User interface design and formal methods

HAROLD THIMBLEBY
University of York

Programming is moving away from the 1960's image of being an 'art' to having a mathematical basis: no longer, we hope, the unreliable bug-ridden programs of the past! Simultaneously, there is greater awareness of the need to write usable programs: real programming is not just a matter of obtaining correct programs, but of obtaining *easy to use* programs. Typically, the user of a computer system has been faced with inaccurate documentation of a difficult to use and bug-ridden computer system. What can be done?

A group of psychologists and computer scientists are collaborating at the University of York in research to combine good empirical user interface design practice with formal methods. One purpose of this article is to introduce the reader to some of this research. Another is to show how formal methods naturally encourage designers to consider issues carefully, and how such careful consideration leads to interesting and practical results.

We start by formalising user-engineering principles, [5,8]. Once formalised, the principles may be fed directly into the software engineering design process. The final system will be consistent with the initial user-engineering basis and this will not only obtain the consistency naturally expected from formal methods, but the consistency should be visible to the user. The user-engineering principles chosen as the basis may also be re-expressed in suitable terms for the user (*eg*, for the user manuals and training material). Thus, our method should result in more coherent and more easily explained systems.

Consistency versus curiosity

Perhaps the foremost aim of formal methods is to replace *ad hoc* decisions in programming (many of which end up as bugs anyway) by a structure that can be carefully and reliably reasoned about. A good way to impose structure is to devise requirements that the program is to satisfy. To take full advantage of the power of mathematical reasoning, the requirements must be expressed formally. For our purposes, we are therefore concerned with mathematical formulations of user-engineering principles.

Many user-engineering principles cannot be expressed in an abstract enough way to have an impact on the formal approach. For instance, a user-engineering requirement that 'an important error message should be displayed in flashing red' is about as exciting as saying '30 + 1 should be larger than 30'. For each such situation there has to be a separate rule, and (as we can plainly see from the size of existing design rule-books) the number of useful guidelines becomes phenomenal. Instead, more economical use can be made of theorems like $n + 1 > n$, which is true for all numbers, not just 30. An analogous, formally interesting, user-engineering principle is 'minimise the number of "modes" presented to the user'. Users can easily forget which mode they are in; the more modes there are the less predictable the computer system *seems* to become. A mode-ridden system puts a burden on the user to remember which mode the system is in; if the user is distracted (by a 'phone call, say) or has a break, then it may be difficult to resume work using the system if it is not obvious what mode it is still in. It turns out that modelessness is easy to express formally.

All good user interfaces should permit the user to recover from mistakes. Some enlightened systems provide an 'undo' command. If the user makes a mistake (which they recognise as such!) they can use the command to 'undo' the mistake. Again, it is easy to express this requirement formally.

Now a surprise: modelessness and undo are incompatible (for non-trivial systems). There are two conclusions: either we have taken a too-restrictive view of modes and modelessness above, or (I think more likely) all systems purporting to have no modes and an undo command must be difficult to use. In this case the difficulties arise in precisely the context that is supposed to help the user recover from errors. If interactive systems are designed informally, the designer will be unaware of the need for the resolution of such problems. Resulting systems will inevitably contain inconsistencies (*ie*, bugs or plain lies in the documentation). I have only shown a trivial example, which will serve to introduce a more user-centred discussion about modes below. Further formal results are described in [3]. How we handle these interaction issues for software engineering purposes will be explained briefly later.

A user-centred view of modes

One purpose of studying modeless designs is to explore ways in which 'mode errors' may be avoided. Concern for the user, beyond the initial idea that modes generally are a bad thing, was only implicit in

the preceding analysis. Users do not make mode errors unless they have intentions that are not met by the user interface. If the user has no particular expectations then any mode-dependent result can hardly be classified as a 'mode error'. A user-centred definition of mode error is required. We may define that a mode error only occurs when the user obtains an unintended effect, but which could have had the intended effect had the computer been in a different mode.

