Manuals as Structured Programs

Mark Addison & Harold Thimbleby

Department of Psychology, University of Stirling, FK9 4LA, Scotland.

Tel: *+44 (0)786 467640*
Fax: *+44 (0)786 467641*
EMail: *maa1@stirling.ac.uk, hwt@compsci.stirling.ac.uk*

A user manual may provide instructions that, if the user follows them, achieve any of certain objectives as determined by the manual designers. A manual may therefore be viewed rather like a computer program, as pre-planned instructions. Accordingly, software engineering and its methods may be applied *mutatis mutandis* to the manual and its design process.

We consider structured programming methods, and show that some difficulties with user interfaces may be attributed to manuals being 'unstructured.' Since there are many programming metrics, and very many styles of manuals for user interfaces, this paper is concerned with justifying the approach and showing how insightful it is.

Keywords: manuals, hypertext, multimedia, human-computer interaction, finite state machines, flowgraphs

1. Introduction

There is much evidence that improved manuals improve user acceptance (Carroll, 1990). There is also the argument that improving manuals by changing the system documented by them leads to improved systems (Thimbleby, 1990). Thus manuals are an essential part of the system life cycle: from requirements and design, through usability, to acceptance.

The importance of manuals certainly extends beyond their use in training and reference. In some sense (whether this is explicit or implicit) a user must 'know' what they are doing to use a system, and the manual is a representation of what they could know. Whether a user could in practice verbalise their knowledge as a system manual is unlikely — it may not even be necessary to be able to do so if the system feedback is sufficient (cf. Payne, 1991); however it is certain that, for many users, the manual is the prime input to their initial system knowledge. Other work has discussed what `program' the user may be executing (e.g., Runciman & Hammond, 1986); here we are concerned with a conventional concrete representation of the program as a manual, rather than a psychological or abstract expression of the user's mental model.

Despite the importance of manuals, however they are generally fitted into the design cycle after implementation has been completed, very much as an output of a finished software engineering process. There might seem to be little choice in this; indeed, it has been argued (e.g., Jackson, 1983) that the manual *should* be written at this stage, since at any earlier stage the system to be documented is not finalised.

This paper argues that manuals can be viewed more productively than the conventional reactive software engineering view suggests, and we give some indications of how one might proceed. We would emphasise, however, that this paper is very much a re-orientation, rather than a final solution. Like writing computer programs, writing manuals is part art and part science, and one bright idea is not going to change everything, nor even be a valid solution for all occasions. Moreover, although we present empirical results, it is extremely difficult to acquire complete and correct manuals for existing systems, whether or not in machine-readable form.

Note. We take 'system' to refer to the computer implementation of the user interface. We refer explicitly to the complete human-computer system inclusive of the environment as necessary.

2. The model represented by manuals

The user manual (a complete and correct user manual) represents a way of expressing the user's potential knowledge about a system. Its complexity must therefore correspond (if only weakly) with the cognitive complexity involved in the user interface (over all tasks covered by the manual). A very plausible corollary is that if a small manual cannot be written for a system, the system will be found difficult to understand and to use. Indeed, Carroll's minimal manual work has been called one of the most robust results in human-computer interaction (Draper & Oatley, 1992): and it is (put simply) that the smaller a manual the better. Of course Carroll's minimal manual approach includes other factors, such as error recovery. The point we need to make from his and other work is that the usability of the complete human-machine system depends, amongst many factors, on the complexity of the manual.

With on-line help, intelligent assistants, and so forth, there is actually very little reason for a user to distinguish between the system and its manual (figure 1). With multimedia and hypertext systems, in a real sense, the system *is* a manual.

That one is conventionally implemented in silicon and the other in paper is merely a convention, and the difficulty of editing paper (as opposed to electronic media) has resulted in the entrenchment of the view that manuals must be designed after systems. Paper manuals are certainly much harder to revise! It would follow from the conventional view that any measure of manual complexity has at best an impact on acceptance, but none on design.

