A hypertext system for UNIX

P.J. Brown
Computing Laboratory
University of Kent at Canterbury

October 1988, revised December 1988

Hypertext is concerned with on-line documentation. Over the past two years, interest in hypertext has grown dramatically and there has been a corresponding explosion in the number of hypertext systems, journal articles and, indeed, hypertext experts.

One of the deficiencies of many of today's hypertext systems is that they are closed systems: they do not work well with their fellow tools. Though it may be possible to exchange information with other tools, this needs to be done via conversion utilities or the like. As Meyrowitz [1987] highlighted in his influential contribution to the Hypertext 87 conference, this drains the blood out of the hypertext system. If hypertext systems are to realise their full potential, they must aim for a seamless interface with other tools, thus making the whole greater than the sum of the parts. Such a philosophy is, of course, one of the cornerstones of UNIX.

The purpose of this paper is to describe how the Guide hypertext system has been designed to exploit the UNIX environment. The word 'seamless', like the world 'user-friendly', is wantonly applied to almost all software: it is fast becoming a noise-word. If we are to stick to the true meaning of seamlessness, Guide cannot claim to have attained it. Nevertheless the claim of this paper is that it has been able to gain a lot of value from interchanging information with its UNIX environment.

It is no use writing a paper about integration unless readers understand what is being integrated. Since some readers will be unfamiliar with hypertext in general and Guide in particular, so we shall introduce these first.

History of Guide

Development of Guide began at the University of Kent at Canterbury in 1982. The work at Kent has continued since, and has been based throughout on UNIX workstations that support a graphical 'WIMPS' interface. The work has been centred on SUN machines and, more recently, on all those workstations that support the X11 window system [Scheifler and Gettys 1986].

In 1984 Office Workstations Ltd. (OWL) became interested in Guide; they adapted it to fit the Macintosh environment and subsequently the IBM PC environment. OWL made many changes in their Guide product, most of which were related to fitting a different environment and a different marketplace.

Development of UNIX Guide has continued at the University, benefitting both from OWL's new ideas and the experiences of UNIX Guide applications. As a result of lessons learned from the latter, a main thrust of the University's work has been to try to exploit to the full the power of its UNIX environment.

Since this paper is very much a UNIX one, we deal mainly with UNIX Guide. References to Guide henceforth should therefore be taken to mean UNIX Guide, though we shall refer again to Macintosh Guide when we discuss user interfaces.

Basic concepts

The essence of hypertext is that it is non-linear. When perusing material presented by a hypertext system, readers generally have various options on where to go next, depending on what interests them. The underlying hypertext document consists of pieces of text (and/or other media) linked together in a directed graph structure. Links can either be *hierarchical* or *cross-reference*. Hierarchical links provide successive levels of detail. Thus a reader may start with an overview, and then, by following hierarchical links, might gradually expand the level of detail on selected topics until he has gleaned the information he needs. Alternatively the reader may wish to follow cross-reference links, which lead to other, related, information. A requirement for a successful hypertext system is that linking should be extremely simple. In Guide links are represented by *buttons* (i.e. hot spots) embedded within the text (and/or pictures) on the screen. The material linked to by a button is called its *replacement*. The reader just selects any required button using the mouse (or whatever other pointing device is in use). Figure 1 shows a small extract from a Guide document.

Figure 1: Guide displaying part of a document

There are three buttons embedded in the document. Two buttons are in bold (**Example** and **More**) and represent hierarchical links; if one of these buttons is selected its replacement is inserted in-line in place of the original button. As a simple example, Figure 2 shows the result of selecting the **More** button in Figure 1: the button is replaced by some expanded material.

Figure 2: The result of selecting the **More** *button in Figure 1*

There is also a cross-reference link in Figure 1 (and, indeed, in Figure 2), represented by the *write-permission* button; if this is selected, its replacement appears in a separate sub-window. The user may scroll to read linearly through a document. The user may also ‘undo’ any replacement.

Guide supports both text and graphics. In this paper we will use the neutral term *material* to mean any mixture of text and graphics.

For further details of the principles of hypertext see Conklin [1987], and for further details of Guide see Brown [1986].

Requirements

One approach to persuading people to adopt a hypertext systems is: ‘‘I have this excellent hypertext system which will revolutionise the way information is presented; I suggest you rewrite all your existing documentation to fit it’’. Sadly, this approach always fails. A requirement for any hypertext system is that it should be able to capture existing documentation, even though the captured documentation will inevitably not exploit the hypertext system to the full. Ideally the capture should be possible as a regular on-the-fly process, rather than a once and for all conversion. The captured material may, for example, be a program that is created and maintained for use by a C compiler, but is occasionally displayed by a hypertext system (e.g. as part of the overall system documentation).

