Int. J. Man–Machine Studies (1986) **25**, 439–451

Equal opportunity interactive systems

COLIN RUNCIMAN AND HAROLD THIMBLEBY
Department of Computer Science, University of York, York, YO1 5DD, U.K.

(*Received 10 February 1986 and in revised form 24 July 1986*)

One view of interactive computer systems is that the user, having problems to solve, supplies the "givens" of these problems to the machine, which in response supplies as output the "unknowns". Reassigning or discarding these labels "givens" and "unknown" is a time-honoured heuristic for problem-solving. Also, people seem to prefer interpretations without such labels for fast interactive systems, and mere speed in systems that do embody fixed distinctions between input and output often contributes little towards ease of use—it may only serve to emphasize a frustrating mechanical dumbness. We therefore apply the same heuristic to the design of interactive computer systems, noting that a number of existing successful interactive system styles can be viewed as the outcome of this approach.

1. Introduction

When several real-world events appear to be instantaneous, an observer may not be able (or may not wish) to distinguish between causes and effects. For example, naïve students of computer science often fail to appreciate the subtleties of character echoing when characters appear to be echoed on a VDU screen *at the same time* as they are typed. A perceptible delay in echoing increases the acceptability of a cause-and-effect explanation. Similarly, users often fail to distinguish between moving a cursor (on a screen) and moving the mouse (which is the input which the program interprets to make the display outputs to move the cursor icon). Users frequently talk in terms of moving the *cursor*, rather than the mouse (or, indeed, their hand). What is strictly *feedback* is viewed as the user's direct *input*.

Also, human problem-solving is rarely confined to a linear chain of thought from given to unknown. The vast majority of mental "links" are undirected—they do not serve as one-way streets of cognition; rather they *relate* information in a general manner. If a person can translate (say) French to English, it is reasonable to assume he can also translate English to French and, indeed, elaborate on the relation between French and English in a general way. In whatever way the mechanism of human translation really works, it does not seem to involve fixed assumptions about what is given and what is unknown. This kind of flexibility is clearly assumed by educationalists who, to distinguish rote learning from deeper understanding, routinely use exercises varying the kind of information given: if you *understand* addition you can complete "6+□=3", if you *understand* quadratic equations, you can not only find roots of given equations, but can also construct equations from roots, check roots, and so on.

So, if a computer system embodies a fixed classification of pieces of information that may be communicated during interactive use—causes and effects, givens and unknowns, inputs and outputs—that system will very likely present an unnatural appearance of frustrating dumbness to its users. Conversely, if a system has no *ab inito*

0020-7373/86/100439+13$03.00/0 © 1986 Academic Press Inc. (London) Limited

predisposition to a particular assignment of roles to pieces of information it will appear to its user as flexible and cooperative.

The starting point for this paper is the observation that it is both practical and useful to build interactive systems which do not, presumptively, impose a fixed mode of cognition, cause and effect, upon their users. We hope to show that the problem-solving heuristic of reassigning the labels "given" and "unknown" (explained both in general and by example in Polya's (1945) classic book "How to Solve It") can be fruitfully applied to the design of interactive systems. We call it the "equal opportunity heuristic". To constrain the scope of this paper somewhat we will make the simplifying assumption that the user is intended to be in full control of a particular dialogue with the computer. In general this need not be the case and, indeed, equal opportunity would suggest equality in interuption and initiative (and other mechanisms of taking control) in the interaction. However we will not explore "equal initiative" (mixed initiative) interactive systems further, nor generalizations to various modes of interaction such as speech.

2. The problem of premature design commitment

Programs are written to support user interfaces. Purely technical programming decisions—for instance that a computation should be organized as a particular fixed sequence of steps—often intrude onto the user interface in the form of limitations that are inexplicable without a thorough understanding of the program implementation. Many such programming decisions are actually quite arbitrary: but the intellectual loads they place on *both* programmer *and* user are significant.

This problem is an instance of a more general one: premature design commitments must be avoided.

So at the outset of design it is more important to specify, for example, what it means to sort data correctly than to choose between, say, insertion or quicksort algorithms. There have been various methods proposed to delay commitments to a particular algorithm as long as possible: use of a purely functional programming system (Turner, 1982) avoids the need to specify sequence in many places where it is mandatory in a conventional imperative system; less radically, Bornat (pers. comm.) uses an idealized programming notation (e.g. guarded commands, concurrent assignment), and provides mechanical rules (involving fixed but arbitrary choices) for translation into a more strictly sequential language, such as Pascal.

