Intelligent adaptive assistance and its automatic generation

Harold Thimbleby† & Mark Addison‡

† School of Computing Science
Middlesex University, Bounds Green Road,
London, N11 2NQ

‡ Philips Research Laboratories
Cross Oak Lane
Redhill, Surrey
RH1 5HA

Phone: 0181 362 5000, 0181 362 5302,
Email: harold@mdx.ac.uk, addison@prl.philips.co.uk
URL: http://www.cs.mdx.ac.uk/, http://www.philips.com/

Abstract
Manuals and interactive help are tedious to provide, difficult to maintain, and difficult to ensure correct, even for simple systems. The result is a loss in product quality, felt particularly by users and designers committed to long-term product development.

This paper shows that it is possible to systematically put a system specification and its documentation into exact correspondence. It follows that much previously *manual* work can be done automatically — and with considerable advantages, including guaranteed correctness and completeness, as well as supporting powerful new features such as intelligent adaptive assistance. This paper shows how interactive assistance can be provided to answer “how to?”, “why not?” and other questions.

Keywords
Assistance, help, training, user manuals, system design, finite state machines, HTML.

1. Introduction

Interactive systems — cash machines, control systems, aircraft cockpits — should be delivered to users with complete and correct documentation, and if clients require system modifications, it should be easy to update the documentation correctly. Context sensitive assistance should be provided so the users can make the best possible use of the system, whether to learn how it works or to recover from error conditions. Wouldn’t it be nice to be able to do all this in a reliable, integrated and efficient fashion?

Interactive help is not always relevant, and can interfere with the proper use of the system. Many systems are supposed to be sufficiently easy to use (e.g., ‘walk up and use’ systems, such as ticket machines), or to be

used by highly trained users (e.g., aircraft flight management systems), that manuals and help would be a distraction. What is important in these cases, however, is to design the system so that it is easy to understand and to use; this may be done by designing the system so that its interactive help, *if it had been supplied*, would have been minimal (Carroll *et al.*, 1988).

Interactive products are often complex and difficult to use, usually requiring additional material for their successful operation. Their manuals are *therefore* inevitably bad — manuals may have presentation problems as well, of course, but if they correctly and completely document a bad system along with its idiosyncrasies, they aren't going to be easy to understand. Addison and Thimbleby (1994) showed that some systems are impossible to document suitably, because they have been badly designed: documenting these systems will inevitably result in poor manuals. These findings suggest some manual problems arise from bad design, and the designer should employ the results to re-design the system.

There is a considerable literature (notably Carroll, 1990; Carroll & Aaronson, 1988) on user manuals. Our approach is more technical, to show *how* manuals and interactive help may be provided through an integrated system. We are not presenting a philosophy or psychology of help: we present an effective technique for providing reliable documentation and adaptive interactive help features.

Our approach simplifies the engineering and design of systems by providing a framework for developing the complete system, including the manual. The approach is formal, based on graph theory, and consequently can be systematically applied, and technically is relatively easy to implement. The benefit of ensuring integrity between the manual and the system is important for developing user-centred system, when the design is performed iteratively, since otherwise the iterations in the design could easily cause version-control errors in the manuals and other user aids.

1.1. A demonstrator system: HyperDoc

We have built a prototype interactive system design tool, called HyperDoc (Thimbleby & Addison, 1995) to facilitate automatic construction of user help and to do various analyses of interface design. HyperDoc enables accurate, context-sensitive and (if required, adaptive or task-oriented) help to be provided in various forms for the user, at very little cost to the system implementors.

Modern complex systems are notoriously hard to document either completely or correctly. The approach we describe in this paper puts an end to this. (Of course, there are many deliberately difficult to use systems: security, games, and some control and safety critical systems that must only be used by qualified users.) If a push-button or menu based interactive system cannot be handled at an informative level by HyperDoc, or a tool like it, then we would suggest the system is likely to be too complex for conventional use in any case.

HyperDoc was conceived as a practical HCI tool for developing and researching interactive systems, and consequently is concerned primarily with the design and analysis of user interfaces and their manuals. It supports a fully integrated design environment for simulating interfaces

and for writing the user documentation (manuals) associated with system states. The concept of placing a system and its documentation in direct correspondence is simple, but the resulting paradigm is very powerful. The system has evolved many user-oriented facilities — derived from graph theoretic operations — and directly maintains integrity between the system and its documentation. In this paper, we describe its ability to demonstrate and investigate interactive help.

HyperDoc provides a tightly coupled environment that readily supports design; and the device being developed can easily be modified at any stage of development without requiring significant resources. HyperDoc supports four basic elements for expressing the interactive system 'buttons', 'states', 'indicators' and 'documentation'. HyperDoc can be viewed as a system containing its manual or as an operational manual representing the system being designed. HyperDoc can present manuals interactively, as context-sensitive help (described later), and of course automatically generate conventional paper-based manuals. New systems are constructed or re-structured by the addition, deletion and revision of such objects and HyperDoc can easily support such operations. The design/simulation environment of HyperDoc is *the same* for both user and the designer. The user and designer share the same environment and consequently the designer is able to perceive the precise user environment and create an appropriate discernible interface.