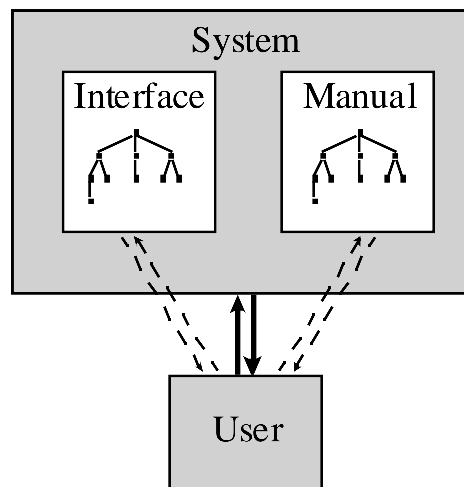

Figure 1: User's view of a system. How does the user distinguish interface from manual?

A manual, then, sits between the computer program and the user.

On the one side, computer program complexity measures are not very likely to relate to usability; but the manual is a function of the program, and since it is intended for the user, complexity measures of it are likely to be far more relevant to usability from the technical perspective. On the other side, cognitive measures are hard to relate to computer software design, they are easier to apply to manual design. In the middle, the manual defines (or is defined by) the system *and* is designed for human comprehension and use, so measures of its complexity will be relevant to usability, both from the system and from the user perspective.

Manuals are more accessible than user models and they are more relevant to user interfaces than programs as such. In short, they are a *concrete*, *accessible and shared* representation sitting between brain and silicon. Can we exploit this shared ground further?

3. The analogy between computer program and user manual

The analogy between computer program and user manual is based on the view that the user is a 'computer' and the manual is a 'program.' If you prefer, the computer (specifically, the processor) is a 'user' and the program is a 'manual.'

However, this view is a bit stark, and can be explained less abruptly: When a programmer writes a program, they do not anticipate what that program will be used for, rather they determine certain aspects of it, such as its control flow. Thus, a programmer does not need to consider what a user of a word processor writes, but how they perform their writing. Likewise, the designer of a manual does not prescribe precisely what the user does with the system (if they could precisely predict user behaviour then the system could automatically perform that task for them!), rather designers prescribe the methods and choices available to the user to achieve their task goals.

Thus, we may view a manual as a program. Like programs, manuals prescribe certain operations and procedures. The user in pursuing their own task's goals, execute steps from the manual, or navigate (when they are skilled or well trained) what is effectively a remembered route through the manual.

The question arises how one might measure or assess the complexity of manuals. Moreover to give such assessment as great an impact as possible, it must apply to manuals *as yet to be written*, to take advantage of the information at a stage in the design process where action can still be taken.

We propose to provide one answer to this question in this paper. Before giving our answer, however, we first provide several arguments leading up to it and together justifying it.

3.1. Argument 1

Over many years programmers have discovered that writing programs requires some discipline of thought. There are various approaches to program design, amongst which the use of structured programming constructs is the least contentious. Dijkstra (1968) argued in a landmark paper that the unrestricted use of the **goto** statement causes problems for the programmer. Specifically, when a sequence of program steps can jump anywhere or can be executed starting from anywhere else, it is very hard to keep track of what is intended and what is assumed by the various routes of program execution. The main reason for this is that no part of a program can be considered in isolation.

Since Dijkstra's article appeared, structured programming has become well-established. If someone writes an unstructured program, an immediate list of questions is asked: do they understand that program? Would anyone else? Does it do what they intended?

Considering the manual/program analogy, we ask corresponding questions:

- Is the manual likely to correctly describe the intended system?
- Is the user likely to be able to understand (or conceptualise) the system from the manual?

If the manual was a computer program, and the user was a computer, the answer to both questions is, "only if the program was properly structured" — only if the manual is a suitably structured manual.

If the program was not properly structured it is very likely that the programmer made mistakes because they did not have a proper grasp of what they were doing. By analogy, a manual is unlikely to describe the system and the user is unlikely to understand it if the manual is not properly structured. (Note that there is no problem on the one hand making the analogy that the user is another programmer understanding the source code of a program, and in the other analogy a human-computer executing the program. Both analogies produce the same answer.)