A second requirement is the complement of the first: material created by the hypertext system should be available to other systems.

A third requirement is the ability to exploit existing UNIX utilities, such as the spelling checker or printing tools. Happily this requirement is generally substained by the two requirements above.

Lastly, when users come to exploit any software of reasonable sophistication they soon encounter a need to program it. Hypertext systems are no exception to this. There is a requirement to introduce conditional facilities so that material can automatically be tailored to the needs and perhaps to the status of the user. We discuss this in more detail later.

All four requirements are, of course, facets of the overall requirement of integrating with the environment.

One key to meeting the requirements is file formats: clearly the hypertext system will require its file format to cater for structural information that is unique to that hypertext system; on the other hand file formats should be suitable for use by other utilities, which may be uninterested in the structural information. Since this issue is so vital, we shall start our discussion of Guide’s integration with UNIX by examining how it tries to meet these potentially clashing demands.

Files

Perhaps the most natural way of representing hypertext material is to use a large number of (usually) small files so that each unit of a hypertext document is in a separate file, and the links connect these files together. Experience in IPSEs (Integrated Project Support Environments) and in other areas shows, however, that small granularity leads to poor performance. The decision was made in Guide, therefore, not to commit to a small granularity but to allow authors to combine a lot of interlinked material into a single file. In practice this has allowed authors who are concerned with performance to optimise according to the pattern of usage. Authors have tended to start by building their hypertext document in a single file — Guide default options encourage this — and, if the file becomes so long that the initial load time becomes a problem, to split the file into sub-files.

It has turned out that the perceived needs of applications have varied widely. In one Guide application concerned with displaying legal information [Wilson 1988], over a megabyte of information is stored in a single file. In another application, the author has chosen to scatter 3½ megabytes of information over some 793 files.

File content

In a Guide file the structural information is embedded within the text and graphics that make up the body of the file. A sample piece of structural information might specify that where a button begins and ends within a file. Buttons also have properties, which we will not go into here, and these are encoded within the structural information. This structural information is represented in the form of *troff* requests. Thus a button called **More** might be represented as

```
 .Bu  1  1  n
More
 .bU
```

Here the *Bu* request means the start of a button, and is followed by a list of the button's properties. The *bU* request means the end of the button.

Clearly if one had a free choice of notations, *troff*'s notation would not be one's first choice. However its use by Guide has had a huge number of advantages, mostly stemming from the fact that the *troff* notation is a UNIX standard for representing documents. In particular:

(1) Utilities such as *spell* can work directly on Guide files, because *spell* automatically uses *deroff* to filter out *troff* requests. (For other utilities it may be necessary to use *deroff* explicitly.)

(2) It is possible to write *troff* macros corresponding to Guide requests such as *Bu*, and thereby to use *troff* to format Guide files. For example, buttons can be printed in bold with, perhaps, superscripts to indicate the button's properties.

(3) Guide requests can be embedded in ordinary *troff* files. If such files are fed to *troff*, without defining the macros described in (2) above, then the Guide requests will be ignored. In this context it is a happy property of *troff* that it ignores any requests it does not understand and gives no error messages. This property has been exploited to allow the same file to be used both as a manual page and as a Guide help file, and we shall discuss this later. (The names of Guide requests all consist of one upper-case letter and one lower-case one, thus avoiding clashes with normal *troff* requests.)

Guide can read an ordinary text file, since such a file simply looks like a Guide file with no requests in it. Indeed Guide can be used as an text editor, though this was not an intended use.

Pictures

Guide allows for bit-map pictures, which are stored in binary, and these can be embedded within Guide files. The majority of UNIX tools are textual and can be upset by strings of arbitrary binary codes, so Guide does its best to ameliorate this problem. Firstly the bytes representing a picture are divided up into ‘lines’ of not more than 80 characters when stored in a Guide file, thus preventing other tools being knocked out by long ‘lines’. Secondly, each line of the bit-map is prefixed with the *troff* comment request (.\"), thus causing utilities that automatically use *deroff* to skip the pictures. Properties of pictures such as their size and number of colour planes are represented in ASCII to help with portability.

Standard notation

Obviously in the future it would be useful to move on to the use of some standard document representation notation such as ODA (Office Document Architecture) [Horak 1985], and it would not be hard to switch to this. However for the immediate future the advantages of the *troff* notation within the UNIX environment are compelling.