1.2. Finite state machines

We are concerned with push-button machines with discrete states: pressing buttons changes the system's state. To support the assistance mechanism, states, buttons and state transitions are annotated in natural language. (Default descriptions, based on button and state names, are provided by HyperDoc if no specific natural language descriptions are provided.)

Such an interactive system can be viewed as a finite state machine (FSM), perhaps with time-outs. Time-outs can be handled like push-buttons: instead of saying "Push the Record button" the system says "Wait 5 seconds" or "Wait until July 4" — *internally* the system's structure is the same, just the terminology differs (cf. Halbwachs, 1993).

Many systems (particularly computer systems) have large keyboards and we can deem classes of keys (e.g., all alphanumeric) to be equivalent for the purposes of manuals and user help, to obtain a small and easily-managed finite state description of the system. Indeed, if a system is not a 'small' finite state machine (under some such abstraction), it is likely too complex for human use — with or without help! Certainly its complete manual would be very lengthy.

Another relevant form of abstraction that is convenient in defining complex FSMs is the statechart (Harel, 1987, 88, 90), which (ignoring features like broadcast communication) define FSMs using hierarchical, default and orthogonal structures. The advantage of a statechart for design is the greatly reduced number of states that need be defined explicitly (also making the notation much clearer). A statechart, being a more concise representation, can be annotated more easily than the FSM it defines, but one could easily generate the appropriate annotations for the defined

machine. In fact, if the structure of the statechart clarifies the system structure (for the designer), its annotation might be more informative than that of the expanded FSM. In this case, the 'expanded' annotation of a state could be the annotation of the hierarchy the state is in, rather than the (lengthy and detailed) product of the state's annotation and other orthogonal states' annotations.

For clarity this paper will assume *every* state has annotations, though in practice one might not annotate specific states, preferring instead to generate annotations algorithmically from combinations of other annotations. The important point is that a user, in any case, should not be able to tell the difference between a system implemented as a FSM, a statechart or other form: whatever its implementation technique, it should work correctly to its specification.

We also annotate transitions, in fact for each button in every state, regardless of there being a state change on the transition. This feature is useful to specify pre- and post-conditions for performing the transition correctly. Consider a simple fire alarm system, with two buttons FIRE and RESET. The RESET transition may have a pre-condition "Check the fire is extinguished," which could be generated as an assistance message.

Since HyperDoc is based on finite state machines, it is easy for it to generate other representations of finite state machines. Notably HyperDoc can directly produce complete system manuals in HTML (hypertext mark-up language, the language used for the World Wide Web). Such manuals are both normal function-based manuals and they are active hypertext, which a user can explore: by clicking buttons in the hypertext manual, the manual automatically goes to the appropriate places to continue reading about the relevant system features available had the button been pressed on an actual system.

1.3. Immediate productivity gains

It is tedious to document a complex system. Often documentation remains incomplete or incorrect, simply because of the complexity of the task facing technical authors, and sometimes, because of production pressures not leaving enough time given the scale of the work required.

In conventional documentation, 'everything' has to be documented. With a system like HyperDoc, the designer of the system need only document S states and B buttons, requiring a total of S+B sentences or paragraphs. Yet HyperDoc then provides help for all S^2 routes between all pairs of states, i.e. all button-pressing possibilities in all states — another S×B paragraphs — and many other sorts of paragraphs for explanation purposes (see section 3). This amounts to a very considerable saving even for trivial interactive systems: for example, a very simple 10 state machine handled this way requires only 10 paragraphs to be actually written and the B buttons named; the system is then able to generate all 100 potential state transitions and additional text components required to answer a range of user questions (see section 3). This saves the designer writing or anticipating at least 90 answers, just for the simple cases, and the saving becomes more dramatic as the number of states (S) increase.

A concomitant improvement in the quality of the documentation is obtained, since the designer (or manual writer) can more efficiently proof

read the fewer sentences and is more likely to get them correct. And since the manual is so easy to generate, users can become more involved in earlier stages of the design process.

2. Equivalent designer, user and technical author environments

The specification of an interactive system, as a finite state machine, can be represented as a labelled directed graph (Parnas, 1969; Thimbleby, 1993). Each state has an associated natural language description, and each transition has an associated button name. A system may have indicators, such as on/off lights. The status of each indicator is recorded in each state. Some indicators may be *conceptual*, that is they are not visible to the user but simply serve to identify classes of state.

For example, a design option for a video recorder, with the object of simplifying the interface panel, may not include a fast-forward indicator light (because the winding mechanism is audible), but it may still be useful to generate a manual as if it had: because fast forward is a salient condition to the user. Conceptual indicators are thus a convenient representation for the designer to express design considerations without affecting the current user's perception of the system.

A finite state machine specification is trivial to animate; indeed, one can go directly from the specification to an implementation for any appropriate hardware. However, as a user operates the machine, the annotation of states is still available and can be used to provide help. For example, to answer the question "What can I do here?" at its simplest is merely a matter of collecting the annotations from the adjacent states and saying which buttons should be pressed to achieve those states. Or, to answer the question "How do I …?" at its simplest is merely a matter of determining the best route from the current state to the required state, and using the annotations to document the route to be taken by the user.

