Chapter XXX Specifying history and backtracking mechanisms

Alan Dix and Roberta Mancini

Abstract

Most hypertext systems and in particular most web browsers include some form of history mechanism. These allow users to see where they have been and to return to previous locations. The form of such mechanisms varies: some are stack based, others keep a full trace of all previously visited locations. Users are often confused by these mechanisms and this confusion is also reflected in some specifications! One reason for this confusion is that history mechanisms are reflexive: you are forced to think in terms of what you have been doing, rather than simply doing it. Undo mechanisms share this reflexivity as well as being similar in other ways. One consequence of reflexivity is that the notion of 'state' can be difficult to define, there being a simple state (the system without history/undo) and a full state (including the history information). This chapter will examine the history mechanisms of several browsers and hypertext systems and show how these can be formally specified over a generic base model. This will also highlight the fact that some such systems have more than one history mechanism simultaneously at work – no wonder the users get confused! The chapter draws on insights gained during recent extensive studies of undo mechanisms by the authors.

URL for related work: `http://www.soc.staffs/ac/uk/~cmtajd/topics/undo/`

1. Introduction

Whereas most of the chapters in this book are specifying the same Netscape-like WWW browser from different perspectives, this chapter takes a single element of a browser, the history mechanism, and compares the formal specification of this aspect of four different hypertext browsers: Netscape Navigator, HyperCard, Microsoft Windows Help system and Think Reference.

The World Wide Web is (amongst other things) a giant hypertext and, as with any hypertext system, users are likely to follow wrong or uninteresting links, to want to return to previously visited nodes and if they do not have superhuman memory and accuracy are all likely to get completely 'lost in hyperspace' (Conklin 1987). History mechanisms are one of the ways users regain control back in such circumstances, allowing them either to get an overview of where they have been, or simply to go backwards step by step through previous nodes.

Finding useful information in large hypertext systems such as the WWW is difficult and time consuming. If there were no history mechanism users would have to leave bookmarks or write down the URL of any interesting web page before navigating further, for fear that they might not be able to get back without having to repeat the whole complex search process. Having a history mechanism or even simply a 'back' button means that users can navigate links knowing that if the place they get to is uninteresting they can return. This enables users to engage in a variety of deep and exploratory search strategies and to perform a simple undo of erroneous or unfortunate navigation. Indeed, studies of web browser use have shown that 30–40% of all movements between pages are made using the 'back' button (Catledge and Pitkow 1995; Tauscher and Greenberg 1997).

Tauscher and Greenberg's studies have also shown that a large proportion of revisited pages have been recently visited and hence are likely to be in a history list. Thus a well-designed history mechanism can significantly reduce the effort of finding these pages. This can be seen in terms of Information Foraging Theory (Pirolli and Card 1995), which suggests that the most efficient interface is one that increases the rate of (useful) information gain per unit time. If recent pages are valuable, they need to be accessible with the minimum of physical (keystrokes) and mental (cognitive load) effort. If the page is in the history list it will certainly reduce the physical effort. However, it is clear that cognitive effort will not been similarly reduced, as many users find history mechanisms quite confusing (Cockburn and Jones 1996; Tauscher and Greenberg 1997).

This difficulty even extends to attempts to formalise aspects of hypertext systems! One reason for the difficulty is that history mechanisms are reflexive: they involve the user and the designer in looking at the process of interaction. Whilst surfing the web, users are deeply *engaged* in the activity; they are not thinking *about* what they are doing, they are simply *doing* it. However, when they use the 'back' button or invoke a history mechanism they must make this shift from acting to thinking about their actions. That is, the very use of any history mechanism is a breakdown in interaction (Dix et al. 1997).

The reflexive nature of history mechanisms makes specification difficult too. During normal interaction the system can be regarded as being in some state (the current node or page) which the user's actions (navigating links) modify. However, the history mechanism needs to talk about the past states of the system. This means

that the system must keep a record of where it has been – previous states. Of course, this record of previous states must be part of the current state of the system!
This reflexiveness is also evident in undo mechanisms for interactive applications such as word processors. In such applications, the system also has to keep some track of previous user actions in order to undo their effect (and possibly redo also). In fact, when formally modelling such systems history and undo mechanisms are virtually identical. The authors have been studying formalisation of undo and the same general formal techniques apply directly to hypertext history.
There is no single history model applied in all web browsers and hypertext systems. Indeed, some browsers embody two simultaneous different methods. In this chapter we will examine and specify several history mechanisms from four different hypertext systems. In all, we will see six different history mechanisms at work. This accords with previous observations of the wide variety of history mechanisms found in hypertext systems (Bieber et al. 1997; Tauscher and Greenberg 1997). In two cases our history mechanisms correspond almost exactly with undo mechanisms found in other applications. Perhaps the most surprising thing is not the variety of mechanisms, but the fact that some browsers embody two simultaneous different methods. Two of the four browsers studied had two different history lists, one for their 'back' command and one for their menu-based history. However, even sharing a common history list does not mean that 'back' and menu selection will have consistent effects on the list!
We will begin the next section by discussing some of the general issues for specifying history. In section 3 we will model each of the mechanisms using the same general framework. Because of the similarity of presentation we are able to compare (in section 4) the mechanisms with each other, ignoring other aspects of the different underlying hypertext systems. We are also able to compare them with similar undo mechanisms.

2. Reflexive specification

We have already noted that history mechanisms are by their nature reflexive. The history mechanism needs to keep track of previous states, but the data structures used to keep this trace of previous states must themselves be part of the state. Happily this apparent infinite regress is not quite as bad as it seems. We can consider the system as having two levels of state. First, the state of the system if it had no history mechanisms: the current web page or node being visited plus any additional information about the viewing of the node such as the portion currently being displayed for a large scrolled page. Second is the state of the system augmented with history information. Components of this second bigger state will correspond to instances of the original smaller state.
We will model both kinds of state using a variant of the PIE model (Dix and Runciman 1985; Dix 1991) originally developed as a general model of interactive systems.

2.1 Browser commands and state

So, first we ignore the state and commands associated with history mechanisms. That is, we imagine a browser with no 'back' button, nor any history list. The set of states of the system we will denote S. In the case of web browsing this state will include the URL of the current page, the position it is currently scrolled to, the contents of any fields if it is a form and even perhaps the state of any JAVA applets executing in the current page. User actions modify this state. The set of user actions we denote C (commands) and their effect is modelled by a state update function *doit*:

$$doit : S \times C \rightarrow S$$

Within any session browsing will start from some initial state (e.g. the user's home page); we write s_0 for this. Also, it is useful to consider the effect of whole sequences of user commands. We call this 'history' of commands H (H=C*).[1] If necessary we will use the obvious extension of *doit* to whole histories of commands and also define the interpretation function I as follows:

$$I : H \rightarrow S$$

$$I(h) = doit(s_0, h)$$

[1] The word 'history' is used in several linked, but different, senses. At this point we are looking at the command history, whereas up to now we have simply talked about 'history' mechanisms as those which allow the user some access to previously visited states or nodes; later we will formally model this 'history' using sequences of states.