Sheil (1983) has made cogent arguments for delaying commitments as expressed in strong typing in "structured" programming languages (e.g. Ada). He argues that the *implementation* problems of the 1970s, which were pretty well solved by structured programming and strong static typing—making programs as committed as possible—are going to be replaced by *design* problems, caused by the very investment in the redundancy which provides security and ensures consistency.

All this is relevant to the concerns of this paper because many design methods (conventional systems analysis in particular) fall into premature commitments by *beginning* with the separate enumeration of inputs and outputs. The designers may be unaware both of this commitment itself and of its consequences in limiting the opportunities for different styles of use. For batch-processing applications such a design strategy may be entirely appropriate, but for interactive systems it is a disaster. Similarly, defining an interactive system *component* as a (mathematical) function must entail the

distinction between the function argument and the function result and (depending on the role of the component) this may also involve distinctions between inputs from a user and outputs from the computer system. Just as it is possible and frequently more productive to design programs without premature commitments to computational order or type specifications, it is desirable to design interactive systems without premature commitments to the rôles of input and output.

3. Equal opportunity defined

Conventionally, programs do distinguish strongly between inputs and outputs: programs are expected to take certain input data as "given"; corresponding output data are "unknowns", to be obtained. For example, given the coefficients of a quadratic equation, a program might obtain (using the standard formula perhaps) the roots of that equation. But even though one can obtain coefficients of a quadratic with given roots, our hypothetical root-finding program is unable to do so. It obtains roots from coefficients and not the other way around.

The exchange of input and output roles is only one of many possibilities. In what we shall call an *equal opportunity* system, all that can be supplied or demanded by the machine can also be supplied or demanded by the user; equivalently, each item of information passed across the man-machine interface can pass in either direction. In this way, there is a form of parity between the participants of interactive computation. This is the ideal; in practice *some* restrictions will be an inevitable consequence of overall design aims—a matter to be discussed shortly.

So the user of an equal-opportunity quadratic equation program could supply both coefficients and roots and have their correspondence checked. Or he might give some coefficients and some roots, in which case the reponse of the machine will be possible values for the others. Even giving neither coefficients nor roots is a possibility.

4. Equal opportunity illustrated

We now want to show briefly that several well-established styles of user interface design that the reader may have regarded as quite separate have, to varying degrees, involved some equalization of opportunity in the sense just described. Indeed, we should like to go further and suggest that this is of the essence in each case, although it takes different forms in the different kinds of system. We will consider spreadsheets; programming-by-example; expert systems; query-by-example; Prolog; and, as an example typical of many interactive systems, the programming environment *COPE*. Our list is by no means exhaustive.

The interface presented by a *spreadsheet* program (Kay, 1984) is a rectangular grid of cells. Associated with each cell there is a rule that governs what value is displayed there. A rule may be very simple (e.g. "is zero") but will often involve values displayed in other cells (e.g. "is the maximum value of other cells in this column"). There is no predetermined concept of "input cells" or "output cells"; the user sets rules in whatever cells he pleases and the system adjusts values displayed in whatever other cells are necessary so that all display rules are obeyed. At one moment the user may inspect a cell to view an "unknown", at the next he may redefine the display rule for this same

cell, thereby introducing a new "given" with potential "unknown" effects on other cells. Even within a single interaction, the boundary between "givens" and "unknowns" may be hard to discern: for example, is a recurrence rule between a group of cells an accelerated means of expressing "givens" or the computation of "unknowns"?

In *Programming-by-Example* (Halbert, 1984; Summers, 1977) the user provides what would normally be considered the output of a program in order to define the program itself. In other words, the user provides the system with an example of what he would like it to do; the system records the user's actions and can perform them again. The system organizes the program so the user can later generalize and parametrize what he did.