If instead a system designer or manual writer is using the system, the answers to such questions need only be editable text for HyperDoc to become a very powerful manual development tool. Because of the quantity of annotations that are involved it is however helpful to have some facilities to help the designer manage the text editing, for instance to record which states have, as yet, no annotations defined. (An indicator is automatically set in such states, which makes them particularly easy to find.)

That the user's system and the designer's system are identical has very many advantages and certainly increases the chances that the system will have a sensible design; it will necessarily have been used by at least one user! But more importantly, if the designer modifies the system, by adding a state, deleting a state, or by modifying a button transition, the manual *necessarily* changes in exact correspondence. Conversely, if a section of annotation is deleted, the state is deleted.

In summary: the system and its manual are the same thing; the design environment and the running system are the same thing; the user's help facilities and the designer's manual-writing facilities are the same thing. (The *active* hypertext manual is another side of this equivalence.) These equivalences make HyperDoc very elegant.

The correspondence with hypertext is not to be ignored: a system's manual could be a hypertext. In the prototype of the system (Thimbleby, 1993), the top of the screen was the simulation of the system, and the bottom of the screen the corresponding state's manual material: hence there was no distinction between using the system or using its manual.

The prototype was implemented for HyperDoc, and was implemented in HyperCard, a general purpose hypertext programming environment (Coulouris & Thimbleby, 1992). HyperDoc is now implemented as a Macintosh application using Symantec's C programming environment; it is available from the authors.

3. Answering Questions

There is a need for interactive systems to provide mechanisms that support users in many different ways; asking questions of the system (and getting it to answer them) is a highly desirable feature that when implemented correctly and consistently greatly enhances usability and user effectiveness.

The system is doing something and the user wants to know how the system can be made to do something else. Generally, the system is currently in a state c and the user can ask how to attain a goal or terminating system state t, or more generally how to attain the 'easiest' or 'best' of any of a set of goal states, $t \in T$.

The user can specify the set T in many ways; the simplest is by menu (giving a choice of appropriately named tasks), the user merely selects one item from among its choices. Each choice selects a predefined set, which the designer or manual writer has named appropriately. A device's indicator lights are supposed to inform the user of useful states or state changes, so expressions involving indicator status form a natural way of specifying these sets for the designer. Additionally, the conceptual indicators of HyperDoc permit additional classes to those actually visible on the device, and these can be set 'on' or 'off' in any way that is convenient. (Conceptual indicators are similar to normal device indicators except they are not visible to the user.)

Often it is important to attain a termination goal either avoiding or maintaining certain conditions. To support this, both a termination set T and an invariant set I are in fact specified through a design matrix (Figure 1) and selected via the user's assistance menu.

For example, a "How to?" question is answered by finding any sequence of states:

- starting in the current state c
- staying in states in I (the invariance set)
- terminating in any state in T (the termination set)

In the case $c \notin I$, there is no answer to the user's question, because no valid state transitions exist. This potential problem can be picked up by the designer (with tools like HyperDoc), though there are cases where it is appropriate for there to be no assistance (except to explain why the task

cannot be performed) for the user who asks how to perform impossible or dangerous tasks.

Suppose a video recorder is playing a tape. We wish to ask how to remove the tape from it, and to finish with the machine switched off. Further, we do not wish to switch the machine on and off unnecessarily until the task is complete.

- c is the current state, "playing a tape."

- I is the set of states with the 'on' indicator true. That is, what ever is done, keep the machine on.

- T is the set of all states with the 'tape' indicator false and the 'on' indicator false. That is, finish when the tape is out and the machine is off.

Figure 1 shows part of HyperDoc's task design matrix, depicting the necessary conditions for this query. The left-hand column represents named indicators, other columns support three invariance conditions 'on', 'off' and 'same' (the indicator assertions which exist from state to state); and four termination conditions 'on', 'off', 'same' and 'flip' (representing the indicator conditions in the terminating state) — 'flip' simply permits a change of indicator status, changing from 'on' to 'off' or from 'off' to 'on', to cause termination.

	I on	T on	I off	T off	I same	T same	T flip
on	☒	☐	☐	☒	☐	☐	☐
tape	☐	☐	☐	☒	☐	☐	☐
play	☐	☐	☐	☐	☐	☐	☐
record	☐	☐	☐	☐	☐	☐	☐
pause	☐	☐	☐	☐	☐	☐	☐

Figure 1. Setting termination goals (T) and invariants (I) with reference to indicator status. The termination goals are shown on-screen in red (i.e., 'stop') and the invariants in green (i.e., 'go').

Figure 1 shows the conditions necessary for "get the tape out and switch off" on the assumption that it is best to leave the machine on until the last moment. The invariant requires the system to be 'on' all the time, and the termination condition is that the tape is out and the machine is off. Note, in HyperDoc such queries can be named and the user is expected to use these task names rather than explicitly using the task design matrix, as the designer does.