Summary: Argument 1 takes the view that writing a manual is a similar task to writing a program, and therefore a good manual (or one that is written correctly) is one that is well structured.

3.2. Argument 2

Hypertext is one way to represent documents, and user manuals are one example of the sort of document that could easily be made into a hypertext. Hypertext can be represented by finite state machines (or variations thereof). Thimbleby (1993) showed that since systems can be represented by finite state machines (or variations), one can place the manual and system into direct correspondence. One therefore obtains a correct (though typically large) manual.

It follows that good system design principles and metrics could equally well be assessed by examining this sort of manual.

Summary: Argument 2 takes the view that a manual should accurately reflect the system's program, and therefore measures (designed for evaluating program complexity) can be meaningfully applied to a manual.

3.3. Argument 3

Manuals are often improved by using clear English, careful exposition, pictures and diagrams. It may be the case, however, that these are cosmetic improvements and the manual could be more significantly improved by addressing its structure. Certainly, achieving a good style would be compromised by changes in structure, but not the other way around. Thus getting a good structure is the primary task, after which refinements can be made.

Summary: Argument 3 takes the view that a manual should, primarily, have a good structure. Unlike the preceding two arguments, Argument 3 does not require a 'programming' type of structure though it is consistent with one.

3.4. Argument 4

Finally, we note that many user manuals are presented as flowcharts or finite state machine diagrams. In some multimedia authoring systems, *Authorware Professional* (Macromedia, 1992) for example, the user's choices are planned using explicit flowcharts. These are not arguments that all systems can be documented by manuals which are flowchart programs, but it certainly shows that the analogy is sufficient and appropriate for some applications.

Summary: Manuals in many cases are already 'user programs.'

4. Structured programming

Structured programming aims to make programs more intelligible and more readable — easier to reason about. It also makes them easier to maintain by programmers other than the original author. Such programs are more likely to be developed correctly in the first instance. Structured programming is usually defined as constructing programs using a small set of well-understood control structures, which combine smaller structured program fragments together. The main control structures are the simple **if-then-else**, **while-do** and **repeat-until** statements and some others available in certain programming languages. The **goto** statement is normally excluded.

The general principle underlying the structured control structures is that they have one entry and one exit; a **goto** statement would provide an additional way out of such structures, and, of course, a corresponding way into other structures which are the destination of the **goto**. Structures compromised by a **goto** cannot be considered in isolation — the programmer has to understand the entire program; secondly, the programmer cannot cope by learning a few idiomatic control structures — with **goto**s, arbitrary control structures can easily be devised and simulated. In contrast, in proper structured programming, programmers become familiar with a small, standard set of control structures. Also, the one-entry one-exit restriction ensures that any component of a program can be replaced simply by a procedure.

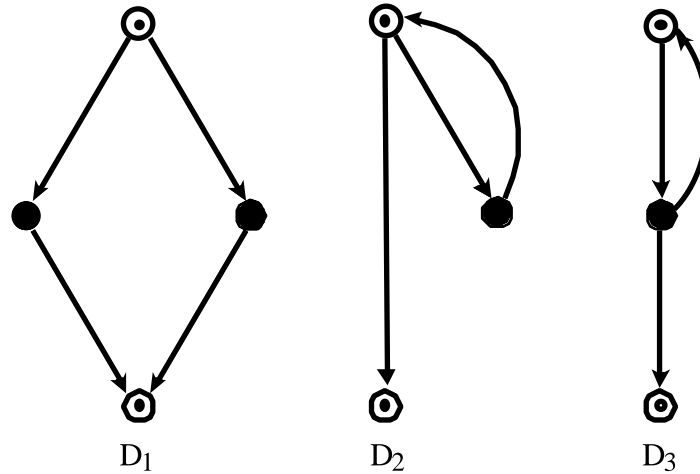

Figure 2: Flowgraphs D_1, D_2 and D_3 correspond to **if-then-else**, **while-do** and **repeat-until** respectively.