That is, the interpretation gives the state reached if the history h is the complete trace of user commands since the browsing session began.

2.2 Components of the browser state and command classes

As noted above, the state will typically consist of two things: the name or location of the node being visited (say from a set N) and some viewing information (V). That is:

$$S = N \times V$$

Given such a state s we will refer to the two components as $s.loc$ and $s.view$.
This then is the state of the browser. In addition, there is the current state of the hypertext being browsed; that is, the content associated with any node:

$$Htxt = N \rightarrow Content$$

This mapping between location and content may be:

- fixed – as in the case of CD-ROM-based hypermedia and also Windows Help
- modified by the user – as is the case with editable HyperCard stacks
- modified or even constructed dynamically by other users or processes – as is the case with WWW pages, especially those generated by CGI scripts

However, for the purpose of this chapter we will concentrate on the state of the browser and effectively assume the hypertext itself is fixed.
Returning to the browser, we can use the two components of the state to distinguish two classes of commands:

- active – those which may change the location of the browser:
  $\exists s \in S$ **such that** $doit(s, c).loc \neq s.loc$
  For example, clicking on a link.
- passive – those which only ever change the view:
  $\exists s \in S$ **such that** $doit(s, c).loc = s.loc$
  For example, scrolling the text.

History mechanisms typically only record *locations* visited, not every intermediate state. So, for example if there is a 'back' button and we used it immediately after a passive command we would expect to return to the node before the last active command. There are two further complications. First, some browsers (including Netscape Navigator and most web browsers) do keep track of some view information for previously visited nodes. Other hypertext browsers simply return you to a standard state for the node (e.g. scrolled to the top). Second, the behaviour when an active command actually stays on the same node may cause confusion. For example, what should happen when a normally active command doesn't cause a change to a different node, for example if an HTML anchor points to itself? In the case of both Netscape and HyperCard this does not add an entry to the history used for their respective 'back' functions. That is, they rely on the instance of the use of the command being active. However, note that in the WWW different URLs (locations) may refer to parts of the same document (using labels such as '`mypage.html#top`'). These *are* regarded as different locations even though they are the same physical page!
Although this behaviour is interesting, and a major source of confusion for the user, we will ignore it for the rest of this chapter. Instead, we will simply assume that the state consists solely of the location of the current node or page and that all user commands are active. This is to prevent our formulations of different mechanisms becoming too bogged down with side conditions of the form 'if the command c is active'.

2.3 History commands and state

The commands in C are simply those for navigating within and between pages. If we include an explicit history mechanism then the possible commands will be augmented by those for accessing this history: the 'back' button, history list or 'GO' menu. This augmented set of user commands we will call C^a and the corresponding traces of commands H^a.
Similarly the state will be augmented by additional components. We will call this augmented state S^a. As previously noted the components of this will typically contain instances of the 'original' state set S. For example, a simple one-step history mechanism where the 'back' button allowed one to view the previous state only might keep two copies of the state: $S^a = S \times S$. In fact all the examples we will look at keep much more extensive history mechanisms and store some sequence of states.

Looking at this augmented system, we see that it too has a state update function $doit^a$, initial state s_0^a , and derived interpretation function I^a:

$$doit^a : S^a \times C^a \rightarrow S^a$$
$$I^a : H^a \rightarrow S^a$$

$$I^a(h) = doit^a(s_0^a , h)$$

The exact form of the augmented commands, state and behaviour depends on the particular history mechanism and we will refer to these by superscripts: e.g. C^{net}, $doit^{net}$, etc. for Netscape.
Although the augmented system has additional state to manage history, the user is principally aware of the current page or node being visited. This can be represented by a mapping *proj* giving for any state of the full system the current node and view of that node:

$$proj : S^a \rightarrow \mathrm{S}$$

2.4 Model and implementation

In section 3 we will give explicit models of the augmented state of different browsers. For example, for Netscape we have:

$$S^{net} = Nat \times S^+$$

where *Nat* is the set of natural numbers {0,1,2,…} and S^+ is the set of *non-empty* sequences of states. Note that this does not mean that Netscape Navigator *implements* its history mechanism in this way, just that this is a faithful model of its *behaviour.* This is even more important in modelling undo mechanisms where it would be ridiculously inefficient to store copies of the state and instead a record of *changes* is kept.
Another important difference between actual implementations and the abstract specifications given here is that in the real browser a large proportion of the state consists of information about the context of the browser and preferences of the user. The state we are referring to is only that directly associated with the nodes being visited. History is *not* undo – we do not expect the 'back' button to reverse the effect of resizing a window, so this information would be stored once, separate from the history of browsing states. Also a lot of development effort and storage in web browsers is dedicated to efficiency, especially the caching of recently visited pages. Again this is ignored for the purposes of this chapter.

2.5 Conservativeness of state

If the user never used any history commands the augmented system should behave exactly like the original system:

$$\forall h \in H : proj(I^a(h)) = I(h)$$

In fact, we expect it to behave like the original system even between uses of history commands. That is, if the user issues a normal navigation command, we expect the effect on the augmented system to exactly parallel that of the original browser without undo.

$$proj(s_0^a) = s_0$$
$$\forall c \in C, s \in S^a : proj(doit^a(s, c)) = doit(proj(s), c)$$

We call this condition *conservativeness of state*. It is equivalent to the commutativity of the diagram in Figure 1. It is very important as it means that the user does not need to be aware of the history mechanism except when it is being explicitly invoked.

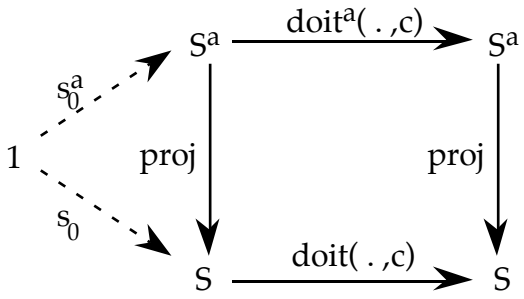

Figure 1. Conservativeness of state.

This is a fundamental property of any system enhancement which purports to extend the functionality of the original system. In particular, it is essential for any sensible history or undo mechanism. Happily, all the history mechanisms we examine in the next section satisfy this condition.
Conservativeness does not capture specific properties of the history mechanism itself, but instead simply states that the original system is preserved within the augmented system. In section 4, we will return to this issue.