3.1. "How to?"

A "How to?" question is easily answered by finding a path in the finite state machine, from the current state to the required termination state(s). For example the shortest path, each transition weighted equally, gives an answer that requires the least number of button presses. We might require a path that changes the least number of indicators, or a path that never switches the system off, or a path that visits states where the user gains

rewards. There are very many possibilities. We will discuss some variations below where the costs vary according to the user's recent use of the system.

3.2. "Why not?"

A frequent cause of frustration is that an interactive system fails to respond as the user thinks it should. The user expected it to go into one state but it is in another. The natural question is "why isn't the system in the expected state?" For the designer, answers to such questions can effectively contribute to the design process, by supporting an iterative design cycle and providing perfective maintenance through redesign.

It is easy to answer "why not" questions. If the user's expectation was reasonable, then there would have been a button press in a previous state that achieved (or more directly led to) the desired state. One way to answer a "why not" question requires finding the best path from the *previous* state not the *current* state, and phrase the answer as "you should have pressed X not Y." Actually, there may be other reasons why the current state is not the one the user might have expected: the target state may require several more button presses to attain, for instance. Hence a general approach to answer "why not" questions is to find the least cost path from the current state from the system's previous state.

Figure 2 shows a simple example, of a user having made a selection where the machine does not enter the user-desired state. The user reasonably wants to ask why the machine did not enter the desired state. If values of weights are equal then transitions 'A' = 'B' = 'C', and the two equal shortest paths are

i) along 'C', with answer:

You have not yet pressed 'c'

ii) along 'B', with answer:

Instead of pressing 'a' you should have pressed 'b'

Answer (*i*) is more practical and (being easily recognised as such) is to be preferred, though the assistant could easily explain all equal routes. The approach generalises naturally when the arcs 'B' and 'C' are multi-step paths.

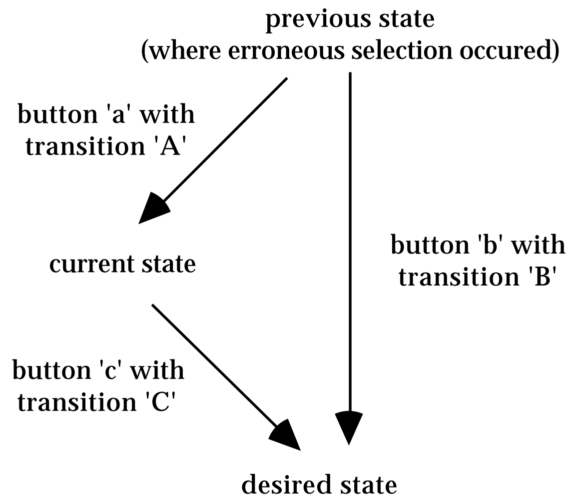

Figure 2. Simplified state diagram to illustrate "why not?" answer generation.

The assistant's search should not reach too far back into the past, otherwise the user will be presented with answers that refer to states too long ago to be relevant to their current situation. (In the simple example above, of the two equally weighted answers, the best answer was the one that reached back least into the past.) In fact, to generate satisfactory answers, we found during the development of HyperDoc by trying alternative strategies, only the previous system state need be considered, though of course, in some applications, say with professional users or teams of users (where one user wants an explanation based on a previous user's actions), there will be requirements to delve further back. One possible extension would be to introduce a time-stamping history of states as they are visited.

When states are error conditions, the assistant must avoid perverse answers. Provided error conditions can be defined, perverse answers are easily avoided by weighting edges out of error states very highly; in other words, the answer generated would rarely or never carry on from a mistake, and advise avoiding making one in the first place. On the other hand, from the designer's point of view, if "why not?" and "how to?" answers often involve error states there is probably something wrong with the structure of the system and hence the design. This type of problem is easily detected by HyperDoc, and hence would encourage redesign.

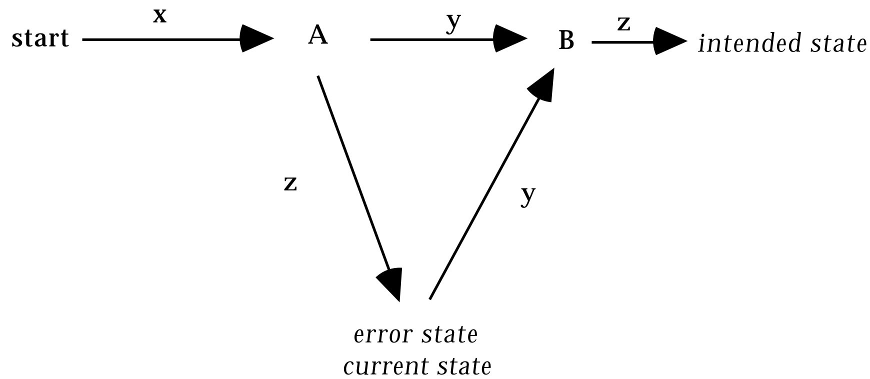

Figure 3. A simple subsystem containing a possible “error” state (here error can mean ‘costly’, ‘dangerous’ etc.)