In *expert systems* (Michie, 1979) equal opportunity interaction can be found in a variety of forms. To begin with, the user may supply such information as he has available so that what is "given" (as input) in one consultation with an expert system may be "unknown" in another and vice versa. In the context of such information an expert system may yield conclusions ("unknowns" to user) and, on request, justify these conclusions by a chain of reasoning; but the user also has the opportunity to offer conclusions (now "givens") and have them similarly justified or refuted by a chain of reasoning. More profoundly, an expert system may treat inference rules (or whatever other representation of knowledge it uses) primarily as input when acquiring new knowledge from a human expert but as output when acting as an intelligent tutor. In general consultation with a knowledgeable user, rules may be used in both of these rôles.

Query-by-Example (Zloof, 1977) is a language and associated interactive system for manipulating the information in a relational database. A user can *request* a tuple-template with appropriate element headings for any relation in the data-base. He has the opportunity under each heading to supply an example value, or to supply a fixed value or to leave the space blank. The system can respond with all actual instances from the database that match the templates: where a template value was only an example the actual value could be anything of the same type (where two example values are the same in templates the corresponding actual values must be the same too); fixed template values must match exactly; blanks are ignored. However, the user can also *supply* templates for new relations; these new relations are maintained dynamically in the correct (template determined) correspondence with the originals. In this way there is considerable adaptability in the role of templates; they may specify sectional views of data for output, but also the construction or input of new relations.

In *Prolog*, (Clockin & Mellish, 1984; Hogger, 1984) a *logic programming* language, it is possible to define procedures for which the input/output roles of parameters are not fixed. Such procedures form an obvious basis for the construction of equal opportunity systems. A standard introductory example is "concat", defined as follows:

```
concat ([], X, X).
concat ([H|T], X, [H|TX]): -concat (T, X, TX).
```

The first clause states that the concatenation of the empty list with any list X is just X. The second clause states that the concatenation of a list with first item H and remaining items T with any list X is H followed by the concatenation of T and X. An intuitive meaning for a call concat (A, B, C) is "if possible make substitutions so that

A and B concatenated together are the same as C". Here are just a few possibilities:

CALL		EFFECT	
concat ([1, 2], [3, 4], Q)		Q = [1, 2, 3, 4]	
concat ([1, 2], Q, [1, 2, 3, 4])		Q = [3, 4]	
concat (Q, [3, 4], [1, 2, 3, 4])		Q = [1, 2]	
concat (P, Q, [A, B])		P = [],	Q = [A, B]
	or	P = [A],	Q = [B]
	or	P = [A, B],	Q = [].

In the dialect micro-Prolog (Clark & McCabe, 1984) this freedom of argument roles extends to arithmetic primitives. An expression such as "$F = 32 + C \times 9/5$" may be used to check that F and C have the desired relation (for denoting temperature in Fahrenheit and Celsius units) when both C and F are given. If, however, only C is given micro-Prolog will determine F as output. In fact many programming languages can perform these two operations (though sometimes requiring a different symbol for "=" in the two cases). But micro-Prolog is unusual in that if F alone is known *precisely the same expression* will determine C. Finally, when both F and C are unknown, micro-Prolog generally requires additional information such as "F between (0 500)", from which it can effectively construct an F−C conversion table over the given range.

COPE, (Archer & Conway, 1981) an interactive programming environment, is "... cooperative both in the sense of being flexible and tolerant with respect to the form of user entries, and in being *willing to perform chores that the user is generally asked to do for himself*" (our emphasis added). Very similar statements have been made about many interactive systems. A common aim is often to help remove the burden of tedious input from the user; this may be done by arranging certain system output to be treated as input under the user's control (cf. 6.2.1).

5. The necessity of some inequality—and its dangers

If a computer system *could* itself move a mouse it would in principle have a choice of valid responses when the user moved the mouse. In particular, rather than tracking the mouse with the cursor, the system could respond simply by moving the mouse back again, leaving the cursor where it was. Under this arrangement both mouse and cursor would be immovable! Hence the convention that the mouse is always an input device and the cursor an output.

Or again, the user of a constraint-based system (Borning, 1981) might supply an equation to convert between Celsius and Fahrenheit. As the user sets and resets the Celsius value the system responds, always displaying the corresponding Fahrenheit temperature. Similarly, if the user changes the Fahrenheit temperature the system alters the Celsius. But it would be unfortunate if equal opportunity extended to all values in the system: why then should not the system instead alter the equation relating Fahrenheit and Celsius so that it becomes true? The convention is, of course, that equations are never considered valid means of system output.