3 Six history mechanisms

In this section, we will look at the history mechanisms from four different hypertext browsers. In the case of two of these browsers, Windows Help and HyperCard, each has two simultaneous mechanisms at work giving rise to six history mechanisms in total. The fascinating thing is that they are *all* different. No wonder users get confused!
The systems we examine are:

- Microsoft Windows Help – used by most Windows applications to display hypertext help. This has two history mechanisms: a button labelled ‘back’ and a ‘history’ menu.
- Think Reference – a hypertext help/reference manual used by many Macintosh programmers. This has a single mechanism accessed by either selecting from a pull-down ‘go back’ menu or simply clicking the ‘go back’ button which acts as the menu’s pull-down tab.
- Netscape Navigator – the WWW browser. This has a single mechanism, which can be accessed several ways: ‘back’ and ‘forward’ buttons, a ‘go’ menu and a ‘history’ window.
- HyperCard – the Macintosh hypertext/interface tool. This has two mechanisms: a ‘back’ button and a ‘go recent’ window.

3.1 Windows Help – a stack and a trace

As noted Windows Help has two mechanisms: a button labelled ‘back’ and a ‘history’ menu.

3.1.1 Windows ‘back’ – a stack

The ‘back’ button is simply stack based: each press of the button takes you one step back. The augmented state S^{wBk} can thus be modelled as a non-empty sequence of states (from S) – the stack of visited nodes, the last of which is the current state:

$$S^{wBk} = S^{+}$$

$$proj^{wBk}(bs) = last(bs)$$

When an ordinary navigation command (from C) is issued, the effect is to use the normal navigation *doit* function on the current state (location) to get the new current state and to add it to the history. The back command simply pops an element from the stack of past states if it is non-empty, otherwise it does nothing:

$$doit^{wBk}(bs, c) = bs^{\frown} doit(last(bs), c) \qquad c \in C$$
$$doit^{wBk}(< s >, back) = < s >$$
$$doit^{wBk}(bs, back) = chop(bs) \qquad len(bs) > 1$$

N.B. these equations use *len* which gives the length of a sequence, *last* which gives the last element of a sequence, and *chop* which removes the last element from a sequence. The operation $x^\frown y$ is the concatenation of y to the sequence x. The same notation will be used for the concatenation of two sequences.

3.1.2 Windows 'history' – a trace

The history mechanism keeps a trace of all states visited and adds the new state to the sequence every time a new node is visited. The state is thus similar to that for the back command:

$$S^{wHy} = S^+$$

$$proj^{wHy}(hs) = last(hs)$$

This is navigated by using a menu which shows all nodes in the history list and allows the user to choose one. We will call the command which chooses the nth element from the menu $choose(n)$. We can assume that n only takes on sensible values as it is impossible to select items not on the menu!
If we look at the behaviour on the history list, ordinary navigation commands simply add to the history list, whereas using $choose(n)$ revisits the relevant node:

$doit^{wHy}(hs, c)$	=	$hs^\frown doit(last(hs), c)$	$c \in C$
$doit^{wHy}(hs, choose(n))$	=	hs	$n = len(hs)$
	=	$hs^\frown hs[n]$	otherwise

(N.B. $1 \leq n < len(hs)$)

Compare this with the *back* mechanism: when using 'back' the last state is removed from the back-stack, whereas when choosing from the history list the new state is added to the history. Also note carefully the conditions on the behaviour of $choose(n)$. If the current item from the history list is chosen nothing happens. However, the condition is $n = len(hs)$ not $hs(n) = last(hs)$. That is, although selecting the top element does not cause a repetition in the history list it is possible to get repeats by selecting another entry further down in the history list which happens to be the same as the current state. For example, suppose you have navigated to nodes a, b, c and then back to b. The history menu would read *abcb* (actually the menu is vertical with a at the bottom). If you then selected the fourth entry nothing would happen, but if instead you selected the second entry (also b) the history list would then read *abcbb*!
Similar things can happen with the back-stack although this is not immediately visible. However, if you perform actions which visit the same node several times in a row, these are all recorded in the back-stack. Subsequent use of 'back' leads to stuttering at that node. This is very different from the behaviour of equivalent 'back' commands which we will encounter in Netscape and HyperCard (and to some extent Think Reference) which avoid repeats based on the semantics of actions (whether the same node is revisited) rather than the syntax (the last item from the history menu).

3.1.3 Combining 'back' and history

These two mechanisms do not share a single history, but in fact both are maintained. That is, the full state is really of the form:

$$S^{win} = S^+ \times S^+$$

Both the back-stack and the history should have the same current state:

$$\forall\ [bs,hs] \in S^{win} : last(bs) = last(hs)$$

The projection can then be constructed from either of these:

$$proj^{win}([bs,hs]) = last(bs) = last(hs)$$

The first component in the state is the back-stack and the second the history trace. We have already specified the effects of ordinary commands on all components of the state, but need to also specify the effects of the *back* command on the history trace and of the $choose(n)$ command on the back-list:

$$doit^{win}([bs,hs], c) = [bs^\frown doit(last(bs), c), hs^\frown doit(last(hs), c)] \quad c \in C$$

$$\begin{array}{lll} doit^{win}([< s >, hs], back) = [< s >, hs] & & \\ doit^{win}([bs^\frown s, hs], back) = [bs, hs^\frown last(bs)] & & len(bs) \geq 1 \\ & & \\ doit^{win}([bs, hs], choose(n)) = [bs,hs] & & n = len(hs) \\ doit^{win}([bs,hs], choose(n)) = [bs^\frown hs[n], hs^\frown hs[n]] & & \text{otherwise} \\ \quad (1 \leq n < len(hs)) & & \end{array}$$

It is unlikely that users will be aware of this complex interweaving between the two history mechanisms. At least if they get confused they can always use the history list which has the trace of everything.

3.2 Think Reference – a recency ordered set

We look next at Think Reference. This is a programmer's on-line reference system supplied with the Symantec C++ compiler for the Macintosh. It also has a simple one-click mechanism which can be accessed by clicking a 'go back' button on its control panel (Fig. 2).

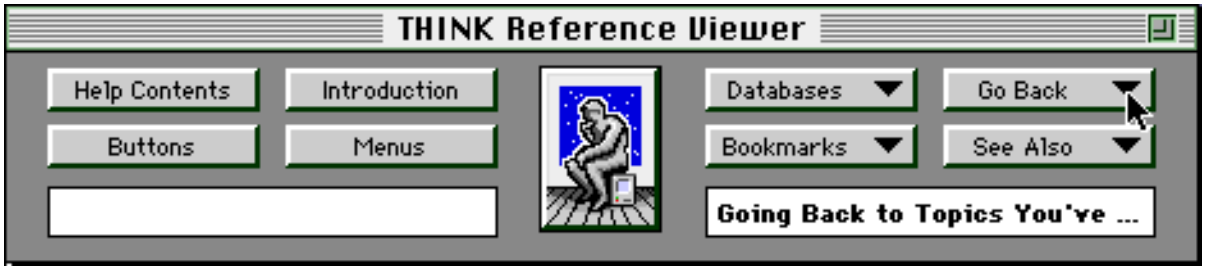

Figure 2 Think Reference control panel

If only this method of access is used and *so long as no node is visited twice* it behaves exactly like the 'back' button in Windows Help.