There are various ways in which it is possible to design user interfaces to reduce the probability of such mode errors. We may change the design: *a*) the user interface language may be made more powerful (so that fewer commands need multiple interpretations); *b*) the range of functions provided by the system might be reduced. Conversely, we may try to change the user's expectations: *c*) the system might try to influence the user's expectations interactively while it is being used; *d*) the user's training may attempt to ensure the user's expectations are consistent with the system (an approach obviously facilitated by a formally-guided design).

We have performed experiments to test our idea *c*), as this approach seems to interfere least with other factors in the design process (in general, we may be unable to dictate the tasks or command language to the customer). The experiment may be described very briefly as follows. A group of people used a system that could make different types of key click, as keys are typed, depending on what mode the computer was in. This was intended to help guide the user's expectations, and so reduce mode related errors. We found a significant improvement [6].

Programmable user models

Rather than taking a psychologically-based user-engineering principle, such as 'reduce modes', as a candidate for formalisation it is possible to take a much more abstract view. If we have a user interface design under consideration, we might ask, 'Are the decisions demanded by the user interface computable?' In other words, could a (suitably programmed) robot use the proposed system? This is a reasonable question to ask for, surely, if the user interface requires non-computable responses it must be exceedingly hard for a person to use. With more psychological bias we would be inclined to ask whether the user's responses are not only computable but are not too hard *for the user* to compute.

We have noticed that interactive systems design effort is usually concentrated almost entirely on the *computer program*. But, of course, the user and computer may be viewed as a pair of communicating processes and the user interface as also being implemented by a *user program*. Users obviously have to 'program' themselves in order to use an interactive system. What can we discover by making the user's programming explicit? We have tried to imagine what a user's programming language would look like, and how explicit programming with it would influence decisions the designer has to make [4]. Would designers make better systems if, in addition to writing computer software, they had also to write user software and demonstrate it could (in some precise sense) operate their design? If a designer had to program a 'programmable user', perhaps as part of the proposed system's acceptance tests, then this would help redress the unequal balance of attention normally loaded for the computer. It could make designers consider the user and the user's needs more carefully.

The user's program and the computer's program are duals of each other. They both implement the user interface dialogue. If the two programs are *strict* duals of each other, then it should be possible to derive one from the other. Obtaining a user program from a computer program (we already know exactly what computer programs are, so this is an easy place to start) is a matter of inverting the program. Do we still need the user program? The answer is 'yes'. Since program inverses are rarely unique, inversion introduces non-determinism (*ie*, uncertainty). The non-determinism represents the program's (*ie*, the designer's) unstated assumptions about the user's tasks and goals.

Design innovation

A computer system is used for problem solving. For instance, a user may have in mind some task (such as printing the week's accounts) and the problem is to express this task in the way required by the user interface. Or we might consider the creative use of computers as problem solving: I certainly had a few problems writing this article, which I solved interactively! Now, people have been studying human problem solving over the years, two notable names being Descartes and Polya. Various heuristics have been proposed to help people solve problems. If these heuristics help people solve 'real life' problems, why not design interactive computer systems to encourage their successful application?

One widely advocated problem solving heuristic is to see what happens if you treat the problem as solved (*ie*, try working backwards). In a conventional user interface, pieces of information flow in both directions between the user and the computer. However, the user is never able to use the working backwards heuristic to help solve the problem in hand. It would mean saying, as it were, to the computer, 'What should I have said to make you say such-and-such'. In other words, we might suppose an improved interface could provide the user with an opportunity to provide information normally provided by the computer – with the intention that the computer 'works backwards' to supply the missing information.

Prolog programmers will be familiar with a similar idea: the parameters of Prolog predicates have no *a priori* disposition to either direction of information flow. Although the property is generally termed 'polymodality' in programming languages, we prefer the term 'equal opportunity' since 'mode' is already overworked in HCI and our phraseology naturally lends

itself to describing 'opportunities' and 'inequalities' in a productive fashion. Using equal opportunity we have not only been able to devise innovative user interfaces to such stable designs as four-function calculators [10], but we have been able to derive (more truthfully, rationalise) certain user-engineering principles [7]. This, in turn, helps us to sharpen our ideas about formalising user-engineering principles.