In practice it is unusual for the user of an application written in Prolog to be able to benefit directly from equal-opportunity procedures in the underlying implementation. Mellish (1981) reports that variation of argument roles is typically slight in large Prolog programs. It seems that most Prolog programmers (having been first accustomed to

programming in, say, Lisp or Pascal) are unaccustomed to this kind of flexibility and fail to exploit its potential. There are exceptions, however, as Warren (1980) highlights in his discussion of the development of a compiler in Prolog. Notice that the "concat" example of section 4 is more usually called "append"; the use of the term "append" rather suggests the *particular* case of taking "givens" (A, B in the example above) one of which it appends to the other to obtain the "unknown" (C).

As these examples illustrate, some inequality of opportunity is invariably (and quite properly) introduced at some point in a system design. The danger is that it can be introduced by hidden and unquestioned assumptions rather than as the result of an explicitly reasoned design step. Of particular concern is that once such assumptions have been incorporated into a design (especially when fixed in its implementation) it can be extremely difficult to reorganize the system to permit greater variation of input/output roles or greater equality of opportunity in interaction. Fixing input/output roles opens the way for the designer to select a particular computational model: any design work within that computational framework may rely too heavily on the assumptions by which it was selected to generalize to other computational models.

Because of this, we believe that many of the standard human factors "success stories" of iterative design (e.g. rationalizing screen layout after user participation) are *entirely* fortuitous. It is easy to imagine a program design based upon a computational model for which it is impossible to redesign a screen layout without redesigning the entire implementation. For example, few pre-VDU hardcopy-based applications could have been easily modified to present their dialogue in a different order. It is an important area of research to find computational models and corresponding user interface styles that are malleable. Otherwise, when our understanding of human factors matures (to the point where we can address deeper cognitive issues), iterative design will become obsolete because of the high cost of software iterations.

Users are likely to underestimate the implementation costs of modifications involving variation of input/output roles. For example, if a system validates data provided by the user (e.g. that Tuesday October 29 1985 is a valid date) it will be curious to users that the relevant information which *necessarily* exists in the computer cannot be used in other ways (e.g. to find the day of the week on December 25 1985). The problem is that the knowledge is "procedurally embedded" and is not accessible except by executing a fixed procedure in a fixed way. Incidentally, people are not immune to a similar problem; the more skilled their performance becomes often the harder it is to conceptualize it: human memory is a classic example. However, we retain a remarkable flexibility even for the least conscious actions.

The more a programmer uses procedural embedding of knowledge the more he will rely on abstractions to manage the complexity of the embedded knowledge. To the user of such a system the abstractions (e.g. the commands) available to him will appear limited and contrived. If a programming paradigm is found which permits the programmer to model artefacts available to the user, the user interface will be cleaner. Ideally, it should exhibit the particular traits or style of the underlying model sufficiently well for the model to be of constructive use (when suitably expressed) for the user. Despite the implicit approval of Prolog earlier in this paper, one disadvantage of it is that its implementation world of lists and clauses is usually completely concealed at the user interface level. In contrast, in object-oriented systems it is usual for objects to be both visible and manipulable by the user. The implementation of these objects encapsulates

information about their visual form, and so the temptation for the programmer to embed knowledge in a fixed procedural form is minimized. (A spreadsheet can be implemented very easily on an object-oriented machine. The cells with which the user interacts map onto objects in the implementation). The so-called, and widely advocated, "direct manipulation" (Shneiderman, 1982) style of user interface can then be readily and *consistently* implemented. Kay (1982) gives further examples.

6. Equal opportunity as a heuristic: design principles

Having discussed the general principle of equal opportunity we now use it as a guide to derive more specific design laws such as the ever-popular What You See Is What You Get (WYSIWYG) (Thimbleby, 1983).

The use of such "generative principles", (Thimbley, 1983; Gaines, 1984; Thimbleby, 1984) is an appealing approach to the design of effective user interfaces. Use of a generative principle can bring a higher-order consistency to the design process, and when suitably expressed a generative principle can be used by both designer and user to advantage. But where do designers get new principles from? In what sense can designers claim that the principles they adopt are valid? How do designers ensure they adopt the principles with sufficient precision? We will attempt to show that equal opportunity interaction is a promising heuristic for developing principles, and for knowing precisely what they mean in the given context. By developing the principles for themselves for their specific applications context, designers can be more certain than usual that the principles are applied appropriately.