In addition, if the 'go back' button is held down a menu appears showing the *previously* visited node (not including the current node). This functions in a similar fashion to the Windows history list: when the user navigates to a node *which has not been previously visited* the previous node is simply added to the end of the current history list. However, if a previously visited node is revisited or if the user selects a node from the 'go back' menu, the behaviour is different – the previous instance of the current node is removed from the history list. That is, if you visited nodes in the order *a*, *b*, *c* then again *b* and finally *d*, the history list would simply read *acb* rather than *abcb*.

In the case of Think Reference the two mechanisms, 'go back' menu and 'go back' button, share a common history list rather than having two. However, the 'go back' button is *not* simply a shorthand for selecting the last element in the list. Whereas simply clicking the 'go back' button always reduces the length of the history list (until it is empty), selecting the last element of the menu leaves it the same length. Although the top element is removed (to become the new current node) it is replaced with the previous current node, effectively toggling the current node and the previous node.

This behaviour is clear if we look at the history model of Think Reference:

$$S^{thk} = S \times S^*$$

Note that in this case the second component of the state is S^*, a possibly empty sequence of states. This is to reflect the fact that the Think Reference 'back' menu does not include the current state

$$proj^{thk}([s, hs]) = s$$

$$\begin{array}{lll} doit^{thk}([s, hs], c) = update(s, doit(s, c), hs) & & c \in C \\ doit^{thk}([s, <>], back) = [\ s, <>] & & \\ doit^{thk}([s, hs^\frown s_n], back) = [\ s_n\ hs\ ] & & \\ doit^{thk}([s, hs], choose(n)) = update(s, hs[n], hs) & & (1 \leq n \leq len(hs)) \end{array}$$

Again we assume that the menu selection *choose*(*n*) can only select actual elements of the history list.
The effect on the history list has been defined using a subsidiary function *update*. This emphasises the fact that the effect on the history is identical whether a new state is visited due to navigation or the use of the 'go back' menu. It also emphasises that using 'go back' as a button, *back*, is different from selecting the top of the 'go back' menu, *choose*(*len*(*hs*)).

$$update: S \times S \times S^* \rightarrow S^{thk}$$

$$update(old, new, hs) = [new, hs']$$
$$\text{where} \quad hs' = hs[1 \ldots m-1]^\frown hs[m+1 \ldots len(hs)]^\frown old \quad \text{if } \exists m \textbf{ such that } h[m] = new$$
$$= prune^{thk}(hs^\frown old) \quad \text{otherwise}$$

We can see now that in the model *back* is the only forgetful command; both navigation, $c \in C$, and 'go back' menu choice, *choose*(*n*), simply add to or re-order the history list, the 'go back' button truncates it. (In fact, there is some forgetfulness in the others too as the history menu loses some of the oldest nodes if it gets too long. This behaviour is modelled by the $prune^{thk}$ function.)
To see the effect the combined history has on *back* consider the following scenario. Imagine you have followed links in the order *a*, *b*, *c* then *b* as previously. The node *b* is current and you press 'go back'. The current node becomes *c*. You press 'go back' again and rather than *b* (as you would get with Windows Help) you get to *a*.
The history list is more 'economical' than Windows Help in that it doesn't repeat the same node name. This is especially noticeable if you move rapidly back and forth from a table of contents which would otherwise consume limited menu slots. The apparently good step of making the two mechanisms share a common history list means, however, that simple 'go back' has some occasional strange behaviour.

3.3 Netscape – a stack with pointer

We now come to Netscape Navigator and WWW browsing. Netscape has a 'go' menu which behaves somewhat similarly to the Windows Help history list in that it is a trace of previously visited nodes. Duplicates are allowed and may even arise from labelled links within a single web page which are regarded as separate entities. The current node is always in this list, but is not necessarily the last element in the list and Netscape puts a tick against the current node. The state history recorded in this menu is also used for the 'back' and 'forward' buttons which move the tick backwards and forwards through the list.
Thus the state consists of a non-empty sequence of nodes and a pointer:

$$S^{net} = Nat \times S^+$$

$$proj^{net}([p, hs]) = hs[p]$$

The operation of the 'back' and 'forward' buttons and menu selection within the 'go' menu all simply manipulate the pointer, leaving the history list itself unchanged:

$$doit^{net}([1, hs], back) = [1, hs]$$
$$doit^{net}([p, hs], back) = [p-1, hs] \quad p > 1$$
$$doit^{net}([p, hs], fwd) = [p, hs] \quad p = len(hs)$$
$$doit^{net}([p, hs], fwd) = [p+1, hs] \quad p < len(hs)$$
$$doit^{net}([p, hs], choose(n)) = [n, hs] \quad (1 \le n \le len(hs))$$

The most complicated case is normal navigation (!), which prunes those states after the current state in the history list.

$$doit^{net}([p, hs], c) = [p+1, hs[1 \ldots p]^\frown doit(hs[p], c) \quad c \in C$$

So, Netscape manages to unify a simple 'back' button with menu-based history. In fact, regarding 'back' and 'forward' for navigation as parallel to 'undo' and 'redo' for update, the Netscape history is identical to undo in Microsoft Word 6.0. However, the price it pays is that, like Word 6.0, it is forgetful. One of the advantages of

having a history mechanism is that it allows you to recover from mistakes. You click on a link, decide the page you have got to is not what you want to see and so press the 'back' button.
We have referred to the similar effect of undo as reducing the *risk* of interaction (Dix et al. 1996). However, the fact that navigation may cause some of the history to be forgotten means that every navigation action becomes itself risky. Imagine you have spent some time navigating various web pages and find a really useful site. With foresight you should of course leave a bookmark at once, but you are too busy seeking information … You want to refer back to a previous page you have seen during your search so you use the 'go' menu (or the 'back' button) to go back a few steps to the page. By using 'back' and 'forward' or menu selection you flick back and forth comparing the information on the two pages. Then you notice an intriguing link on the *older* page and without thinking you click it. Instantly you have lost track of all the pages since that point – risk!

3.4 HyperCard – a first use ordered set and a navigable trace

Finally we look at HyperCard, a Macintosh program which is a cross between a hypertext authoring tool and interface prototyping tool. As it is programmable its default behaviour can be modified in various ways, but for the purpose of this chapter we will look at its default behaviour when used as a simple hypertext browser.
Like the other systems it has a 'back' button and a more extensive history in the form of a 'recent cards' window. All the other systems refer to nodes or pages by name. In contrast, the HyperCard 'recent cards' window has a miniature view of each card visited (Fig. 3).