Software engineering

We have started to build a novel interactive system, based loosely on Knuth's literate programming ideas [9]. The idea is to produce a program support environment which permits users to specify, program, and document a system concurrently. We elected for a novel user interface to help ensure that if we obtained a 'good' usable system, we had indeed obtained our results by careful formal design – if we had chosen to redesign a conventional application (*eg*, for word processing) our prior knowledge of successful word-processing systems might have influenced what is supposed to be formal design. When our prototype literate programming system is working we will be able to do experiments to test the validity of our approach and how it impacts users. It is fairly certain that we will obtain an effective system, for *even* if the user interface rules we chose as a basis were quite arbitrary the final result would still be consistent and easier to document clearly. It should therefore be easier to learn and to use. What we do not know yet is whether users can make very effective use of precise knowledge about interactive systems. It is certainly the case that the arbitrariness in many conventionally designed systems will have innoculated most users against believing anything that purports to be a precise 'rule' of interaction!

To implement a complex system such as this, and to retain the full advantages of formalised user-engineering principles, we have devised a three-level method. The first level is abstract, and is oriented towards formal proofs, such as are needed to ensure the various user interface requirements are met. Reasoning about the design at this level may proceed without worrying about implementation details. This model is then transformed into a 'conventional' formal specification (we happen to use an equational method). The point is that the transformation preserves the correctness of the earlier proofs. Normally, it would be very difficult to prove such requirements were satisfied by a concrete formal specification. At the third level, the concrete specification is transformed into an implementation.

As usual, there is a danger that such an abstract approach to software design compromises efficiency. We have developed a technique called 'interface drift' which allows us to optimise performance (specifically, to optimise module interfaces) without compromising correctness. The technique of interface drift is analogous to first implementing a program simply (and correctly) and then introducing buffers and caches to obtain improved performance. We have also introduced other methods to facilitate the formal specification of issues arising in window managers. A more complete discussion may be found in [1,2].

Conclusions

Formal methods in user interface design, contrary to fears of 'formalising away the human element', in fact make designers pay more careful attention to the needs of the users than they might otherwise. The attention to detail, and the consistency obtained through formal methods, can also help the user form a reliable understanding of interactive systems.

On the other hand, there is a danger that the precision of formal methods may lead the designer to be even more proud of the design than usual. It may be more tempting to blame design failures on the user, since (provably!) the design is correct and blameless! But in our experience, formal methods bring to light many otherwise hidden and unstated assumptions. Realising their significance can only instill a sense caution into the designer: user interfaces are remarkably complex things. We have found the formal approach to user interface design, contrary to the dry school-book view of mathematics, an exciting adventure. It has generated, and it still promises to generate, innovative and better user interface ideas.

References

1 A. J. Dix & M. D. Harrison (in press), *Principles and Interaction Models for Window Managers*, in *People and Computers: Designing for Usability*, A. Monk & M. D. Harrison eds., Cambridge University Press

2 A. J. Dix, M. D. Harrison & E. E. Miranda (in press), *Using Principles to Design Features of a Small Programming Environment*, in *Proceedings Software Engineering Environments*, I. Sommerville ed., Peter Peregrinus (pub)

3 A. J. Dix & C. Runciman (1985), *Abstract Models of Interactive Systems*, in *People and Computers: Designing the Interface*, P. Johnson & S. Cook eds., Cambridge University Press, pp. 13–22

4 N. V. Hammond & C. Runciman (in press), *User Programs: A Way to Match Computer Systems and Human Cognition*, in *People and Computers: Designing for Usability*, A. Monk & M. D. Harrison eds., Cambridge University Press

5 M. D. Harrison & H. W. Thimbleby (1985), *Formalising Guidelines for the Design of Interactive Systems*, in *People and Computers: Designing the Interface*, P. Johnson & S. Cook eds., Cambridge University Press, pp. 161–171

6 A. Monk (in press), *Mode Errors: A User-Centred Analysis and Some Preventitive Measures Using Keying-contingent Sound*, International Journal of Man-Machine Studies

