chapter 13).

The most important conclusion from the project was that adaptation is not a single idea but a family of techniques that can be applied from the earliest stages in design through to run-time. Although the AID project did not achieve any specific major breakthrough which delivered outstanding user benefits, it contributed enormously to the understanding of the mechanisms of adaptation, both within humans and machines. In particular, it established constraints which had not previously been realized on the circumstances in which adaptation could be used effectively in interface design.

7.9 Summary

This chapter has discussed a variety of dialogue delivery systems. In general, a modular system structure is adopted, with the objective of making the range of dialogues and applications supported as wide as possible. Each system provides some form of *specification language* for the user input, but most are weak on describing the output side of the interactions.

Different systems provide different levels of support for *dialogue design and development*. Some systems are simply *toolkits* for building interfaces of a particular type; others also provide support for subsequent *simulation* and *evaluation*. Where post-design support is provided, it may be either independent of, or integrated with, the application software. However, there are as yet no aids to check whether the designer is adhering to HCI guidelines. Similar approaches are taken in the design of direct manipulation dialogue systems: in view of their importance and widespread exploitation, these are discussed under the general heading of *windowing systems* as Chapter 10 of this book.

References

Akscyn, R. M. and D. L. McCracken (1984) 'ZOG and the USS Carl-Vinson: lessons in systems development', *Interact '84, Proceedings of 1st IFIP Conference on Human Computer Interaction*, **1**, 303–308.

Akscyn, R. M., E. Yoder and D. McCracken (1988a) 'The data model is the heart of interface design', *CHI '88, Conference Proceedings on Human Factors in Computing Systems (Washington, May 15–19)*, Soloway, Frye and Sheppard (eds), Addison-Wesley, Wokingham, 115–120.

Akscyn, R. M., D. I. McCracken and E. A. Yoder (1988b) 'KMS: A distributed hypermedia system for managing knowledge in organizations', *Communications of the ACM*, **31** (7), 820–835.

Alty, J. L. (1984a) *The Conversational Node Executor: CONNECT*, University of Strathclyde MMI Group, Glasgow.

Alty, J. L. (1984b) 'The application of path algebras to interactive dialogue design', *Behaviour and Information Technology*, **3** (2), 119–132.

Beech, D. (ed.) (1986) *Concepts in User Interfaces: A Reference Model for Command and Response Languages*, Springer, Berlin.

Bevan, N. and P. Pobgee (1981) 'MICKIE—a microcomputer for medical interviewing', *International Journal of Man–Machine Studies*, **14**, 39–47.

Edmonds, E. A. (1981) 'Adaptive man–computer interfaces', in *Computing Skills and the User Interface*, M. J. Coombs and J. L. Alty (eds), Academic Press, Orlando, Fla., 389–426.

Edmonds, E. A. and S. P. Guest (1984) 'The SYNICS2 user interface manager', *Interact '84, Proceedings, 1st IFIP Conference on Human Computer Interaction*, vol. 1, 53–56.

Guest, S. P. (1982) 'The use of software tools for dialogue design', *International Journal of Man–Machine Studies*, **16**, 263–285.

Hayes, P. J. and P. A. Szekely (1982) 'Graceful interaction through the COUSIN command interface', *International Journal of Man–Machine Studies*, **19** (3), 285–305.

Hill, I. D. (1983) 'Natural language versus computer language', in *Designing for Human–Computer Communication*, M. S. Sime and M. J. Coombs (eds), Academic Press, Orlando, Fla., 55–72.

Rich, E. (1984) 'Natural language interfaces', *IEEE Computer*, **17** (9), 39–47.

Robertson, G., D. McCracken and A. Newell (1981) The ZOG approach to man–machine communication, *International Journal of Man–Machine Studies*, **14**, 461–488.

Robinson, J. and A. Burns (1985) 'A dialogue development system for the design and implementation of user interfaces in Ada', *Computer Journal*, **28** (1), 22–28.

Shneiderman, B. (1981) 'Multi-party grammars and related features for defining interactive systems', *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-12** (2), 148–154.

Shortliffe, E. H. (1976) *Computer-based Medical Consultations Mycin* (North-Holland), Elsevier, New York.

Simons, G. L. (1983) *Towards Fifth-generation Computers*, NCC Publications, Manchester.

Walton, P. L., R. R. Holland and L. I. Wolf (1979) Medical guidance and PROMIS, *IEEE Computer*, **12** (11), 19–27.

Wasserman, A. I., P. A. Pircher, D. T. Shewmake and M. L. Kersten (1986) 'Developing interactive information systems with the user software engineering methodology', *IEEE Transactions on Software Engineering*, **SE-12** (2), 326–345. Also in Baecker and Buxton (see under 'General').