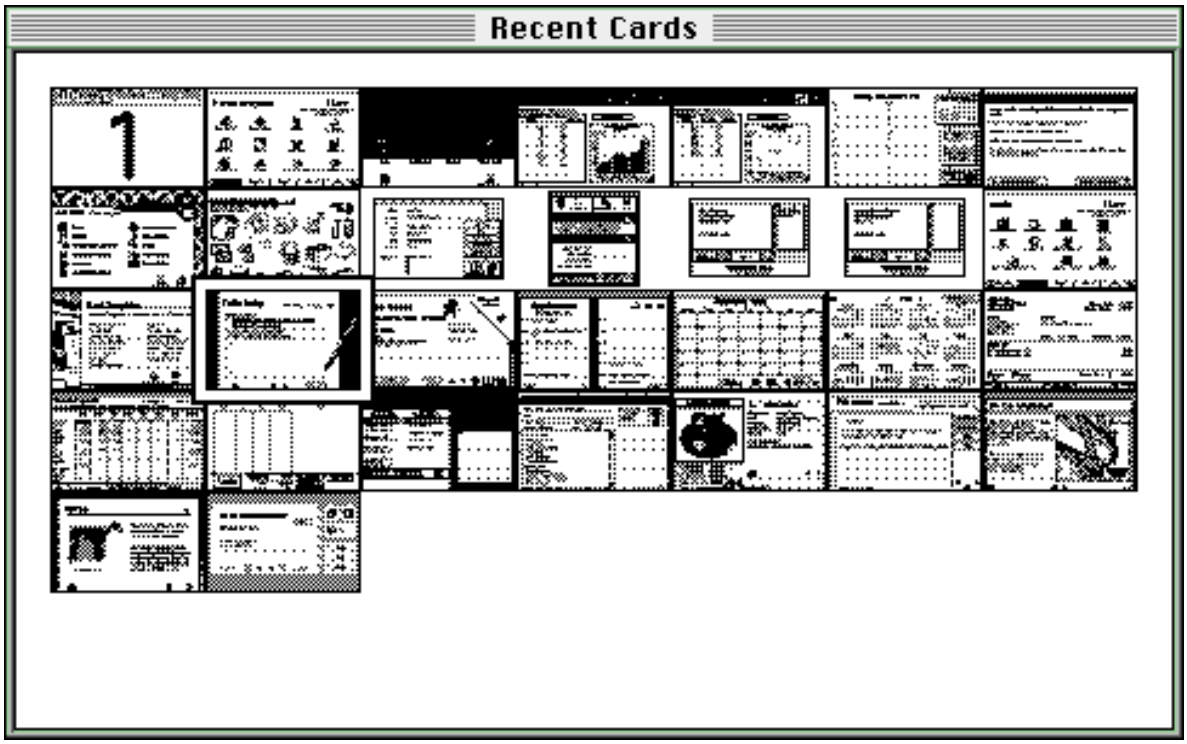

Figure 3 HyperCard 'recent cards' window

As was the case for Windows Help, these two mechanisms 'back' and 'recent' behave virtually independently. Although the 'back' button at first appears to behave similarly to the stack-based mechanisms of the other systems, it is actually significantly more complex. Hence we'll deal with the 'recent cards' window first.

3.4.1 HyperCard 'recent cards' – a first use ordered set

Rather like the Think Reference 'go back' menu, the HyperCard 'recent cards' window has no repeats. However, it achieves this by simply not adding a new node to the window if it is already there. Like Netscape it keeps track of which is the current node, in this case by adding a box around the relevant card image (second card on the third row in Fig. 3).
The state of the system can thus be represented in a similar fashion as pointer and non-empty sequence:

$$S^{HCrt} = Nat \times S^{+}$$

$$proj^{HCrt}([rp, rw]) = rw[rp]$$

The update function for selection from the window is trivial (exactly like Netscape):

$$doit^{HCrt}([rp, rw], choose(n)) = [n, rw]$$

The update when a navigation command is used is slightly more complex. If the card is already in the recent window it is not added, otherwise it is added to the end. However, adding a card to the recent window may mean that there are too many cards to fit in the window in which case the first row of cards is scrolled off the top and disappears for ever.

$$doit^{HCrt}([rp, rw],c) = addRecent(doit(rw[rp], c), rw) \qquad c \in C$$

$$\begin{array}{lll} addRecent(s, hs) = [n, hs] & \exists n \textbf{ such that } hs[n] = s \\ = [len(hs'), hs'] & \text{otherwise} \\ & \text{where } hs' = prune^{HCrt}(hs^\frown s) \end{array}$$

The function $prune^{HCrt}$ represents the scrolling off of excess cards. Note that the current card is always in the 'recent cards' window (either it is already there when there is no pruning, or it is at the end). However, the previous card can easily be pruned. Imagine you visit card *a*, then visit lots of other cards so that the 'recent cards' window becomes full, with *a* on the top line. You then visit *a* again. As *a* is already on it, the window does not change. Then you go to another card *b* which you haven't visited before. This new card *b* is added to the end of the 'recent cards' window and the top line including *a* is scrolled away!

3.4.2 HyperCard 'back' – a navigable trace

Happily you can get to the previous card using the 'back' option in the HyperCard 'Go' menu (there is no corresponding 'forward'). However, this is probably the only simple thing about it. In fact its specification is simpler to understand than its behaviour so we'll start with that. Like the Netscape history it can be thought of as a trace of past states with a pointer:

$$S^{HCbk} = Nat \times S^{+}$$

$$proj^{HCbk}([bp, bs]) = bs[bp]$$

Ordinary navigation commands add the new state to the end of the history and set the pointer to that location:

$$doit^{HCbk}([bp, bs], c) = addBack(doit(bs[bp], c), bs) \qquad c \in C$$

$$\begin{array}{lll} addBack(s, hs) = [len(hs), hs] & last(hs) = s \\ = [len(hs) + 1, hs^\frown s] & \text{otherwise} \end{array}$$

The 'back' command moves the pointer backwards, but it also adds the new current state to the end of the history:

$$doit^{HCbk}([bp, bs], back) = doBack(bp, bs)$$

$$\begin{array}{lll} doBack(0, hs) = [0, hs] \\ doBack(p, hs) = [p - 1, hs^\frown hs[p - 1]] & p > 0 \end{array}$$