There are also important psychological issues. How should the principles be rephased (e.g. in task-oriented terms) for users? How will such principles (e.g. expressed as "golden rules") interact with the user's world knowledge?

6.1. DERIVING "WYSIWYG"

Suppose a word processor provides an editor (E) which operates on text (T). The word processor also provides a formatter (F) which formats the current text and causes it to be printed. Initially we suppose the word processor is not of the WYSIWYG genre, so that the text probably contains all sorts of incantations to a text formatter (meaning "centre this text", "start a paragraph", "underline" and so on). More formally we can view E as a collection of functions and F as a single function.

E_i: $T \rightarrow T$

F: $T \rightarrow P$

T for texts

P for printed forms

The directions of the arrows in Fig. 1 make plain the assumptions about input/output roles for data involved in the editing and formatting operations.

Now let us apply the ideas of equal opportunity to F by trying to discard the fixed assumptions about input and output roles of T and P. That is, we want to replace the two arrows labelled F by undirected links. The main obstacle in the way of this plan is that, for all systems of formatting directives known to us, F is severely many-1. So

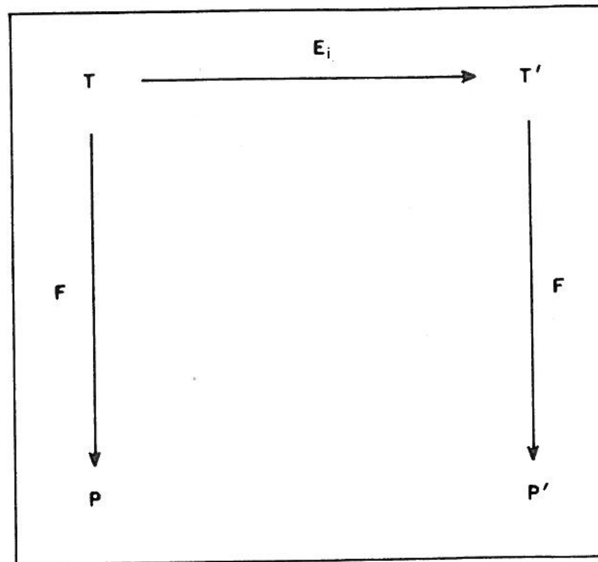

FIG. 1.

it seems we must force either non-determinism, or arbitrariness or irritating choices on the user.

But this problem has a standard solution: we conceal unwanted distinctions between equivalent members of T by making T an internal representation and only retaining quotients of it (e.g. P) in the external specification. The concealment is indicated by shading in Fig. 2.

The result is one interpretation of WYSIWYG.

Unfortunately this reasoning is over-simplified. For example, the user editing a large document may have a view of it that is limited to a single A3-size screen; this is much smaller than the size of the final printed document, which will fill many pages. Such

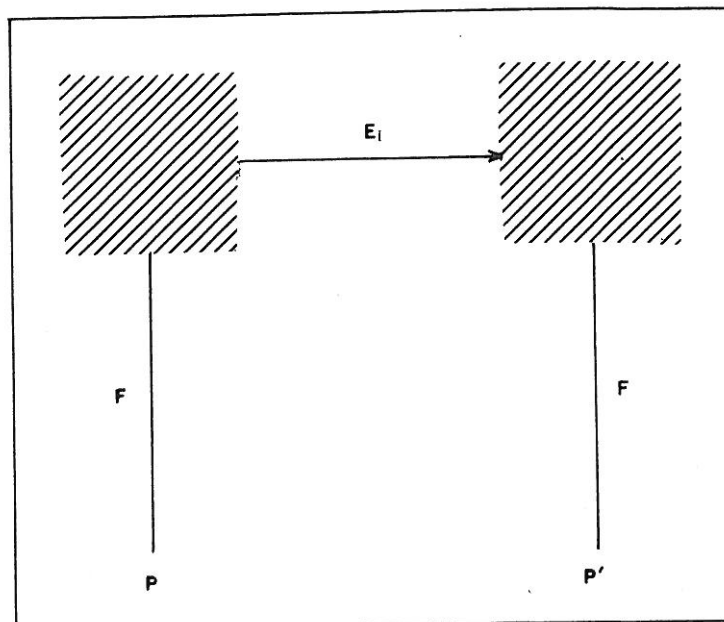

FIG. 2.

issues, which are of little concern here have been more carefully worked out elsewhere (Dix & Runciman, 1985). More general issues of formalising interactive system properties are discussed in Harrison & Thimbleby (1985).