Weizenbaum, J. (1966) 'Eliza—A computer program for the study of natural language communications between man and machine', *Communications of the ACM*, **9** (1), 36–45. Reprinted in *Communications of the ACM 25th Anniversary Issue, Jan. 1985*, **26** (1), 23–27.

Wilensky, R., Y. Arens and D. Chin (1984) 'Talking to Unix in English: an overview of UC', *Communications of the ACM*, **27** (6), 574–593.

Winograd, T. (1972) *Understanding Natural Language*, Academic Press, Orlando, Fla.

Further reading

General

Baecker, R. M. and W. A. Buxton (eds) (1987) *Readings in Human–Computer Interaction*, Morgan Kaufmann, (Afterhurst), Hove, E. Sussex. (The editors have selected about 60 papers from major researchers in the HCI field.)

Pfaff, G. E. (ed.) (1985) *User Interface Management Systems*, Springer, Berlin. (Reports on the 1983 Seeheim workshop.)

Introduction

Alexander, H. (1987) *Formally Based Tools and Techniques for Human–Computer Dialogues* Ellis Horwood, Chichester.

Anderson, R. H. and J. J. Gillogly (1976) *Rand Intelligent Terminal Agent (RITA): Design Philosophy*, Rand Corporation, Santa Monica, Calif.

Backus, J. W. (ed.) (1960) 'Report on the algorithmic language Algol60', *Communications of the ACM*, **3**, 229–314.

Chomsky, N. (1957) *Syntactic Structures*, Mouton, The Hague.

Cockton, G. (1986) 'Where do we draw the line?—derivation and evaluation of user interface software separation rules', in *People and Computers, Proceedings of the Second BCS Conference on HCI*, M. J. Harrison and A. F. Monk (eds), Cambridge University Press, Cambridge, 417–431.

Dance, J. R., T. E. Granor, R. D. Hill, S. E. Hudson, J. Meads, B. A. Myers and A. Schulert (1987) 'Report on the run-time structure of UIMS-supported applications', *Computer Graphics*, **21** (2), 97–101.

Green, M. (1986) 'A survey of three dialogue models', *ACM Transactions on Graphics*, **5** (3), 244–275. Also available in *User Interface Management Systems*, G. Pfaff (ed.), Springer, Berlin, 1985.

Guest, S. P. (1982) 'The use of software tools for dialogue design', *International Journal of Man–Machine Studies*, **16**, 263–285.

Newman, W. M. (1968) *A System for Interactive Graphical Programming*, *AFIPS SJCC*, **32**, 47–54.

Rosson, M. B., M. Susanne and W. A. Kellog (1988) 'The designer as user: building requirements for design tools from design practice', *Communications of the ACM*, **31** (11), 1288–1298.

Totterdell, P. A. and P. Cooper (1986) 'Design and evaluation of the AID adaptive front end to Telecom Gold', in *Proceedings of the BCS Conference on HCI*, M. J. Harrison and A. F. Monk (eds), Cambridge University Press, Cambridge, 281–295.

Wasserman, A. I. (1985) 'Extending state transition diagrams for the specification of human–computer interaction', *IEEE Transactions on Software Engineering*, **11** (8), 699–713. Also in Baecker and Buxton (see under 'General').

COUSIN

Hayes, P. J., P. A. Szekely and R. A. Lerner (1985) 'Design alternatives for user interface management systems based on experience with COUSIN', *CHI '85, Conference Proceedings on Human Factors in Computing Systems*, Addison-Wesley, Wokingham, 169–175.

Transition networks

Cockton, G. (1985) 'Three transition network dialogue management systems', *People and Computers, Proceedings of the First BCS Conference on HCI*, Johnson and Cook (eds), Cambridge University Press, Cambridge, 138–147.

Denert, E. (1977) 'Specification and design of dialogue systems with state diagrams', *International Computing Symposium*, North-Holland, Amsterdam, 417–424.

Parnas, D. L. (1969) 'On the use of transition diagrams in the design of a user interface for an interactive computer system', *Proceedings of the 24th National ACM Conference*, 379–385.

Wasserman, A. I. (1985) 'Extending state transition diagrams for the specification of human–computer interaction', *IEEE Transactions on Software Engineering*, **11** (8), 699–713. Also in Baecker and Buxton (see under 'General').

Woods, W. A. (1970) 'Transition network grammars for natural language analysis'; *Communications of the ACM*, **13** (10), 591–606.

CONNECT

Alty, J. L. and A. Brooks (1985) 'Microtechnology and user friendly systems, the CONNECT dialogue executor', *Journal of Microcomputer Applications*, **8**, 333–346.

Brooks, A. and C. Thorburn (1988) 'User driven adaptive behaviour, a comparative evaluation and an inductive analysis', *People and Computers IV, Proceedings of the Fourth BCS Conference on HCI*, D. M. Jones and R. Winder (eds), Cambridge University Press, Cambridge, 237–255.