4.1. Flowgraphs: a particular application of the analogy

It has been proposed that the count of control structures used is a useful measure of program complexity. Unfortunately, any one program can be expressed in various ways, and a syntactical count of control structures is not adequate. However a program can be decomposed into a maximal set of structured components (Fenton & Whitty, 1986). Such a decomposition is unique and therefore complexity measures based on it are not sensitive to the original concrete form of expression of the program. The method allows for using **goto** statements to simulate various control structures perhaps which are not provided in the actual language being used.

Since user manuals rarely have any explicit control structures, and are almost entirely ordered by sequence and by **goto**s, flowgraph decomposition is an ideal approach to determine their complexity.

A flowgraph $F = \langle G,a,z \rangle$ is a triple consisting of a directed multigraph G, together with distinguished vertices a, z of G satisfying:

- a (the start vertex) has directed paths to all other nodes of G. The start vertex need not have zero in-degree.
- every vertex has a directed path to z (the stop vertex), which has zero out-degree.

It follows that a flowgraph is connected but not strongly connected. Figure 2 shows some standard control structures as flowgraphs.

A sequential program or manual can be considered a flowgraph, presuming only that it has a start state and that it terminates. If a program does not terminate (e.g., it is a continuous process), it can be converted to a flowgraph in various ways, the most satisfactory being to identify an ‘off’ or ‘idle’ state and split it into two adjacent states. Thus the graph without

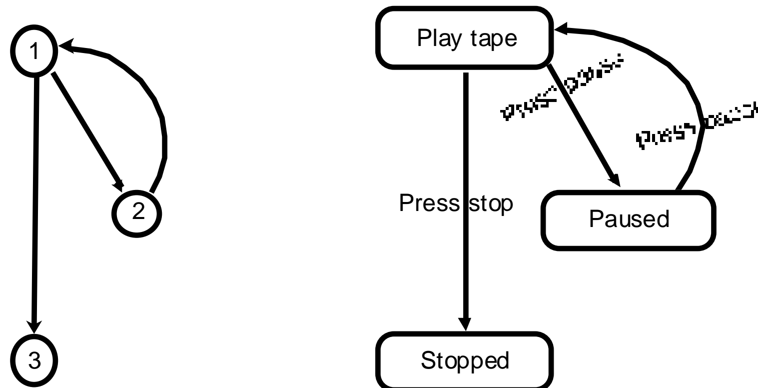

Figure 3: *Left*: flowgraph drawn abstractly.
Right: flowgraph drawn as part of the diagram of a tape recorder.

the new edge is a flowgraph, and each execution of the flowgraph is an execution of the program from idle_1 to idle_2 state.

The notion of flowgraphs captures crucial points of well-structuredness in programs. A structured control structure should have precisely one entrance (no **goto**s into it) and precisely one exit (no **goto**s out of it): hence, structured program constructs are certain flowgraphs. The simplest is sequence, as represented by the Pascal semicolon. The Pascal `a:=1; b:=2` can be represented as a simple two vertex flowgraph (`a:=1`)→(`b:=2`).

The following Pascal, with numbered lines, can be represented as a flowgraph (see figure 3):

while n < 10 **do**	… 1
n := n + 1;	… 2
write(n);	… 3

As suggested by figure 3, the same flowgraph can also be used for the following simplified extract from a user manual, where we have used the word '**goto**' to show how the user can move from paragraph to paragraph as the manual is read ('executed'). In many ways, this is merely a compiled **while** command, and is further support to the argument that users follow programs, very low level programs at that, which happen to be written as manuals.

Play:	press *Pause* to pause the tape **goto** Pause	… 1
	press *Stop* to stop playing the tape **goto** Stop	
Pause:	press *Pause* to resume playing the tape **goto** Play	… 2
Stop:	…	… 3

Any edge (u,v) in a flowgraph can be replaced by a flowgraph $\langle G,u,v\rangle$: this is a composition operation called *nesting*, and is how more complex programs are constructed, given the

appropriate building blocks. Sequential composition (in Pascal, using a semicolon) is simply nesting on the flowgraph P_2, which is •→•→•. For a given set of control structures, it is sufficient to consider nesting on vertices with one out edge (since the vertex uniquely specifies the edge to be replaced); however this entails treating as distinct each form of various program control structures (such as the **case** statement and guarded commands, which require vertices of arbitrary out-degree)

Given a flowgraph, a problem called *decomposition* is to determine how it might have been constructed from smaller, nested sub-flowgraphs. It is easy to prove that the maximal decomposition is unique.