The effect of this is to store a backwards history at the end whilst using 'back'. Suppose you visit the cards *a*, *b*, *c* and *d* and then use *back* three times. The card being shown will be *a*. Now you navigate to *e*. If you then use back again repeatedly, you would see the nodes in the order *e*, *a*, *b*, *c*, *d*, *c*, *b*, *a*. This is because the stored history was *abcdcbae*. That is, HyperCard did not just record the end point of your going back, but every step along the way. As the history used by 'back' is not visible this behaviour can be very confusing. Note also that 'back' steps further and further back until any other command is used at which point the pointer jumps again to

the end of the list. At this point further use of ‘back’ will undo the last command, but then start acting a bit like a forward command, reversing the effect of previous undos!
In some ways the behaviour is quite reasonable, it records every state you are in – a true history. However, the sheer number of intermediate states with no visible list or menu to orientate oneself makes this very confusing. In fact, this behaviour is identical to that of Emacs undo, which appears to be equally confusing. Indeed, when experimenting with Emacs the authors were only able to make sense of its behaviour when they found explicit documentation on its undo mechanism.
Some experimental systems have largely similar history systems, but their history of visited nodes is explicitly displayed as a branching tree (Cockburn and Jones 1996; Ayers and Stasko 1995). An alternative would be simply not to record the ‘backwards’ states while undoing and to use simple video-recorder-like buttons to wind back and forward through recent history. However, given the variety of history mechanisms on offer, each presumably selected as the best candidate by its respective designer, it would be foolish to declare which is best without strong empirical evidence.

3.4.3 Combining ‘recent cards’ and ‘back’

The overall state of HyperCard requires both history and pointers:

$$S^{HC} = Nat \times S^{+} \times Nat \times S^{+}$$

As with Windows Help these two must share a common current state:

$$proj^{HC}([rp, rw, bp, bs]) = rw[rp] = bs[bp]$$

The recent window is updated by cards visited by ‘back’ in the same way as ordinary navigation. Also use of recent is treated as normal navigation by the

$$doit^{HC}([rp, rw, bp, bs], c) = [rp', rw', bp', bs'] \qquad c \in C$$
$$\text{where} \quad [rp', rw'] = addRecent(s', rp, rw)$$
$$[bp', bs'] = addBack(s', bp, bs)$$
$$s' = doit(rw[rp], c) = doit(bs[bp], c)$$

$$doit^{HC}([rp, rw\ , bp, bs], choose(n)) = [n, r, bp', bs']$$
$$\text{where} \quad [bp', bs'] = addBack(rw[n], bp, bs)$$

$$doit^{HC}([rp, rw, bp, bs], back) = [rp', rw', bp', bs']$$
$$\text{where} \quad [bp', bs'] = doBack(bp, bs)$$
$$[rp', rw'] = addRecent(bs'[bp'], rp, rw)$$

One thing to note here is the effect of ineffective commands, that is commands which do not change the current card. In the case of the recent window the new (unchanged) card is the same as one of the existing cards in the window, hence there is no change in the recent window. Similarly, the *addBack* function does not add superfluous copies to the end of its history list. However, as a side effect of such an ineffective command the ‘back’ pointer gets reset to point again to the end of the list. For example, suppose you have used ‘back’ to move back several cards and then open the ‘recent cards’ window. You change your mind and simply reselect the current card thinking this will have no effect (which is correct normally), however, this has reset the back pointer and so ‘back’ starts to behave like ‘forward’!

4. Comparing the methods

4.1 Back or not back?

All the browsers implement a single button press (in some cases also available as a function key or menu option) labelled ‘back’ or ‘go back’. At first sight these appear to behave equivalently. You navigate to a new node, you press ‘back’ and you are back to the previous node. In fact, so long as you only use ‘back’ once and haven’t just revisited a node, they do indeed all behave the same. This is OK, but what about when you use it repeatedly?

Actually, the equivalence is a little stronger than that. Even when you use 'back' several times (with no intervening commands) *all* the systems behave equivalently *so long as* (i) it has been a long time since you used 'back' last (more commands than the number of 'back's you are about to use) *and* (ii) there haven't been any recent revisited states. The first condition (i) is there because of HyperCard's rather odd backwards/forwards history of states.
If condition (i) is dropped then Windows Help, Think Reference and Netscape all behave identically. When no other history command is used and no state is visited more than once these mechanisms all behave identically – a simple stack. However, when the same state is revisited more differences emerge. Windows Help can have multiple instances of the same node one after another in the stack, Think Reference removes the older instance of a node from the history list when it is revisited and Netscape's stack does not have successive multiple instances unless they are to different labels in the same page!
So even for something as simple as going back there are four different interpretations for four different hypertext browsers.

4.2 Visible history

Each browser also had its own visible method of selecting from past states. This took the form of a textual menu or window in most cases and the graphical 'recent cards' window for HyperCard. The differences here are more marked between the browsers: a stack with pointer for Netscape, complete trace for Windows Help and different 'without repetition' versions of the history for Think Reference and HyperCard 'recent cards'.
In both the latter cases only a limited number of past states are recorded, so not having repetitions is important to avoid losing too many past states when the history is pruned. Think Reference retains the most recent entries, so that the list is always guaranteed to hold the most recent $n-1$ states where n is the length of the menu (it always keeps the very first state hence $n-1$). In contrast HyperCard prefers to keep the images of cards in the same position; that is, it exhibits a kind of display inertia or visual consistency. This same principle is continued when a whole row of images is scrolled off the top of the card. This 'wastes' screen space compared with simply deleting the first entry, but means that a card always occupies the same position in the row.

4.3 One mechanism or two

In two of the systems we considered, button press 'back' and visible history shared a common mechanism. In the other two they had separate mechanisms.
Those with separate mechanisms had *no* means of making the history use by 'back' visible. As we have seen, there are many different interpretations of the ideal behaviour of 'back', and the lack of visibility makes it difficult for users to learn the behaviour and to predict the effect of their actions. However, if the 'back' mechanism were made visible then the user would surely expect to be able to navigate using this as well as the alternative visible history, which leads naturally to a questioning of the need for two such mechanisms!
Netscape and the Think Reference get round this difficulty by having a single unified mechanism. However, in so doing they sacrifice one aspect or other. Netscape has a bias towards the idea of the stack and so does not keep track of all visited nodes, only those on the 'path' from the initial home page. On the other hand Think Reference keeps a complete track of all visited states (up to a maximum), but in so doing compromises the simple behaviour of 'back'.

4.4 Completeness of histories

The history mechanisms have different levels of completeness; that is, in how much information they retain.

- backtracking – Some 'back' commands remove the nodes through which they backtrack, either straightaway (Windows Help) or at the end of the 'history' subdialogue (Netscape – when ordinary navigation is used).
- resource limitations – Some systems have a maximum number of elements that can be in the history list: in HyperCard the number of card icons that can fit in the window, in Think Reference ten items, in Windows Help about forty, in Netscape apparently unlimited, but not all available from the menu (only fifteen under Windows 3.1 and thirty two on the Macintosh).
- repeated elements – Partly because of resource limitations some systems try to suppress repeated items. Many try to stop subsequent elements in the history being identical, others look at the whole history: Think Reference removing the previous occurrence and HyperCard recent not adding duplicates. This whole area is complicated by the different rules used to determine what is the 'same'!