Figure 3 shows how these ideas work: the user asks “Why not *intended state*?” when the system is in the state identified as *error state*. The equal-weights answer would be: “Your press ‘z’ was correct, but you now need to press ‘y’ (to get to ‘B’) then ‘z’ again.”* But the press ‘z’ was *not* correct! Instead, by weighting the transition ‘y’ from the error state highly, the answer becomes: “Instead of pressing ‘z’ (when you were in ‘A’), you should have pressed ‘y’ to do ‘B’.”

Of course, recovery from an error state might take us closer to the goal state than by any other route, even with a high (but not infinite) weighting: but this is likely to be a result of poor system design — the assistant’s advice would be correct. Importantly, the existence of such poor answers can be identified very easily automatically, because error state transitions can be traced between pairs of states in the finite state machine, permitting them to be designed out of the final system.

3.3. “Why?”

Why is the system currently performing a particular activity; why is an indicator in a particular on/off condition? To answer this sort of “why” question, the system need only maintain a record of the most recent button press that changes the status of each indicator.

For example, “why is the machine recording (why is the record indicator on)?” might obtain the answer “The record indicator is on because you pressed Record.” Sometimes systems simulated by HyperDoc provide bizarre answers: to the question “why isn’t it showing the pause light?”, one video recorder we investigated generated the assistant’s reply, “because you pressed Play, which made it Record”! On this video recorder, the Play button is confusingly overloaded, used both to play a tape and to remove the video recorder from being paused. A tool like HyperDoc can list all possible assistant’s answers and a designer would obviously look out for confusion of this sort.

3.4. “What now?”

Users may be unaware of particular states until they accidentally or inadvertently perform an operation that enters one of those states. This

* The word ‘again’ is easy to add, and makes the assistant’s English appear more fluent as well as making the instructions easier to understand.

can result in confusion (the user is uncertain of how to proceed) or gratification (the user has discovered a new and possibly useful feature). In the case of confusion there is an associated feeling of anxiety, how do they recover from their current predicament "what can I do now?" or "how do I extricate myself from this position."

The first question is easy to answer: simply list the annotations of the states adjacent to the current state. Alternatively the user might want an answer phrased in terms of the (actual or conceptual) indicator status of these states.

The second question poses much more subtle problems, which we consider next.

3.5. *"How do I go back?"*

Undo is a special help request, so common it has a special name. "Whatever I was doing a moment ago, I'd rather be doing that than what I am doing now. Please *undo* my last action!" This sort of assistance is therefore easy to specify: it only requires one button labelled UNDO, or one menu choice in the "How to?" menu. As undo is a frequent need, it might seem surprising so few devices capable of supporting undo actually provide the facility.

Undo may mean more than simply entering the previous state: there may not be a direct transition from the current state back to the previous state (although there must have been a transition from there to the present state). Thus the answer to an undo request can be complex, and can provide the choices we discussed above for other forms of assistance. It may be that the user may not wish to proceed with an undo when the consequences are fully understood; for example, it is possible that an undo requires taking the system through an undesirable state (such as off) and the user may prefer to proceed in some other, more controlled, way than simply undoing to the previous state.

Unfortunately undo raises serious design problems. Which we believe have been largely ignored, perhaps because properly conceived undo is generally so difficult to implement, and restricted implementations forbid the situations arising we wish to address; see Thimbleby 1990; and Dix & Abowd, 1994. What should undo mean just after using an undo button? Many systems undo the undo; some undo the previous transition, up to some fixed limit of transitions the system records.

Consider the following scenario. A life support machine (e.g., a ventilator) is switched on. Pressing some button 'X', on the interface panel, the operator can cycle the machine through a ring of n states (6 in Figure 4), that provide different sorts of clinical features, 'a' to 'e'. Just after press $n+1$, i.e. in state 'on' in figure 4, the operator presses the undo button. What should undo do? It seems uncontentiously it should enter state 'e'. But suppose undo is now pressed $n-1$ more times, so the machine is in state 'on'. What should undo do here? The simplest design options are that the machine switches 'off' (and maybe as a consequence the patient dies) or the machine enters state 'e' because of the ring of states and that *the last time* state 'on' was reached it had been reached from 'e'. Both are

plausible design choices. Other design options, for undo, may require the user making a choice which may involve several design alternatives.

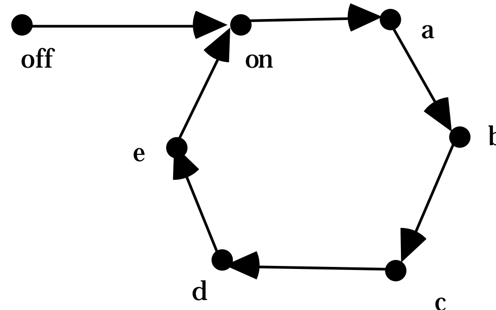

Figure 4. An example system which highlights a general problem for undo.

As the example (Figure 4) shows, what design choice is preferable depends on the use and purpose of the device. Presumably it is more important to keep the patient breathing than to see a particular monitor display; for different scenarios, alternatives, including the machine switching off, may be preferable.