Grammar-based systems

Fountain, A. J. and M. A. Norman (1985) 'Modelling user behaviour with formal grammar', *People and Computers, Proceedings of the First BCS Conference on HCI*, Johnson and Cook (eds), Cambridge University Press, Cambridge, 3–12.

Hopcroft, J. E. and J. D. Ullman (1960) *Formal Languages and Their Relationship to Automata*, Addison-Wesley, Reading, Mass.

Reisner, P. (1981) 'Formal grammar and human factors design of an interactive graphics system', *IEEE Transactions on Software Engineering*, **SE-7** (2), 229–240.

Scott, M. L. and S.-K. Yap (1988) 'A grammar-based approach to the automatic generation of user-interface dialogues', *CHI '88, Conference Proceedings on Human Factors in Computing Systems (Washington, May 15–19)*, Soloway, Frye and Sheppard (eds), Addison-Wesley, Wokingham, 73–78.

Natural language

Weizenbaum, J. (1976) *Computer Power and Human Reason: from Judgment to Calculation*, W. H. Freeman, New York.

Adaptive dialogues

Browne, D. P., R. Trevellyan, P. Totterdell and M. Norman (1987) 'Metrics for the building, evaluation and comprehension of self-regulating adaptive systems', *Interact 87, Proceedings of 2nd Conference on HCI*, H. J. Bullinger and B. Shackel (eds), North-Holland, Amsterdam, 1081–1087.

Cockton, G. (1987) 'Some critical remarks on abstractions for adaptable dialogue managers', *People and Computers, Proceedings of the Third BCS Conference on HCI*, D. Diaper and R. Winder (eds), Cambridge University Press, Cambridge, 325–343.

Fowler, C. J. H., L. A. Macaulay and S. Siripoksup (1987) 'An evaluation of the effectiveness of the adaptive interface module (AIM) in matching dialogues to users', *People and Computers, Proceedings of the Third BCS Conference on HCI*, D. Diaper and R. Winder (eds), Cambridge University Press, Cambridge, 345–359.

Gargan, R. A., J. W. Sullivan and S. W. Tyler (1988) 'Multimodal response planning: an adaptive rule based approach', *CHI '88, Conference Proceedings on Human Factors in Computing Systems (Washington, May 15–19)*, Soloway, Frye and Sheppard (eds), Addison-Wesley, Wokingham, 229–234.

Hoppe, H. U. (1988) 'Task-oriented parsing—a diagnostic method to be used by adaptive systems', *CHI '88, Conference Proceedings on Human Factors in Computing Systems (Washington, May 15–19)*, Soloway, Frye and Sheppard, Addison-Wesley, Wokingham, 241–247.

Totterdell, P. A. and P. Cooper (1986) 'Design and evaluation of the AID adaptive front end to Telecom Gold', *Proceedings of the BCS HCI Conference on HCI*, M. J. Harrison and J. F. Monk (eds), Cambridge University Press, Cambridge, 281–295.

Totterdell, P. A., M. A. Norman and D. P. Browne (1987) 'Levels of adaptivity in interface design', *Interact '87, Proceedings of 2nd Conference on HCI*, H. J. Bullinger and B. Shackel (eds), North-Holland, Amsterdam, 715–722.

Tyler, S. W. (1988) 'SAUCI: a knowledge-based interface architecture', *CHI '88, Conference Proceedings on Human Factors in Computing Systems (Washington, May 15–19)*, Soloway, Frye and Sheppard (eds), Addison–Wesley, Wokingham, 235–240.

Direct manipulation

Bewley, W. L., T. L. Roberts, D. Schwit and W. L. Verplank (1983) 'Human factors testing in the design of Xerox's 8010 "Star" office workstation', *Proceedings of CHI '83, Conference on Human Factors in Computing Systems*, ACM, 72–77. Also in Baecker and Buxton (see under 'General').

Carroll, J. M. and S. A. Mazur (1986) 'Lisa learning', *IEEE Computer*, **19** (10), 35–49.

Jacob, R. J. K. (1985) 'A state transition diagram language for visual programming', *IEEE Computer*, **18** (7), 51–59.

Jacob, R. J. K. (1986) 'A specification language for direct-manipulation user interfaces', *ACM Transactions on Office Information Systems*, **5** (4), 283–317.

Shneiderman, B. (1983) 'Direct manipulation: a step beyond programming languages', *IEEE Computer*, **16** (8), 57–69. Also excerpted in Baecker and Buxton (see under 'General').

Smith, D. C., C. Irby, R. Kimball, B. Verplank and E. Harslem (1982) 'Design of the Star user interface', *Byte*, **7** (4), 242–282. Also in Baecker and Buxton (see under 'General').