- granularity – As well as issues of what constitutes the 'same' state, systems also differ on what constitutes a change of state. For example, in Windows Help we saw that selecting the current node in the history window did not get recorded as a new (repeated) state, but selecting the same node from further down the list did. In HyperCard, typing or scrolling text on a card does not constitute an action as far as 'back' is concerned (in the sense that the pointer keeps moving backwards), but any navigation action even when it does not change the current card (even using recent and reselecting the current card) breaks the chain of backward movement. There is also a similar sort of granularity effect in Netscape whereby a sequence of 'back' and 'forward' commands acts as a unit broken by the first 'normal' navigation command.

An obvious question to ask about a history mechanism is: if you have once visited a node in a hypertext can you guarantee to be able to find it again using the history mechanism? Because of either resource limitations or explicit 'forgetting' the answer is NO for all the mechanisms we have seen, with the exception of the HyperCard 'back' command (although even that may have some large internal limit we have not discovered). However, in the case of HyperCard although getting to any past state is *possible* it is hardly trivial!
It is strange that completeness is the property that you would expect of a visual 'history' window, but in fact the only example that is really complete (albeit complex) is a stepwise 'back'!

4.5 Searching

Whereas visual history suggests a record of past states, the obvious application of stepwise 'back' is to facilitate depth first searching. In fact, so long as the depth is not so great that resource limitations take effect and so long as there are no repeated nodes in the search tree, this is *possible* with all the 'back' mechanisms. In the cases of Netscape, Windows Help and Think Reference the obvious algorithm can be used:

1. Start with some initial node to search from
2. if you have investigated all interesting links from the current node:
   2.1. if this is the initial node: FINISH
   2.2. if not, use the 'back' command
   2.3. continue from the beginning of step 2
3. choose an interesting link from the current node
4. navigate to the chosen node
5. continue from step 2.

HyperCard is obviously rather different (!); however, strangely enough the same algorithm works. It works, but in a very different fashion. You will visit all the nodes *eventually*, but with an large number of repeats. Indeed, using the above algorithm, to search a tree with n nodes will typically (for a balanced tree) take around $O(n^2)$ commands, compared with about $2n$ for the other mechanisms.

5. History and undo

We have already noted that history and undo have many similarities.

5.1 State and conservativeness

In Section 2 we discussed the multi-level nature of browser state and the *conservative extension* relationship between the levels. In each example we have seen how given browser navigation behaviour (without history), modelled by the simple *doit* function, is extended by history mechanisms to give the overall system behaviour $doit^a$. Further, in each case we have seen how the overall system state space S^a is composed of several instances of the simple state (S) and sequences thereof.
Note that in each case the underlying browser navigation is different, but in fact any of the history mechanisms could have been applied to any underlying browser. For example, we could imagine using HyperCard's 'back' button and recent window in a web browser. The abstraction of the formalism can help us see the similarities and differences.
In addition, at this level descriptions of history and undo are virtually identical. In the case of undo, we also use a two-level model of the underlying system and the overall system augmented with undo (and possibly redo) commands.
The state conservativeness condition is also clearly necessary for any sensible undo system. In addition, for undo systems we have also used a similar condition for the command history. In addition to the *proj* function relating the augmented state S^a to the underlying state S, we also have a function *eff*, the effective history,

linking the augmented command history H^a and the underlying command history H. This should agree with *proj* in the sense that Figure 4 should commute. We call this relationship *encapsulation*.

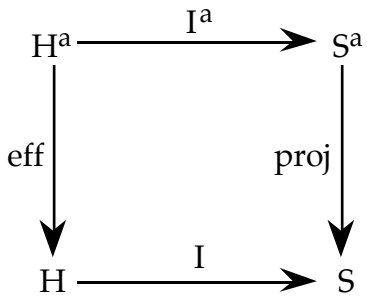

Figure 4 Encapsulation.

The effective history is thus a sequence of commands on the underlying system which would have given rise to the same underlying state as obtained by the full system (with history/undo). On its own it merely captures the fact that the augmented system cannot reach underlying states that are not reachable by the original commands. This is surely a property that should also hold for any history mechanism (and indeed does for all those considered here).
In addition, we can state a conservativeness condition on the command history:

$$eff(<>) = <>$$
$$\forall\ c \in C, h \in H^a : eff(h^\frown c) = eff(h)^\frown c$$

It can also be expressed as a commuting diagram (Figure 5).[2]} This basically says that ordinary commands simply add to the effective history – again, a property of any sensible history mechanism.

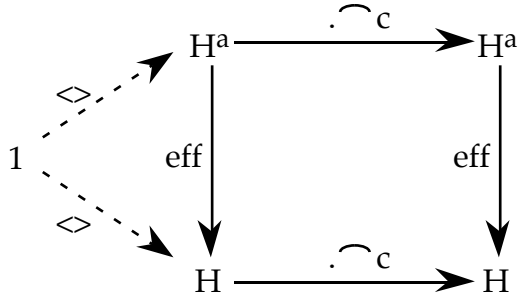

Figure 5 Conservativeness of effective history.

In fact, if an augmented system both is an encapsulation and has a conservative effective history, then it follows that the state is also conservative. These properties together we call a *conservative encapsulation* and this is the fundamental property that must hold true of any augmented system whether for undo or history. The three commuting diagrams can be put together into a single elegant (albeit somewhat full) commuting diagram we call *'the cube'* (Fig. 6).

[2]Note that '1' in the diagram is the set with only one element. It is used as a categorical 'trick' to capture an initial state within a commuting diagram.

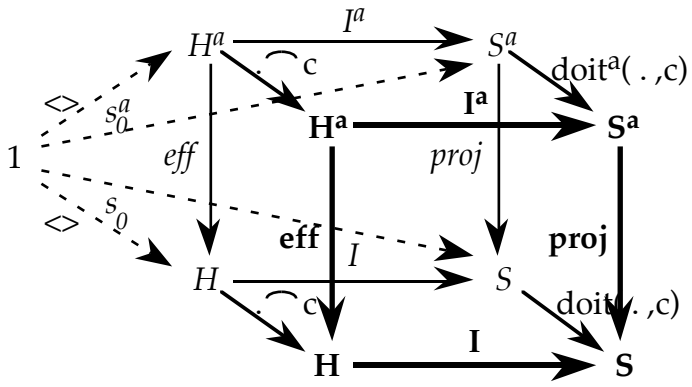

Figure 6 The cube.

Note that *conservative encapsulation* only tells us what happens when the additional facilities are *not* being used – it is about the preservation of the original system within the augmented system. The individual behaviour of the additional commands (for history or undo) need to be considered separately.

5.2 Names, states, icons and actions

The cube links together the user actions and state in a system. These to some extent form complementary views of the behaviour of a system. One of the biggest differences between history and undo systems is not their behaviour, but the presentation of history to the user.