Representing replacements

We now move on to a more esoteric — and, for those not interested, skippable — topic: the representation of hypertext links in Guide files. The user may imagine these as pointers (e.g. a cross-reference link points at its destination). However files containing explicit pointers, either to other positions within the same file or to offsets within a different file, are fragile: any other tool that manipulates the file by adding or deleting material will invalidate links. Partly for this reason, Guide files do not contain explicit links. Instead, the basic mechanisms are as follows.

Associated with each cross-reference button is a *definition*, which supplies the material that is to act as a replacement of the cross-reference button. As a small example, consider the definition of a button called ‘Safety Manager’. Its replacement might consist of the name ‘Mr Smith’, followed perhaps by some instructions as to how to find him. There may be several cross-reference buttons within a Guide document that use this definition.

Linking between cross-reference buttons and definitions is done simply by name matching; there are no explicit links between the two. Indeed different users can use different sets of definitions for the same term, e.g. the Safety Manager might depend on the user’s location. (In this case the author would provide different sets of definitions, together with criteria for selecting which set to load.)

Definitions may be in the same file as their usage(s) or they can be in a separate file. In the latter case the filename can be a UNIX symbolic link rather than an absolute filename, thus making change easier. The author of a document is responsible for defining how files are to be organised, and for specifying the files to be used for definitions.

The merits, within a hypertext system, of this scheme of name matching as against a more direct linking system can be argued. However name matching has the advantage that it re-enforces the textual nature of Guide files, and makes them more portable and easier to process by other tools.

Within a Guide file, hierarchical buttons are usually immediately followed by their replacement, and hence no linking is involved. Thus if we assume our earlier **More** button is a hierarchical button its replacement might be represented thus

```
.Bu  l  l  n
More
.bU
.Re
not  only  is  the  name  of  the  file  ...
 ...
 ...  lost  for  ever.
.rE
```

The *Re* and *rE* requests show the beginning and end of the replacement.

(To be precise, the Guide hierarchy need not be a tree: instead two separate points in the 'hierarchy' can use the same replacement, although the structure will appear to the user to be hierarchical. In such cases hierarchical buttons use the same name-matching mechanisms as cross-reference buttons.)

Capturing existing documents

Any document that is marked up in a systematic way to indicate its structure can be converted to Guide form. For example a section-heading can be converted to a hierarchical button with the body of the section made into its replacement. With a little more effort citations within the document, e.g. [Bloggs 1988], can be turned into cross-reference buttons. With even more effort, constructions of the form 'see page 83' can also be turned into cross-reference buttons. All that is needed to accomplish such conversions is a filter. Guide allows material to be piped into it, so filters are easy to use. The use of automatic filters will inevitably produce imperfect results, and could not match an intelligent hand-crafted conversion; nevertheless with large volumes of material there is no alternative, and even a crude conversion to hypertext form can be much more attractive to read than the original.

As one practical example of such work, macros have been written to convert UNIX manual pages to Guide form. This is done by using an alternative form of the *man* macros, which produce a structured Guide file rather than the normal formatted output that is produced by the *man* command. In this file, headings have been turned into Guide buttons.

Guide can then be used to view a manual page, and the reader can expand the headings he wants to examine. There are also further advantages: for example if the user changes the size of the window currently being used to view the manual page, then Guide will automatically reformat paragraphs to fit the new screen size.

Manual pages can be further enhanced to exploit other Guide facilities, such as the use of cross-reference buttons to explain jargon terms or to follow SEE ALSO links. This can be done without prejudicing the use of the file as a normal manual page, since *troff* will ignore the extra Guide requests. In particular this has been used to allow a manual page to serve the secondary purpose as a help-file. When Guide is called up to act as a help system it converts the manual page to its own form, and focusses on the part of the document that is likely to help the user in his current state. The manual page may be augmented by extra tutorial material that is seen by Guide but treated as a comment by *troff*.

Programmability

Several different kinds of programmability are needed in hypertext systems. Firstly a hypertext system that caters solely for static pre-defined material is limited in its application. For more generality it is desirable to allow material to be generated dynamically by running a program. (Such a program may, for example, extract from a stock control system some information about the current stock of a certain product. This gives an immediacy that is not possible if pre-

prepared material is used.) In a UNIX environment such programmability is easy to achieve. The author of a Guide document can attach a shell script to a button — in this case we shall assume that the button is called **Stock**. (The default shell used throughout Guide is, in fact, the Bourne shell.) This shell script is executed when the **Stock** button is selected and the output from the shell script (both the standard output and any error output) are piped back into Guide. Guide displays this output as the replacement of the button. The user is normally unaware that whether material is static or dynamic — the user is only interested in reading the material, not in how Guide produced it. When initiating a shell script, Guide sets a small number of environment variables in order to tell the shell script about the current Guide environment.