6.2. OTHER PRINCIPLES

In less detail now we briefly mention some more design principles that may be viewed as products of the equal opportunity heuristic. The actual derivations of the corresponding formal principles and choice of application domains is left to the reader.

6.2.1. If you can see it you can use it

Any output (displayed on a terminal screen) should be able to be reused and provided as input for other functions. Teitelman (1979) argues that this gives the user "pronoun reference": the user can say to the system *this* expression or *that* value. This greatly increases the system's effective input bandwith available to the user. Some systems permit textual output to be selected, edited and then resubmitted as new input to the system; this operation is easily represented on a bitmapped display together with pointing operations. "Block-mode" VDU terminals readily support equal opportunity in this sense, and as local VDU editing commands are generally fixed in firmware, the user has the additional assurance that the commands for selecting, editing and reusing text are identical for all applications.

6.2.2. Commensurate effort

The relative computational costs of tasks carried out by the system should be reflected in the relative amount of work the user must do to invoke the necessary computations—a principle applied by Jensen & Wirth (1974) in the design of Pascal. For example, if creating X requires more effort than creating Y then so also should the effort needed to destroy X be greater than that needed to destroy Y. Further, a user must not be able to destroy data significantly faster than he can create it: this may seem restrictive, but it may also be seen to encourage the provision of accelerated ways of creating information in the first place!

6.2.3. Non-pre-emption (Swinehart, 1974)

This could be seen as an extreme corollary of "commensurate effort". A computer sytem should not demand input from the user, refusing to continue until the user supplies that input. A pre-emptive system corners the user, forcing him to supply input before proceeding. This can be very disruptive. An advantage of multi-window multi-process systems is that the user is generally able to provide input to whichever processes he chooses, thereby sidestepping the problem of unequal opportunity subsytems. Pre-emption then becomes an issue when interacting with the window manager itself because it is not possible to sidestep *it*. Pre-emption may sometimes be a necessary protection mechanism—an "inevitable inequality".

6.2.4. Self demonstrability

The provision of help and guidance is another aspect of equal opportunity: the system output is intentionally selected as descriptive of plausible or potential input. It is less demanding of the computer system if it can describe how it may be used by way of a (say) script-following demonstration (Thimbleby, 1984).

7. Equal opportunity as a heuristic: case studies

We now present three illustrative case studies of equal opportunity employed as a heuristic in the design process. A more substantial example has been published elsewhere, (Thimbleby, 1986*a*).

7.1. MECHANICAL PROGRAM PROVING

We will first illustrate the value of the equal opportunity heuristic by tracing the efforts of computer scientists in the early 1970s to solve the probelm of proving programs correct.

Initially, programs were assumed to be the "givens" of this problem and their proofs were regarded as the "unknowns". In other words, the goal was to construct a system which read a program text and would print its proof. The plausibility of this view was enhanced by the anticipated potential of mechanical proof procedures. However, a prerequisite for proof is the formulation of an exact specification, and in nearly every program proof the most important parts are the invariant assertions without which the rest of the proof cannot proceed:

program → specification → invariant assertions → proof

However, determining suitable specifications and invariants for programs turned out to be a very hard problem. Eventually, it was found productive to take the formal specification as "given" from which the "unknowns" program and proof were to be derived:

specification → invariant assertions → program *and* proof

7.2. INTERACTIVE LITERATE PROGRAMMING: A BRIEF INTERFACE DESIGN CASE STUDY

Cweb is a batch-style system which enables documentation and program code to be interleaved. Interleaved documentation/code is fed through cweb which can then produce either compilable code or high-quality typeset code with cross-referenced documentation. The notation for interleaving is baroque.

