ELECTRONIC PUBLISHING, VOL . 1(19), 1–4 (JUNE 1995)

Embedded or separate hypertext mark-up: is it an issue?

P.J. Brown, Heather Brown

Computing Laboratory
The University
Canterbury
Kent, CT2 7NF

SUMMARY

Most hypertext systems used in the field embed some form of mark-up in each hyperdocument in order to represent the hypertext structure. Indeed, more generally, most document preparation systems use this approach. Hypertext researchers, on the other hand, say that the structure of a hyperdocument should be separate from its content. This paper investigates whether the two approaches, embedded v. separate, are really at odds with one another, and describes a technology for combining some of the benefits of both.

KEY WORDS Hypertext Mark-up Guide WWW CD-ROM UNIX Microcosm Hyper-G

1 INTRODUCTION

Mark-up of any document can either be embedded within the document or stored separately. In the hypertext field, all the expert researchers will say that separate mark-up is the only respectable approach — the structure should be separate from the content, whereas virtually all the hypertext systems that are widely used in the field are based on embedded mark-up. This paper examines whether the issue of embedded versus separate mark-up really is an important one.

We will start with four background points — highly diverse in nature — which will throw light on the central issue.

1.1 Background point 1: the experts and the masses

In any field the 'experts' despise the choice of the masses. At the time of writing the best-selling book in the UK is a thriller by Dick Francis. If, however, you suggested to Professor of English Literature that he should replace his course on, say, the works of T.S. Eliot by a course on the works of Dick Francis, you are unlikely to get a reply that treats you as an intelligent being.

The dichotomy between the experts and the masses applies in computing as much as any other field. Try suggesting to a University Computing Science department that BASIC be made the main programming language that is taught.

Thus the clash between the hypertext experts and the masses should not come as a surprise. Note that in each field the expert's choice requires more effort to understand: sometimes, as in the T.S. Eliot case, it may be beyond the comprehension of many.

1.2 Background point 2: is small still beautiful?

A lot of us feel that small-is-beautiful is a good guiding principle for computer software. Indeed one of the reasons for the original success of UNIX was it encompassed this principle. The UNIX philosophy was (is?) that monolithic software tools are wrong. Instead you should have a collection of small tools, each good at one particular task, that are easy to connect together so that a set of tools can co-operate in performing a task. In particular UNIX provides the pipe as a connecting mechanism, though pipes work at a low level (an unstructured stream of bytes).

If this philosophy is applied to hypertext systems then hypertext systems should just do hypertext. They should not, for example, offer clever editing facilities (pattern matching and so on) since a specialist editor would do the job better. As a corollary, it should be easy to switch between the hypertext system and the editor and in particular the editor should be able to process the hypertext system's data formats.

An extension of this point is the 'ubiquitous hypertext' feature we mention later.

1.3 Background point 3: confusing embedded mark-up with content

A recently posted *troff* document reads: 'submit your contribution in the following form:

BibTeX

```
.NH
Your title
.LP
...'
```

Devotees of *troff* will recognize a familiar trap here: the content contains lines such as .NH and .LP that are *troff* mark-up lines. The author therefore needs to mark these lines in a special way so that they are treated as content, not as mark-up.

The same point applies to any other embedded mark-up: unless the character set used in the mark-up is separate from that used in the content, there is a danger of parts of the content being mistaken for mark-up.

1.4 Background point 4: Knuth's literate programming

Donald Knuth [6] has championed an approach to program documentation called 'literate programming'. In this approach the documentation of the program, with its embedded mark-up, is itself embedded in the program, the idea being that you can use this composite whole to produce a well-formatted and, one hopes, literate explanation of the program. There are tools to extract a formatted document from the composite, and alternative tools to extract the program itself in a form that a compiler would accept.

Knuth's idea can be generalised to cover other pairs of related texts, e.g. specification and design. The essence is that by interleaving two (or more) related documents, you make it easier to see the relationships between them.

On the face of it, Knuth is compounding the sin of embedding.

2 BASICS

These background points will come into play later. It is now time to start the basic exposition of this paper by explaining some terminology, in particular *mark-up* and *content*. The mark-up delimits and describes the logical objects of the content. The content is the underlying text, pictures, video, etc. Actually, the

BIBTEXmark-up is sometimes concerned with physical appearance, e.g. 'switch to bold face', but we shall concentrate on the logical level here. Mark-up can be at several different levels:

(1) *embedded positional mark-up*: here the mark-up is embedded in the raw content at the position it is to be applied. This is the most common form of mark-up: typical items of mark-up say that a title begins/ends here, or the source of a hypertext link begins/ends here. Since almost all forms of embedded mark-up are positional, we will simply call this *embedded mark-up*.
(2) *separate positional mark-up*: this is conceptually similar to (1), except that the mark-up is stored in a file (or table, etc.) separate from the content. At its lowest level this mark-up takes the form 'in file XXX a title runs from byte 75 to byte 88, the source of a link runs from 100 to 106, ... '. Two possible elaborations are to use higher-level objects than bytes (e.g. words for text), and to allow the same mark-up to bring together several content files.
(3) *(separate) calculated mark-up*: at the next level the mark-up involves some (semi-)intelligent processing of the content file, e.g. 'search for tag XXX or string YYY, and make this the start of a hypertext link'. HyTime [8], for example, supports a particularly rich set of features of this nature. This form of mark-up is rarely used at the moment, but might become more popular in the future: its disadvantage is that it is not the sort of mark-up that an author can create behind the scenes by creating logical objects using a WYSIWYG editor (e.g. by making the selected text a link source), but needs to be created explicitly by an intelligent author. Its advantage is that it is much less fragile, if the underlying content changes, than byte offsets.
(4) *generated mark-up*: in this case there is no pre-existing mark-up, but a program is run to create the mark-up for a given content file. The program could, for example, try to identify titles in a raw text file, and insert appropriate mark-up round them. Often such programs have associated tables. For example a table could specify a set of words, and the job of the program could be to find all occurrences of these words in the content file, and to turn each into some specified form of link. In all cases the program is likely to generate mark-up of form (1) or (2) above, which can then be used directly by an underlying hypertext system.
(5) *dynamic generated mark-up*: finally, we can carry (4) one stage further, and run the program as the user views the document. For example when the user selects an object on the screen (a word, a phrase, a picture) the program can then be run to see if the object is, for example, the source of a link. (The program could do this by searching tables and dictionaries to see if there is any further information about the object, and, if there is, to link to this information.) Microcosm [4], with its *Generic links* and *Compute links*, has excellent features of this nature. Overall the dynamic approach has gone way beyond what we normally think of as mark-up, and, indeed, way beyond the scope of this paper.