The above kind of programmability is not hard to provide, though there are some challenges in providing good error recovery when authors inadvertently do mad things in shell scripts initiated from Guide.

A second kind of programmability, which is much more challenging, is to allow programmability of Guide itself. Such applications arise particularly when Guide is used as a front-end: Guide's strength is in displaying information, and it can be profitably coupled with other tools that extract or generate information. An example of the kind of facilities needed is as follows. An author may want the user, when he reaches a certain point in a Guide document — or when some other tool sends a signal to Guide — to have an option to initiate the following sequence of operations:

(1) By means of a dialogue, get the user to specify a filename.

(2) Save the current Guide document in the file specified by the user.

(3) Load a new Guide document in place of the current one.

(4) Select certain buttons in the new document, so that the user is focussed on a particular piece of information within the document. It may even be required that the buttons selected in the new document depend on the state of the old — e.g. whether the user had previously selected an **Expert** button lying within the old material.

The facility for initiating this sequence of actions may itself be made available by means of a button embedded in the document. Indeed Guide supports such 'action-buttons', which complement the more normal 'replace-buttons' that generate a replacement.

Now there are certainly dangers if authors develop a boundless enthusiasm for such programming. A long sequence of automatic actions may be played out while the user watches bemused. However there is undoubtedly a good case for using *some* programmability, and this requirement needs to be met.

The temptation is to create a completely new programming language for programming Guide, of a roughly similar nature to Apple's HyperTalk language for programming HyperCard [Goodman 1987]. The facilities that need to be covered are:

(a) selecting Guide menu commands (such as *Save* in our example).

(b) selecting/undoing buttons.

(c) testing the current state of buttons (e.g. testing if an **Expert** button has been selected).

(d) dialogue with the user.

(e) file manipulation.

(f) conditionals, looping, case statements, etc.

(g) variables of various data types.

However, although Apple, with all its resources, have made a great success of HyperTalk, it is nevertheless best to avoid the temptation to create a new language unless the need is absolute. In particular in a UNIX environment there already exist programming languages likely to be familiar to most authors, and it is best to take advantage of these. Thus it was decided to only provide a simple programming language to cover operations (a) to (d) above, which are unique to Guide, and to use an existing programming language to cover the rest. The operations (a) to (d) can in fact be provided by a set of about thirty functions, which provide handles on the main elements of Guide's functionality. Thus Guide supports a mini-language, which we shall here call *MiniL*; a 'program' in MiniL just consists of a sequence of calls of the built-in functions.

MiniL programs can be executed in one of three ways:

(1) by attaching the program to a Guide button.

(2) by executing the program from within a shell script initiated by Guide. In this case a special *callguide* command is provided. This command takes as its argument a MiniL program and returns an exit status to indicate whether the program was successful. It might also return strings (e.g. material typed by the user if the program involves a dialogue); such strings are sent to the standard output.

(3) by using a roughly similar mechanism to (2), but within a C program rather than a shell script. Here there is a library function called *callguide*.

As an example, if an author wanted to add simple conditional facilities on top of a MiniL program he might use approach (2) and write a shell script of form:

```
callguide ´a MiniL program´
if test $? -eq 0; then
        ...
else
        ...
fi
```

Alternatively, if more elaborate programmability is required, the MiniL program can be executed from within a C program.

Approaches (2) and (3), where MiniL programs are embedded in another programming language (a shell or the C language), are an example of a *mixed-language* approach to programming.

Evaluation of the approach

The first point to be made about programming Guide is that it is only used by a minority of authors. Maybe it is only after you have learned to program that you want to program everything in sight. A lot of Guide authors have no knowledge of programming and see no need for it within Guide.

For those authors who do want to program the mixed-language approach has attractions. In the majority of applications MiniL on its own is adequate. The MiniL manual is only six pages long, and does not represent a big learning hump. When further programming is needed — and we are now down to a few percent of the applications — authors can revert to familiar and well-loved (well-hated?) languages, such as a UNIX shell or C.