The paper on cweb (Thimbleby, 1986*b*) suggests an interactive variation which removes the burden of the user knowing this notation. For example, it was suggested that graphical devices (such as boxes) can take the place of explicit bracket symbols. But so far as input/output roles are concerned, this "new" user interface is just like the original batch system: the user merely interacts through a sanitized (graphical) notation.

If the starting point of design is the ideal of equal opportunity interaction, rather than a superficial avoidance of poor notation, then a more interesting interface merges. For example, the user could edit code or documentation at will (both were outputs of the original cweb) and the system—now much more sophisticated—would ensure that the two representations remain projections of a common data object. As in our derivation of WYSIWYG, the user need now never see this internal data.

7.3. THE UNIX®† COMMAND LANGUAGE

What UNIX command should you use to find out who owns a file? A textbook answer (Bourne, 1982) is "ls −l *file*", meaning list in full the details of the named file. But if

† UNIX is a trademark of AT&T.

you *only* want to know the owner, this does far too much, contrary to the small single-purpose software tools philosophy of UNIX (Kernighan & Pike, 1984). More seriously, from the user interface perspective, it is not obvious how to discover such a command; a keyword search of the UNIX manual would suggest "chown" (which changes ownership of a file). The difficulty is that UNIX commands have fixed assumptions about inputs and outputs which restrict their use. In the case of "chown", no visible output is given at all!

In considering a variation of "chown" in which some attempt has been made to provide equal opportunity interaction, let us first change its name to "ownsfile". The style of application here is very close to that of a Prolog procedure call.

USER	SYSTEM
ownsfile *user file*	yes/no
ownsfile — *file*	owner of *file*
ownsfile *user* —	names of files owned by *user*
ownsfile — —	list of files and their owners

According to this interpretation "ownsfile" does not change owners or files but simply supplies information—it has no side-effects.

To specify intended change in a manner consistent with the style of the UNIX command language one might provide a "−f" (for force) option so that, for example, the user could enter

ownsfile −f *user file*

to set file ownership. There are other possibilities.

The advantage of this approach is that each command would localize access to *all* information pertinent to a particular issue, such as file ownership as in this case. See also Fraser (1980) who discusses the potential of editing the output of commands. For example, a generalized editor may permit the user to edit file properties (by effectively editing the output of some file enquiry) in order for the user to change certain properties. A debugger is a core-image editor; an electronic mail system is a mail editor and so on. The issues of equal opportunity arise because once the "unknowns" (outputs) of these commands (file lister, debugger, mail system *etc.*) are edited the user will then wish to assert them as new "givens". A high degree of interface uniformity can be attained by making as many as possible state-changes achievable by the same editing mechanisms: the "if you can see it you can use [edit] it" maxim again (cf. section 6.2.1).

8. Summary and Conclusion

Removing assumptions about what is given and what is unknown first appeared as a heuristic for problem-solving. It is used in education to distinguish understanding from rote learning. In some sense, demonstrating ability to accomodate such variation demonstrates intelligence or at least a certain advantageous and cooperative flexibility in approach.

For interactive computer systems, we have adopted the "equal opportunity" slogan to advocate a corresponding removal of assumptions about which side of the interface must supply a particular piece of information.

Delaying design commitments to the point where they can be properly analysed and justified is essential. At that point, commitment trade-offs should be made explicit and carefully reasoned through. Starting with the idealized assumption of equal-opportunity interaction is one way to delay commitments in the design of interactive systems. A large number of successful interactive system styles incorporate this idea to varying degrees. It appears that placing equal opportunity interaction as an ideal can usefully guide user-interface design, and even when innovative styles do not emerge we would contend that useful insights will still be obtained by the designer who adopts this problem-solving heuristic.

Whether users have the capacity or inclination to exploit equal opportunity effectively remains to be explored in complex cases. It will certainly depend on the purpose of the user's particular interactive session. The user's performance in a "creative" session may be degraded by the explosion of choices open to him, whereas a "problem-solving" session may be facilitated by the greater adaption by the system to the user's framework. (Of course, the user's perception of his performance and his preference for equal opportunity will be determined by many factors other than his actual performance). The cost of implementing full equal opportunity (indeed, even when it is possible) may be prohibitive; consistent restrictions of the ideal—which the user readily comprehends—may be difficult to determine.