In the case of a single 'back' or 'undo' action they both behave similarly; the focus is on the currently displayed state. The reflexive aspect is weak – although users have to think sufficiently *about* the interaction to decide to use back or undo, they do not see any representation of the history itself.

When we look at visible history lists and undo menus the situation is quite different. We noted that the history mechanism in Netscape is identical, in terms of its formal description, to that of Microsoft Word 6. In each case there is a list of states with a pointer to the current location within the list. The back (or undo) command moves the pointer back through the list, the forward (redo) command moves the pointer forward. In the Netscape menu one is shown exactly this, a list of *names* of the pages one has visited with a tick on the current one (where the pointer is). In contrast, Word 6 has two menus, the undo menu listing the *commands* that got you to the current state and the redo list showing the *commands* that lead on from the current state.

This reveals a fundamentally different orientation in the way we look at and talk about the system. Hypertext systems talk about *where* you have been; undo systems about *how* you got there.

In a word processor there is no convenient name for the states. They are simply the same document with more or fewer changes and edits. Even if the system has some name for them, it is unlikely that it will be meaningful for the user (with the possible exception of timestamps). The natural thing for the user is to think "I'd like the document as it was before I deleted the big paragraph".

In contrast many hypertexts have convenient titles for the pages or nodes; for example, web pages have a `<title>` tag to give the page a name, and also a URL. This forms an easy reference point for the user (so long as the titles are sufficiently informative!). It is also more natural for the user to think "I'd like to get back to that page about history mechanisms".

HyperCard faces problems here as it has some database-like aspects and it is quite common to have many anonymous cards. A menu with items such as 'card id 7392' would not be generally accepted as user friendly! This is why it opts for the alternative iconic approach, relying on our ability to match visual features. This would obviously not be very useful on the heavily text-based pages of a Help system, and would be totally useless for an undo facility of a word-processor, but might work for the WWW. In fact, on the Macintosh, many graphics programs set the icon of files to be a thumbnail of the file contents, on the principle that this will be more readily recognisable than the file name. Unfortunately, HyperCard still has problems with database-style applications as all the records are simply copies of the same card with different details: not very easy to distinguish from a thumbnail!

This state vs. command emphasis is also evident if you try to construct the effective history functions for the different history mechanisms. It is possible to construct them, but difficult to do so without explicit reference to the state. This is because the rules for adding to the history are more semantic based than those for undo, which simply add commands of a particular class whether or not they were effective in the particular context. Because

undo is more command oriented it tends to have simpler (syntactic) algebraic properties. For example, with a pure backtrack undo and any normal command *c* in *any* context we have:

$$c \quad undo \quad \sim \quad null$$

An equivalent history property for most of the systems we have looked at would need a side condition to say "except in contexts where the command *c* is passive".

6. Summary

Given the number of different solutions to the design of a history mechanism, it is not surprising that users find it difficult to comprehend! By modelling these different solutions within a generic formal framework, we have been able to examine the history mechanisms in isolation from other features and also to see the similarities and differences between systems. In addition it has allowed us to see clearly the connection between history and undo mechanisms. However, it has also highlighted important differences between history in hypertext systems and undo in manipulative systems in terms of the emphasis on state vs. commands, and in the notion of state itself.

We have seen that even the simplest history mechanism, the 'back' button, has had a different interpretation in each system studied. This suggests that users need a clear indication of what 'back' is doing. However, we also saw that attempts to unify the expected behaviour of a 'back' button with a visual history posed problems as well.

Although many people report confusion with Netscape's history mechanism it is clearly one of the more straightforward offerings! Its main fault in terms of its underlying model is that it is not a full history. However, the source of confusion does not lie in this but in the comprehension of what constitutes going to a new page. This confusion is not helped by the fact that there is nothing in the 'go' menu to indicate visually which entries are to labelled positions within pages, and which to the page as a whole.

In this chapter we have not sought to give an overall judgement on the best history mechanism but to specify clearly those mechanisms that are in use and some of the trade-offs between different desirable properties. The range of different mechanisms shows that there is no current consensus on what constitutes the best form of hypertext history. Indeed, the authors have also analysed Internet Explorer and found that it embodies three different history mechanisms. An additional complication that we have not considered in the paper is the behaviour of 'back' in web browsers that support frames. When users are allowed to go 'back' on a frame-by-frame basis it is possible to reach states (in terms of combinations of frame contents) that are *unreachable* via ordinary browsing. Such a state of affairs would *not* constitute an encapsulation.

Developers of history mechanisms are faced with a hard task. Modern interfaces emphasise action – doing things, whereas history is reflexive – talking about doing. Formal specification can help comparisons between different models for this, and the reflexive nature is clearly captured in the models we have presented. However, the challenge for the developer is finding appropriate ways to portray and interact with this reflexive information at the user interface.

Acknowledgements

Many thanks to Fiona for proofreading this chapter and converting it, including all the mathematics, into Word.

References

E. Ayers and J. Stasko (1995). Using graphic history in browsing the World Wide Web. *Proceedings of the Fourth International World Wide Web Conference*, http://www.w3.org/pub/Conferences/WWW4/papers2/270/, Boston,

M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian and H. Oinas-Kukkonen (1997). Fourth Generation Hypermedia: Some Missing Links for the World Wide Web. *International Journal of Human Computer Studies (forthcoming),* .

L. Catledge and J. Pitkow (1995). Characterizing browsing strategies in the World-Wide Web. *Proceedings of the Third International World Wide Web Conference*, http://www.igd.fhg.de/www/www95/papers/, Darmstadt, Germany,

A. Cockburn and S. Jones (1996). Which way now? Analysing and easing inadequacies in Web navigation. *International Journal of Human Computer Studies,* **44**.

J. Conklin (1987). Hypertext: an introduction and survey. *IEEE Computer,* **20**(9): 17–41.

A. Dix, R. Mancini and S. Levialdi (1996). Alas I am undone - Reducing the risk of interaction? *HCI'96 Adjunct Proceedings*, Imperial College, London, pp. 51–56.
A. Dix, R. Mancini and S. Levialdi (1997). Communication, action and history. *Proceedings of CHI'97*, Atlanta, USA, ACM Press. pp. 542–543.
A. J. Dix (1991). *Formal Methods for Interactive Systems*. Academic Press.
A. J. Dix and C. Runciman (1985). Abstract models of interactive systems. *People and Computers: Designing the Interface*, , Cambridge University Press. pp. 13-22.
P. Pirolli and S. Card (1995). Information foraging in information access environments. *Proceedings of CHI'95*, , ACM Press. pp. 51–58.
L. Tauscher and S. Greenberg (1997). How people revisit web pages: empirical findings and implications for the design of history systems. *International Journal of Human Computer Studies (forthcoming),* .