To measure program complexity, a program is decomposed (using standard algorithms: see below) into the maximal set of flowgraphs. Various metrics are then possible: for example the smaller the components (or the greater the number of them for a given size of program), the more structured the program. The metrics can be normalised by the size of the program to obtain complexity densities. One can also immediately establish what control structures have been used, and easily indicate any that are non-standard.

5. An experiment

As an experiment, we took the finite state machine description of a submachine of a domestic video cassette recorder. This description can equally be viewed as a manual fully documenting the machine (Thimbleby, 1993). We took the off-with-tape-out state as start and end states (as described above) and hence obtained flowgraphs. We applied a flowgraph decomposition algorithm (Lengauer & Tarjan, 1979; Fuchs, 1993) to the graphs. The somewhat surprising result was that there was *no* proper decomposition. (A proper decomposition is a decomposition into at least two non-empty sub-flowgraphs.) Moreover, there was no proper decomposition no matter what state was taken as the start/stop state.

A more impressive way of saying this is that one might have saved a human manual writer from the impossibly tedious — and in this case, fruitless — job of searching for a well structured manual. Not one of the billions of possible *distinct* manuals that might have to have been examined by hand (by manual methods?) can be considered structured.

If we take manuals as isomorphic to directed spanning trees, the number above is a count of the distinct labelled directed spanning trees rooted at the start vertex of the relevant flowgraph. For our small example system the number is over 10 million for each root. Given that many everyday systems are far larger, the space of their manuals is astronomical, and automatic means of optimising manual design are necessary.

Of *any* group of paragraphs in any manual, more than one must refer to paragraphs outside the group, or elsewhere other paragraphs must refer to more than one inside the group. These correspond to **goto**s out of the group or **goto**s into it, not counting the start and end of the group.

If the manuals were to be made more verbose or more extensive by documenting states as sequences of paragraphs (instead of one paragraph per state, or one paragraph per state×button, as assumed above), then the flowgraph decomposition would at once decompose these as P_i subgraphs (i.e., strictly linear flowgraphs with no branches); the quotient graph would then not be further decomposable. That no such P_i were found in the original case shows that, in user interface terms, there were no states in which one could only next do one thing and that one thing could be done no other way.

What the results mean, especially give that one can modify the system to be documented in various ways (e.g., by not writing a manual for all of it, as we assumed here), will be considered more fully below.

6. Discussion

If a large computer program was found to have no proper flowgraph decomposition, one would justifiably complain to the programmer that it was unstructured; it is likely that the programmer could not easily explain what the program did; it is likely that nobody else could understand the program at all easily; and it is likely that the computer anyway does something other than what the programmer intended. The implications of unstructured program writing are serious.

If the manuals for the VCRs we studied were written by humans, and we view them as 'programs' for their users to achieve the various functions the VCRs implement, then there is a very strong possibility that those writers did not understand what they had done, or, rather, they would not know what a user would do with what they had done.

A programmer may 'know' what he or she has programmed or intended to program, but they may be surprised at what the computer does: this is why debugging programs is so difficult. Likewise, users may not apply the manual instructions as intended (or any aspect of the design as intended): this is why user interfaces need to be 'debugged' in real tests with users.

What they had done was likely to be incorrect in details, and, further, that they would be unable to explain what they had done. Even if by some chance the manual was correct, it is extremely unlikely that anyone else would be able to maintain it. All this is the programming experience. In user interface terms, additionally it means that there would be little reason to suppose a user could understand the manual either. (There is little debate about whether computers understand their programs!)

The structured programming experience is that trained programmers have difficulty reading and comprehending badly structured programs: how much harder would users (untrained as programmers) find unstructured manuals?