Continued on page 18

will be willing to contribute to the discussion. It might also be argued that such meetings would take ages to come to any firm decisions about anything. This reservation is not borne out in practice and there are methods available of curtailing discussion or asking for a vote at some juncture (although better facilities for formalising and structuring meetings are certainly essential). It is not the case that electronic meetings must be seen as a substitute for face-to-face meetings, there are some matters that, because of their nature, or the speed with which a decision is required, can only be resolved by people sitting round a table. However there are many more mundane meetings which could be adequately converted to an electronic equivalent without any loss of quality or timeliness to the decisions reached. CBMS do have certain advantages for group working, as Kerr and Hiltz [5] observe, the medium

> a) *Increases (virtually to infinity) the size of the common 'information space' that can be shared by communicants (and provides a wider range of strategies for communicants to interrupt and augment each other's contributions).*
> b) *Raises the probability of discovering and developing latent consensus.*

It has been found that groups communicating in this way tend to exhibit a high degree of personal interaction, involvement and group cohesiveness; something which my own experiences certainly bear out.

With the next generation of communications software will come a widening of the varieties of groups that will choose to communicate *via* this medium. The December 1985 edition of *Practical Computing*, for example, printed a list of over 20 *new* bulletin boards run mostly by enthusiasts. The next generation of users will not be interested in building their own modems or getting more out of their micro's comms package. They will be interested in communicating with people for reasons totally unconnected with the medium itself. The first such groups are likely to be communities of professionals whose communications costs are met by their companies. Specialists in all sorts of fields from medicine to archaeology to philosophy could group together and exchange views much more easily and more readily than at annual conferences and other formal occasions. The Usenet community in the USA and UK is already a vast network where people subscribe and contribute to many different newsgroups. This is currently a specialised community made up of a high proportion of academics, it has still to reach the man in the street – but it will.

Human factors has an extremely important role to play in all this. The provision of facilities embedded in communications software that will allow the allocation of roles, the use of tools and structuring of the individual's communications environment are all areas that need a well thought out, user centred approach. Just how to get the most from groups communicating *via* a CBMS will require considerable knowledge about how humans communicate in general, how groups are formed and led, and how such attributes as trust, honesty, frankness and equality can best be engendered between people that meet *via* their computers. The human factors expert ought to be the person that systems designers turn to when decisions about such matters need to be made. I believe that in many ways human factors is the axis upon which the next generation of communications that will see Man into the 21st century hinges. In order really to take advantage of the synthesis of communications and computing technologies we need to make sure that representatives of these technologies, at all levels, are aware of the role that human factors can and must play in the successful design of such systems.

References

1 Wilbur, S., Rubin, T. and Lee, S. 1986. A Study of Group Interaction over a Computer-Based Message System. In press. Proceedings of HCI '86, University of York, Sept. 22–26, 1986.
2 Carey, J. 1980. Paralanguage in Computer Mediated Communication. Proceedings of the Association of Computational Linguistics.
3 Keisler, S. 1986. The hidden messages in computer networks, The Harvard Business Review, January–February 1986.
4 Maude, T. I., Heaton, N. O., Gilbert, G. N., Wilson, P. A. and Marshall, C. J. 1984. An Experiment in Group Working on Mailbox Systems. Interact '84 – First IFIP Conference on 'Human-Computer Interaction', London.
5 Kerr, E. B. and Hiltz, S. R. 1982. Computer-Mediated Communication Systems, Academic Press.

User interface *Continued from page 15*

7 C. Runciman & H. W. Thimbleby (1986), *Equal Opportunity Interactive Systems*, YCS 80, Department of Computer Science, University of York
8 H. W. Thimbleby (1984), *Generative User-Engineering Principles*, in Proceedings INTERACT '84, B. Shackel ed., North-Holland, pp. 661–666
9 H. W. Thimbleby (1986), *Experiences of 'Literate Programming' Using Cweb (A Variant of Knuth's WEB)*, Computer Journal, **29**(3), pp. 201–211

Acknowledgements

The work reported here was done by Alan Dix, Nick Hammond, Michael Harrison, Eliot Miranda, Andrew Monk, Colin Runciman, Harold Thimbleby, and Paul Walsh under a SERC/ALVEY grant. The original idea for 'programmable users' comes from as-yet unpublished work by Thomas Green and Richard Young. Additional details may be obtained from Dr M. D. Harrison, Department of Computer Science, University of York.