The correct implementation of undo depends on the specific task requirements. It may further depend on specific details of use under specific circumstances. Therefore: there is no general undo strategy for a finite state machine which commonly contain cycles. This observation has implications for the design of systems like HyperDoc and how they support undo in the systems being simulated. We suggest that, with more research, a simulation tool could report on the quality of undo, just as HyperDoc already reports other metrics such as connectivity.

Since HyperDoc neither reads the mind of its users, nor is fixed in its assumptions of the task domain, when the user asks to undo from a state that has been entered from alternative states at different times in the past, the simulator says so and provides the user with options to choose amongst those states (or to cancel the request or to ask for other assistance). This general strategy may not be appropriate for some applications.

4. Natural language

It is crucial to the success of an assistant that its advice is intelligible and even courteous. This paper only discusses *what* the assistant should say, rather than *how* it should express its answers. We note, however, that a correct manual or correct help requires the phrases used to have appropriate syntax and semantics for the states they annotate; our system, as it is currently implemented, does not attempt to enforce this.

The same facts require several modes of expression depending on the linguistic and system contexts in which they are used. For example, different tenses are required to describe the current, past and future intended states. Proper discussion of these topics are beyond the scope of this paper, but for our purposes, in developing HyperDoc, it was sufficient to have just three forms: indicative (you are in this state), imperative (get in this state) and interrogative (is the system in this state?). These forms provide a simple yet very powerful representation of explanation material. The system keeps three tenses for the action of pressing a button

(e.g., “press,” “pressed,” “pressing”) since different systems may need different verbs (switch, hit, slide, click…) The annotations may be as brief or as long as desired. Likewise, transition conditions require three words, usually “assure” (as well as “assured” and “assuring”). Obviously these ideas could easily be extended to deal with whatever style of description was desirable.

The annotations are inserted into templates, which are chosen appropriately to answer each question. In HyperDoc, templates are selected depending on the number of steps involved and other simple grammatical criteria. A trivial string generator caters for is/are, this/these, twice/3 times, and other simple lexical issues (like the placement of commas and full stops) to improve the quality of the English presentation of explanation. The advantage of this approach is that there is very little natural language knowledge in the tool, and it is easy to convert the assistant’s working language from, say, English to German or French.

Rather than speaking or typing a question such as “How do I make the machine record on a tape?” our current system has several menus, as described earlier. One menu is for asking “How?” questions, and if selected currently provides a list of all of the “How?” questions, which will include, for instance, “Record on a tape.” In other words, HyperDoc can answer preset questions without having to invoke natural language parsing. Alternative presentation strategies need to be investigated for complex systems which maintain a large number of states.

Thus, we do not need to handle questions that have no answers! “How can I fry eggs?” is a sensible “How?” question but not one that a video recorder might know how to answer; by providing a menu of options the user is forced into phrasing a plausible question for the particular system.

5. Static analysis of an assistant’s help

Although the assistant is envisaged as an interactive aid, its answers can be analysed statically. For example, for one video recorder we have simulated with HyperDoc, the answer to “Why is the record light on?” is not always that the Record button was pressed (the Play button makes it record if it was previously paused recording)! Or, for an ATM, the answer to “Why not?” is sometimes that the user has not yet pressed Confirm — but not always.

We gave an example above (in section 3.2, which discussed “Why not?” questions) where we would want to modify the system design if the assistant could ever give error states as part of its suggestions. We have found it convenient to do such analyses in *Mathematica* (Wolfram, 1991), which is a powerful general purpose symbolic mathematics system; it can analyse finite state machines and draw their transition diagrams. *Mathematica* is the textual representation for HyperDoc and readily supports examining system designs.

Since HyperDoc saves system specifications directly as *Mathematica* expressions,[*] it is easy to redesign a system within *Mathematica* itself, based on its analyses or other criteria.

The extreme static assistant is the user manual, a static collection of instructions. This may be generated automatically in various forms. A simple manual might be the tabulation of answers for "how to" questions from each state to each other state. Generally, however, manuals require more structure if they are not to become overwhelming to the user, and static analysis can help the designer find a more appropriate structure — or even suggest modifications to the system so that its manual can be shortened.

A companion paper will discuss the analytic support that HyperDoc provides, taking advantage of *Mathematica* and World Wide Web hypermedia manuals. See the brief paper (Addison & Thimbleby, 1995). These 'non-interactive' features are beyond the scope of the present paper however.

6. Training versus assistance

The term 'help' is ambiguous. Does it mean *educational* help, that is, "learn this procedure"; does it mean *advice*, that is, "do this now"; or does it mean *active* assistance, as in "Shall I do … for you now?" In the first case, help might provide documentary material the user can read and learn; in the other cases the assistant can easily achieve the requested objective but should normally ask the user to confirm their intentions (they may have asked the wrong question, for instance!) All these cases may be combined, of course, as in Figure 5, where the "Do It" button enters the additional dialogue with the user if they want more active advice.