A crucial point is the following: If there is no proper flowgraph decomposition, then a reader of the manual *cannot* use divide and conquer to help understand the manual. No part of the manual can be considered in isolation (i.e., 'dividing' the manual results in components that are incorrect or incomplete).

A solution to the correctness problem of manuals is to construct the manual automatically. This solution has been proposed and demonstrated by Thimbleby (1993), though this does not directly address the problem of its complexity. (Arguably it exacerbates it, since it guarantees arbitrarily complex manuals are correct.) Since the decomposition of a manual into flowgraphs is unique and independent of its syntactical form, its complexity (in this regard) can only be improved by changing the system. Thus one has to use a system development environment where such complexity measures can be established early enough in the design cycle so that they can impact the design itself; this was also proposed in Thimbleby (1993).

6.1. Limitations of flowgraphs

Flowgraphs are just one of many ways of approaching the manual/program analogy. They have limitations for both software engineering and manual analysis. An important feature,

however, is that the limitations are very precise. Flowgraphs not only provide an approach for measuring complexity but also provide a demarcation for the factors being measured.

Flowgraph based analysis will be particularly relevant when the user is following a completely documented procedure, as in error recovery, fault diagnosis or servicing applications. In applications where the manual is intended to lead to understanding of system principles, other approaches would seem more appropriate. Flowgraphs are not specially appropriate for declarative manuals (ones that declare properties or invariants of the user interface, as opposed to specific procedures).

Although there is no flowgraph decomposition in the example studied, one might respond that if certain edges were deleted then a structured manual could be obtained. For example, the Operate button switches the VCR off [sic] in many (but not all) states. Deleting edges to off would reduce the number of **goto**s out of many states. In fact, one can search for the 'best' edges to delete; and the manual would have to discuss these separately, as well as any exceptions (e.g., Operate does something different when the VCR is off). What is most interesting, however, is that a purely automatic procedure (flowgraph decomposition) has identified task-related issues, in turn which would raise various design trade-offs.

Orthogonal components are common in user interfaces (e.g., a TV sound on/off component is usually orthogonal to the TV channel control), but flowgraph decomposition only considers nesting, not Cartesian product. (One can consider flowgraph decomposition of orthogonal components: essentially applying flowgraph metrics to independent parts of the manual separately.)

Undo causes similar problems; indeed undo is not handled very well by finite state machines — push down automata are better models.

6.2. Aren't all programs equivalent to structured programs?

Böhm and Jacopini (1966) showed that any (conventional imperative) computer program can be converted to [what we might now call] a structured program using a limited number of simple control structures. In their original terms, they showed that however complex a flowchart or Turing Machine program, it can always be converted to a program using a few (specifically: 3 by their first result, or 2 by their second) simple control structures. Their result was an important one in convincing the programming community that there was no program that had to be written in an unstructured form.

It is sometimes forgotten that their result showed more precisely that any program can be converted to a structured program perhaps only by way of introducing state variables. For example any **while** loop from which you want to do an exit in the middle can be converted to a well-structured **while** loop, with a body that is a simple conditional (see figure 4), by introducing a state variable.

In manual-writing terms, this means that *any* manual can be converted to a 'structured manual' but perhaps only at the expense of introducing one or more variables. One is thereby shifting the complexity of the manual's *flow* structure to the cognitive load of remembering the states of the respective *variables*. It may be that a few variables (7±2!) are an acceptable trade-off, though it seems more sensible for the system to display the state of the variables at the appropriate places by using indicators — though this approach would exacerbate the standard problem of system/manual synchronisation.

```
while test do begin                     flag := true;
        do1;                            while test and flag do begin
        if stop then goto cheat;                do1;
        do2                                     if stop then flag := false
end;                                            else do2
cheat:                                  end
```

Figure 4: Using state variables to permit structured programming. The program fragments shown are equivalent (assuming flag is not used elsewhere!), but the one on the left is not flowgraph decomposable.