Nevertheless there is, as ever, a price to pay and a mixed-language approach like Guide's is certainly not sensible for every circumstance where programmability is needed. The two main disadvantages, which apply strongly to the use of *callguide* within shell scripts and which become increasingly tiresome as programs get larger, are:

(1) shell scripts (and to a lesser extent C programs) with lots of embedded uses of *callguide* run slowly.

(2) programming in a mixed notation is error-prone. It is easy to make trivial syntactic errors, particularly with the use of quotes. For example a MiniL program can involve quotes and the program may itself be enclosed in quotes since it is an argument to *callguide*. More fundamentally the human mind has problems in switching notations every few lines. Similar problems arise when, say, *awk* or *sed* scripts are embedded in shell scripts.

User interface

As we have said, it is a vital characteristic of a hypertext system that following links should be simple and natural. All experience shows that if a hypertext system lacks this property (and just because a software designer says something is simple, it does not follow that users will say the same), the hypertext system will never be used.

Guide has successfully achieved simplicity through the using a WIMPS interface as a foundation. There is, however, a compile-time option in Guide which causes it to use the *curses* package and run on a standard terminal with no pointing device. This alternative interface has not, however, been a success, since following links is no longer simple and natural. The key problem is that the *curses* version of Guide simply provides a pale imitation of the WIMPS interface. It should be possible to write a simple and natural interface that runs on a minimal terminal, but this would not be achieved by redesigning the user interface to fit this environment rather than imitating parts of a WIMPS interface.

When the user interface is such a vital component to the success of a system it is vain to hope that minor tweaking will cover a host of different user environments.

The above point, which relates to different implementations within UNIX, applies even more strongly when contrasting non-UNIX implementations with UNIX ones. It may therefore be of interest to compare the UNIX implementation of Guide to OWL's Macintosh implementation. Although the two versions of Guide are similar in what they do, their appearance differs considerably. The OWL product is well-integrated with the Macintosh environment, and therefore follows the strong Macintosh house-style. In addition to determining look-and-feel this also affects the way things are done: for example Macintosh users expect searching to be done by using a pull-down menu that brings up a dialogue box which offers certain standard options. At the time Guide was developed there was not even a hint of a house-style for UNIX workstations. UNIX Guide was therefore influenced only by the user interfaces of the few graphical tools that were available. Nowadays, however, a number of house-styles for graphics workstations are being promoted. Nevertheless we feel that it is unlikely that a Guide user interface that allowed it to integrate well in a UNIX environment would closely resemble a Macintosh environment, or indeed any other environment designed for radically different users, hardware and operating system.

Conclusion

Over the last five years the technique of cut-and-paste has become widely known and well-integrated throughout a range of software packages. Nowadays there is a large body of users who use cut-and-paste in such a familiar way that its use is almost subconscious; users expect the facility to be available on all software for which it is relevant. Meyrowitz looks forward to the day when hypertext techniques achieve similar familiarity and integration. We still have a long way to go. Indeed achieving the dream is not entirely under the control of designers of hypertext systems, since it requires the collaboration of the creators of operating systems, window systems, toolkits, etc.

Guide cannot claim to be completely integrated into the Unix environment. With its WIMPS interface it has a different look and feel from the traditional UNIX tools. Nevertheless Guide can effectively share information with other UNIX tools, and also builds on the existing programming features of UNIX. Guide gains enormously from the surrounding UNIX environment, and hopefully a UNIX environment also gains from the presence of Guide.

References

P.J. Brown, A simple mechanism for the authorship of dynamic documents, in van Vliet (Ed.): *Text processing and document preparation*, pp. 34-42, 1986.

P.J. Brown and M.T. Russell, Converting help systems to hypertext, *Software — Practice and Experience 18, 2*, pp. 163-165, 1988.

J. Conklin, Hypertext: introduction and survey, *IEE Computer 20, 9*, pp. 17-41, 1987.

D. Goodman, *The complete HyperCard handbook*, Bantam Books, N.Y., 1987.

W. Horak, Office document architecture and office document interchange formats: current status of international standardization, *IEE Computer 18, 10*, pp. 50-60, 1985.

N. Meyrowitz, *The missing link: why we are all doing it wrong*, position paper, Hypertext 87, Univ. of N. Carolina, 1987.

R.W. Scheifler and J. Gettys, The X window system, *ACM Transactions on Graphics 5, 2*, pp. 79-109. 1986.

E. Wilson, Integrated information retrieval for law in a hypertext environment, Yves (Ed.): *ACM SIGIR 11th Annual Conference*, pp. 663-677, 1988.