---

[*] HyperDoc also stores pictures (of the system, its buttons, indicators etc.) in the *Mathematica* file. Since HyperDoc is a Macintosh application, this feature is supported by the Macintosh's file resources. It allows a full system specification to be saved along with pictures, but yet is a standard *Mathematica* file.

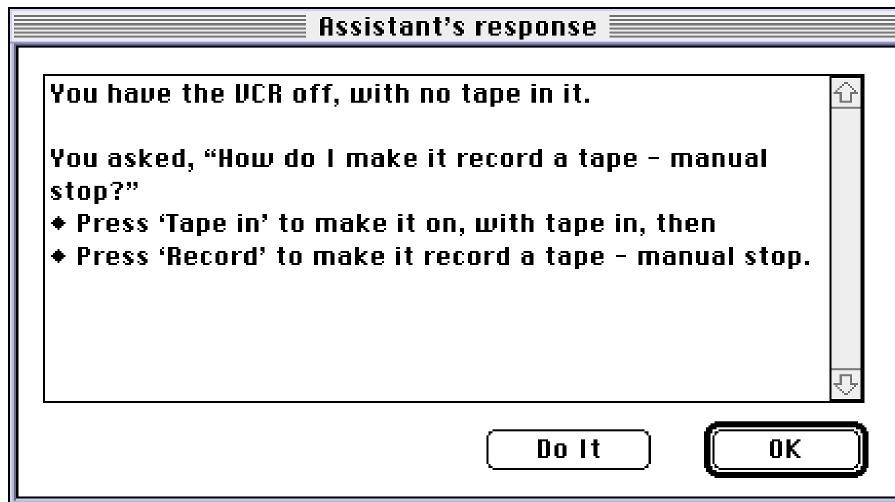

Figure 5. Assistant's answer to a question how to make a VCR record a tape (with manual stop).

If the user presses "Do It" HyperDoc helpfully animates and explains the sequence of steps to get to the state. But if an assistant can take a user directly to a requested state, why not do that directly and dispense with the explanation? There are many answers to this question, including:

- For many devices, it is important to enter states satisfying certain conditions and in a restricted sequential order. Thus, you cannot record a video tape without first inserting a tape.

- Distinctions can be drawn between normal use, training the user, error diagnosis, and error recovery. For normal use, it is likely that pressing buttons is most efficient, though for unfamiliar circumstances, help may be desirable.

- An interactive system typically has far more states than it has buttons. Selecting a target state from the assistant may therefore be more tedious than progressing through a series of familiar states that are selected in a sensible way via buttons.

- To obtain assistance it may be easiest to describe target states by referring to indicators (e.g., that the record indicator is desired to be on).

In summary, there is a continuous trade-off between intrusive assistance and skilled use. This trade-off can be explored very easily; a flexible development tool like HyperDoc enables the designer to modify the behaviour of buttons to perform complete tasks identified through applying the help assistant.

7. Further work

Our approach provides a flexible, reliable and very adequate form of help, which can be analysed quantitatively. In comparison it is surprising how

inconsistent and unreliable other systems' help systems are — with the user sometimes not comprehending what the help is suggesting.

For example the Apple Newton has an assist feature that can sometimes *do* operations, but will sometimes only tell the user about *how* to do them. Moreover, in telling the user, it often uses obscure terminology (e.g., "Tap the appointment marker," and the user may not know which of various symbols is that marker) when it could perform the operations itself. Some things it could give help on are not provided. There are innumerable inconsistencies — *all* of which could have been avoided by constructing the assistant automatically, as we suggest in this paper.

We see, then, that an important area of further work would be to devise ways of specifying (or implementing) complex systems so that they can be annotated as we have done for finite state machines, and have the benefit of their manuals and help being automatically derived. Current programming languages, even ones that are object oriented, have no objects that correspond closely enough to our states for this to be easily done.

7.1. Adaptive help

Previously we have shown that answers can be provided for serious questions by finding least cost paths. If the costs of transitions are all equal, then the answers will give the actions that require the least number of button presses. But there are many other possibilities. For example, on a cash machine (ATM) the costs could be expected financial costs, and the answers would then provide the cheapest options for the user (or perhaps the bank.)

Clearly weights can be varied so that answers better suit different requirements; in particular they can be dynamically adapted. There are two forms of adaptive help, whether the weights are adjusted based on the user's *current* behaviour or on *previous* training. In the first case, adaptive help can easily provide answers that use familiar routines that the user frequently uses; in the second case, the help can be trained on certain styles of use, such as emergency response or expert use. Suppose the question is "how do I switch off and eject the tape?" — one answer, based on weights from the current user would be based on how this user (or this machine) generally reaches the intended state. The other answer could be based on how an expert might reach that state.

7.2. Designing new buttons

HyperDoc could enable the meaning of a button to be set to the answer of an assistant question. Thus, if the question is "how do I make the record indicator stay on and the pause light go off?" a new button's transitions could be initialised to achieve this condition from all states. The designer specifies a set of invariants (e.g., "record indicator on") and a set of termination conditions (e.g., "pause light off"). The button would do nothing in those states not satisfying the invariant. Of course, the designer can subsequently edit a button's behaviour. Again we note the pleasing symmetry between the needs of the user and the needs of the designer.