As Dijkstra (*op. cit.*) noted for the Böhm and Jacopini result, a translation of a bad structure by their mechanical method to one that avoids **goto**s does not necessarily achieve a program that is any clearer. We are not suggesting our approach helps write or generate better manuals; it merely measures something that indicates whether a manual may be badly written. That programs can be well-written without **goto**s is now a matter of history, and, as with manuals, it is essentially a stylistic rather than a theoretical issue.

7. Conclusions

We have suggested that the design of (and evaluation of) user manuals may be approached using methods derived from software engineering. This approach suggests various approaches and metrics, which can be taken directly from software engineering. Moreover, the arguments that justify programming metrics can be applied with equal force in the user manual case. Program complexity measures correspond (amongst other features) to the likelihood of correctness and intelligibility of manuals, both of which one would usually wish to optimise simultaneously! It follows that manuals may be improved by such methods; designers and manual writers can now easily *question the trade-offs involved in unstructured features of a user interface*.

The appropriateness or otherwise of a particular metric will depend on the user's tasks and what should be optimised by the design. Specifically, this paper considered flowgraph decomposition, which is only one of many plausible complexity metrics. From software engineering, they are known to be a good measure of complexity from the writer's and maintainer's points of view. An important advantage of flowgraph-based complexity measures is that they can be automated.

Just as some computer programs are best written in an unstructured way (for example, because efficiency is paramount, or because they are microprogrammed and use some unconventional ideas of control flow), it is certainly not the case that structured methods are a universal panacea. Some user interfaces will benefit from structured manuals, some won't. Games, teaching systems (CAL, CBT) and high security systems are common examples of systems where the user is *supposed not* to understand everything; on the other hand, safety-critical systems, office systems and many consumer products are generally supposed to be ones that are easy and reliable to use under the sorts of assumptions that our approach makes. Users should therefore have structured manuals, well structured in the software engineering sense. In this paper we have provided some initial ideas on how that may be achieved.

Acknowledgements

This research work has been conducted as part of an SERC-supported project "Systems, manuals, usability and graph theory," Grant No. GR/J43110.

References

BÖHM, C. & JACOPINI, G. (1966) "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," *Communications of the ACM*, **9**(5), pp.366–371.

CARROLL, J. M. (1990) *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*, MIT Press.

CARROLL, J. M. & AARONSON, A. P. (1988) "Learning by Doing with Simulated Intelligent Help," *Communications of the ACM*, **31**(9), pp.1064–1079.

CARROLL, J. M., SMITH-KERKER, P. L., FORD, J. R. & MAZUR-RIMETZ, S. A. (1988) "The Minimal Manual," *Human-Computer Interaction*, **3**(2), pp.123–153.

DIJKSTRA, E. W. (1968) "Go To Statement Considered Harmful," *Communications of the ACM*, **11**(3), pp.147–148.

DRAPER, S. W. & OATLEY, K. (1992) "Action Centred Manuals or Minimalist Instruction? Alternative Theories for Carroll's Minimal Manuals," in *Computers and Writing, State of the Art*, pp.222–243, HOLT, P. O'B. & WILLIAMS, N., editors, Intellect Press.

FENTON, N. E. & WHITTY, R. W. (1986) "Axiomatic approach to Software Metrication through Program Decomposition," *The Computer Journal*, **29**(4), pp.330–339.

FUCHS, N. (1993) "An Improved Algorithm for Deriving the Decomposition Tree of a Flowgraph," *Structured Programming*, **14**(3), pp.93–101.

JACKSON, M. A. (1983) *System Development*, Prentice-Hall.

LENGAUER, T. & TARJAN, R. E. (1979) "A Fast Algorithm Finding Dominators in a Flowgraph," *ACM Transactions on Programming Languages and Systems*, **1**(1), pp.121–141.

PAYNE, S. J. (1991) "Display-based action at the user interface," *International Journal of Man-Machine Studies*, **35**(3), pp.275–289.

RUNCIMAN, C. & HAMMOND, N. V. (1986) "User Programs: A Way to Match Computer Systems and Human Cognition," *Proceedings of British Computer Society Conference on People and Computers: Designing for Usability,* pp.464–481, HARRISON, M. D. & MONK, A. F., editors, Cambridge University Press.