References

ARCHER, J. JR & CONWAY, R. (1981). COPE: a cooperative programming environment, TR. 81-459, Department of Computer Science, Cornell University.

BORNING, A. (1982). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems* **3**, 353-387.

BOURNE, S. R. (1982). *The UNIX System.* Addison-Wesley.

CLARK, K. L. & MCCABE, F. G. (1984). *Micro-PROLOG: Programming in Logic.* London: Prentice-Hall.

CLOCKSIN, W. F. & MELLISH, C. S. (1984). *Programming in Prolog* (2nd edn).Berlin: Springer-Verlag.

DIX, A. & RUNCIMAN, C. (1985). Abstract models of interactive systems. In JOHNSON, P. & COOK, S. Eds, *Proceedings People and Computers: Designing the User Interface*, pp. 13-22.

FRASER, C. W. A generalized text editor. *Communications of the ACM*, **23**, 154-158.

GAINES, B. R. (1984). Dialog shell design. *Interact'84, First IFIP Conference on Human Computer Interaction*, London, pp. 344-349.

HALBERT, D. C. (1984). Programming by example. OSD-T8402, XEROX Palo Alto Research Centers.

HARRISON, M. D. & THIMBLEBY, H. W. (1985). Formalising guidelines for the design of interactive systems. In JOHNSON, P. & COOK, S. Eds, *Proceedings People and Computers: Designing the User Interface*, pp. 161-171.

HOGGER, C. J. (1984). *Introduction to Logic Programming* (*APIC Studies in Data Processing, No.* 21). London: Academic Press.

JENSEN, K. & WIRTH, (1974). *Pascal User Manual and Report.* Berlin: Springer-Verlag.

KAY, A. (1982). New Directions for Novice Programming in the 1980s. In *Infotech State of the Art Report*, pp. 210-247 Pergamon.

KAY, A. (1984). Computer software. *Scientific American* **251**, 41-47.

KERNIGHAN, B. W. & PIKE, R. (1984). *The UNIX Programming Environment.* Prentice-Hall.
MELLISH, C. S. (1981). The Automatic generation of mode declarations for prolog programs. DAI Research Paper No. 163, Department of Artificial Intelligence, University of Edinburgh.
MICHIE, D. Ed. (1979). *Expert Systems in the Microelectronic Age.* Edinburgh University Press.
POLYA, G. (1945). *How to Solve It.* New Jersey: Princeton University Press.
SHEIL, B. (1983). Environments for exploratory programming. *Datamation,* **29**, 131-144.
SHNEIDERMAN, B. (1982). The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology,* **1**, 237-256.
SUMMERS, P. D. (1977). A methodology for LISP program construction from examples. *Journal of the ACM,* **24**, 161-175.
SWINEHART, D. C. (1974). Copilot: a multiple process approach to interactive programming systems. Stanford University AI Memo 230.
TEITELMAN, W. (1979). A display oriented programmer's assistant. *International Journal of Man-Machine Studies* **11**, 157-187.
THIMBLEBY, H. W. (1983). 'What you see is what you have got'—a user engineering principle for manipulative display? In *ACM Proceedings Software Ergonomie,* pp. 70-84, Nuremberg.
THIMBLEBY, H. W. (1984). Generative user-engineering principles for user interface design. In *Interact*'84, *First IFIP Conference on Human Computer Interaction,* pp. 62-107. London.
THIMBLEBY, H. W. (1986*a*). The design of two innovative user interfaces. In HARRISON, M. & MONK, A. Eds, *Procceedings People and Computers: Designing for Usability,* pp. 336-351.
THIMBLEBY, H. W. (1986*b*). Experiences of 'Literate Porgramming' using CWEB (a variant of Knuth's WEB). *Computer Journal,* **29**, 201-211.
TURNER, D. A. (1982). Recursion equations as a programming language. HENDERSON, P. & TURNER, D. A. Eds, *Functional Programming and its Applications.* pp. 1-28. Cambridge: Cambridge University Press.
WARREN, D. H. D. (1980). Logic programming and compiler writing. *Software—Practice and Experience,* **10**, 97-126.
ZLOOF, M. M. (1977). Query-by-example: a database language. *IBM Systems Journal,* **16**, 324-343.