7.3. *Tracking manual writing*

Keeping track of writing the manual entries is a task that HyperDoc could do even better. A conceptual indicator could be introduced to mean a state's annotation has not been written. We could then define a goal set to include all states where this indicator is set.

To write a complete manual, then, all the designer need do is write any manual entry, and then ask the assistant how to get to a state that does not have a manual entry, write its manual entry, and so on. Following this process guarantees every state is annotated.

7.4. *Tracking user testing*

Similar methods to tracking manual writing could be used to help in user testing to ensure every state is tested. Additional types of annotation may be permitted so that test users can record their comments about states (e.g., that their indicator lights are incorrectly set, or whatever). These annotations can be gathered in exactly the same way that a manual would be created, but to summarise the user's criticisms.

The existing feature of weighting edges to provide better advice could also be summarised so that evaluators know what routes users have been taking. (At present, the weights matrix is only available as a Mathematica expression — but one can do *anything* with it.)

7.5. *Other work in HyperDoc itself*

There are many approaches to calculating path costs between two states to answer questions, such as the "How to?" Answers can be optimised based upon traversing routes of states most often entered — in this case, providing help based upon familiarity to the user (i.e., "you've done this sort of thing often before"); or help can be optimised based upon traversing routes of states never entered — in this case, providing a style of user instruction (i.e., "you'll learn something new going this way"). The current version of HyperDoc weights each action with a number that can be edited by hand, but which is not automatically adjusted.

Since it is generally undesirable that the assistant ever provides answers suggesting that the user visit error states, it is a simple matter to run the assistant over all pairs of states and flag — to the designer — any anomalous answers. Any such answers indicate that the system design and/or the weights need modifying.

8. Conclusions

Humans perform many complex tasks that require interaction with interactive systems. Many systems provide a vast range of facilities and the choices open to the user may be unknown or confusing. To apply complex technology effectively often requires some form of assistance to support the user's needs. Yet providing assistance is itself a complex task and one that requires automation to be done reliably.

Our approach opens up a whole range of possibilities for user and designer assistance for various systems, and the extent of assistance provided is powerful, flexible and useful. This paper showed how the user can be helped in many ways very easily, and several ways how help may be generated automatically and adapted, if required, to fit users' or experts' requirements or adapt to their behaviour. Our approach opens up

possibilities, such as training a system to generate different styles of help provision — for emergencies, routine maintenance and so forth. However, the main advantage of our approach remains its simplicity and reliability. We raised a number of issues concerning the provision and development of interactive systems; and we related these ideas to HyperDoc, our exemplar which illustrates the approach. Tools like HyperDoc are sufficiently general to implement many interactive systems. We would argue, further, that they are reliable, powerful yet so simple that their implementation cost can be recovered immediately from the quality and speed gains in the overall design process.

Our work shows that automatically constructed assistance is much easier for the designer, and much more powerful for the user — and it can be done consistently and reliably, with guaranteed correctness and completeness. We showed that all these features can be available simultaneously — the designer, technical writer, and user can all use the *same* tool — and this has huge advantages for clarifying and solving design issues.

Acknowledgements

This work was supported by SERC Grant No. GR/J43110, "Systems, manuals, usability and graph theory." We would also like to thank the anonymous referees and the journal's general editor for their helpful comments.

References

Addison, M. and Thimbleby, H. (1994) "Manuals as Structured Programs," in G. Cockton, S. W. Draper and G. R. S. Weir (editors), BCS Conference HCI'94, *People and Computers*, **IX**, pp. 67–79, Cambridge University Press.

Addison, M. and Thimbleby, H. (1995) "Hypermedia manuals for interactive systems," *The Authoring and Application of Hypermedia-based User-Interfaces*, IEE Digest No. 95/202, pp. 5/1–5/4.

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.

Coulouris, G. F. & Thimbleby, H. W. (1992) *HyperProgramming*, Addison-Wesley.

Abowd, G. D. & Dix, A. J. (1992) "Giving Undo Attention," *Interacting with Computers*, **4**(3), pp. 317–342

Halbwachs, N. (1993) *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers.

Harel, D. (1987) "Statecharts: A visual formalism for complex systems" *Science of Computer Programming*, **8**, pp. 231–274.

Harel, D. (1988) "On visual formalisms" *Communications of the ACM*, **31**(5), pp. 514–530.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politit, M., Sherman, R., Shtull-Trauring, A. & Trakhtenbrot, M. (1990) "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, **16**(4), pp. 403–414.

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

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

Thimbleby, H. W. (1993) "Combining Systems and Manuals," in Alty, J. L., Diaper, D. & Guest, S. (editors), BCS Conference HCI'93, *People and Computers*, **VIII**, pp. 479–488, Cambridge University Press.

Thimbleby, H. & Addison, M. (1995) "HyperDoc: An Interactive Systems Tool," in Kirby, M. A. R., Dix, A. J. & Finlay, J. E. (editors), BCS Conference HCI'95, *People and Computers*, **X**, pp. 95–106.

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