THIMBLEBY, H. W. (1990) *User Interface Design*, Addison-Wesley.

THIMBLEBY, H. W. (1993) "Combining Systems and Manuals," *People and Computers*, **VIII**, pp.479–488, ALTY, J. L., DIAPER, D. & GUEST, S., editors, Cambridge University Press.

MACROMEDIA (1992) *Authorware Professional Tutorial,* Macromedia Inc. CA.

WOLFRAM, S. (1991) *Mathematica*, 2nd. ed. Addison-Wesley.

Appendix

The following *Mathematica* (Wolfram, 1991) definition is the submachine of the JVC HR-D540EK VCR used as an example in the paper. Its interpretation is as follows: States are numbered from 1. In state `stateLabels[[s]]`, pressing button `buttonLabels[[b]]` changes state to `transitions[[s,b]]`, or does nothing if this is zero.

```
{buttonLabels -> {  "Play", "Operate", "Forward", "Rewind", "Pause", "Record",
                    "Stop/Eject", "Tape in"},
```

```
 stateLabels   -> {  "fast Forward", "off Tape In", "off Tape Out", "on Tape In",
                     "on Tape Out", "play Forward", "play Pause", "play Rewind",
                     "play Tape", "record 030", "record 060", "record 090",
                     "record 120", "record 150", "record 180", "record 210",
                     "record 240", "record Pause", "record Pause 030",
                     "record Pause 060", "record Pause 090", "record Pause 120",
                     "record Pause 150", "record Pause 180", "record Pause 210",
                     "record Pause 240", "record Tape", "rewind Tape"},

 transitions   -> {  {0, 2, 0, 0, 0, 0, 4, 0},
                     {0, 4, 0, 0, 0, 0, 3, 0},
                     {0, 5, 0, 0, 0, 0, 0, 4},
                     {9, 2, 1, 28, 0, 27, 5, 0},
                     {0, 3, 0, 0, 0, 0, 0, 4},
                     {9, 2, 0, 0, 7, 0, 4, 0},
                     {9, 2, 1, 28, 0, 0, 4, 0},
                     {9, 2, 0, 0, 7, 0, 4, 0},
                     {0, 2, 6, 8, 7, 0, 4, 0},
                     {0, 2, 0, 0, 19, 11, 4, 0},
                     {0, 2, 0, 0, 20, 12, 4, 0},
                     {0, 2, 0, 0, 21, 13, 4, 0},
                     {0, 2, 0, 0, 22, 14, 4, 0},
                     {0, 2, 0, 0, 23, 15, 4, 0},
                     {0, 2, 0, 0, 24, 16, 4, 0},
                     {0, 2, 0, 0, 25, 17, 4, 0},
                     {0, 2, 0, 0, 26, 27, 4, 0},
                     {27, 2, 0, 0, 0, 10, 4, 0},
                     {10, 2, 0, 0, 0, 20, 4, 0},
                     {11, 2, 0, 0, 0, 21, 4, 0},
                     {12, 2, 0, 0, 0, 22, 4, 0},
                     {13, 2, 0, 0, 0, 23, 4, 0},
                     {14, 2, 0, 0, 0, 24, 4, 0},
                     {15, 2, 0, 0, 0, 25, 4, 0},
                     {16, 2, 0, 0, 0, 26, 4, 0},
                     {17, 2, 0, 0, 0, 18, 4, 0},
                     {0, 2, 0, 0, 18, 10, 4, 0},
                     {0, 2, 0, 0, 0, 0, 4, 0}}
}
```

Note. This transition table was determined by experiment from the device, and not derived from the manual. We have included it here for several reasons: it is actually very tedious to determine; it is good practice to include experimental data; we may have made mistakes. This last point is most interesting: of course, we don't think we've made a mistake — the specification behaves, so far as we can tell, in a 'reasonable' way. But it may be incorrect. This is a central problem of user interface design: we think we have a user interface design (here, a simulation of a VCR), we have tested it, yet we may still be wrong. This serious problem arises in our case because manufacturers do not specify VCRs formally.