It is worth saying a bit more about what we have called calculated mark-up. If the content is known to be in a form where logical objects within it can be identified, the way is open for the calculated mark-up to recognise and use these logical objects.

Examples where there is potential for the new mark-up to use knowledge about the structure of the content are:

(a) a C program. Here logical objects such as functions can be identified. Mark-up on top of a C program could therefore refer to the functions within it, e.g. the declaration of function XXX could be made the object of a link.
(b) a PostScript program. The output from such program is divided into pages, and thus calculated mark-up on top of it could identify and refer to pages.
(c) a document that is already marked-up in, say, *troff*. Here logical objects such as titles should be

BIBTEX visible.

(d) a WWW document, marked-up in HTML. Again, logical objects such as titles should be visible. Interestingly, the new, separate, mark-up could also be HTML-based. This would apply if an author wanted to add mark-up to some-one else's WWW page, e.g. 'Add the following HTML after the title in WWW page ...'. Moreover this process of augmentation could be carried to several *layer*s: one author augmenting another's augmentation — though in some applications this could lead to corresponding layers of knotty copyright/ethical issues.

The general point that these examples bring out is that if mark-up is specially designed for one sort of content, PostScript, say, then it can take advantage of its knowledge of the logical nature of that content, and refer to the content in terms of its logical objects using some form of calculated mark-up.

Finally, as an added complication, the mark-up can itself contain extra material that is to be added to the content. Thus the mark-up may specify the name of a button that is to be added to the underlying content.

3 ADVANTAGES OF SEPARATE MARK-UP

The case in favour of separate mark-up is certainly not just an academic one. Some solid advantages are:

(1) Mark-up can be applied to a content file (we will use the term 'file' though in fact there are many other possibilities, e.g. an item in a database) that the author cannot change. Thus the content could be stored on a CD-ROM or in someone else's file system.

(2) The mark-up does not change the content. Thus separate mark-up can be applied to a word processor document in a proprietary format, to an image or to a video. Adding embedded mark-up in these cases would make the content unusable for its intended purpose.

(3) Many alternative mark-ups can be built on top of the same content. An original exponent of this is George Landow [7], with every one of his English Literature students adding their own hypertext mark-up to act as a critique of some underlying content. This work originally exploited some of the fine generality of Intermedia [9], which supported separate mark-up.

(4) The content of the document can, within strict limits, change independently of the mark-up. This applies especially if a high-level form of calculated mark-up is used. Thus if the mark-up of a C program says 'make the heading of function Y the destination of a link', then the C program could change (provided that function Y remained) without upsetting the mark-up. Brailsford [1] quotes the problems with embedded mark-up in such cases. He was working with Acrobat mark-up added to journal pages. If mark-up is embedded into the content, and then a new version of the content is issued, it is a desperate job to extract the mark-up from the original content and re-embed it in the new content. Annoyingly, this tedious task is strictly needless if the mark-up relates to logical objects in the content that have not changed.

(5) Separate mark-up makes 'ubiquitous hypertext' possible. Instead of hypertext being a facility only available within a specialist piece of software called a hypertext system, hypertext facilities can be made ubiquitously available, in the same way that cut-and-paste is a ubiquitous feature in many environments. Thus in the middle of a word processing session it should be possible to call up a hypertext system that causes objects within the word processor document to act as links. If, when the word processor was originally written, its creators had no awareness that hypertext facilities would be added later, then clearly the hypertext mark-up must be separate. Microcosm, Hyper-G [5] and the PenPoint operating system [3] are three pioneers of the approach. (Indeed one of Microcosm's approaches is to build on the ubiquity of cut-and-paste.)

Interestingly many of these advantages apply to mark-up in general, not just to hypertext mark-up. For example, if one had some raw text on a CD-ROM it would be useful to be able to add separate mark-up that specified, e.g., titles, paragraphs, citations, etc., within it. This mark-up could then be used by a formatter.

BibTeX4 DISADVANTAGES

The compensating disadvantages are just as solid:

(a) ‘One is simple, two is complex’: clearly having *two* separate objects (mark-up and content) to manipulate and to keep in synchronisation with one another adds complexity. This is a manifestation of our first background point: the experts’ choice being the more difficult one.
(b) If the content is independently edited then the mark-up may become unusable, and this is not obvious at the time the edit is done. There are, however, certain palliatives to this. Firstly there could be an all-encompassing operating system that is aware of the relationships between mark-up and content files, and keeps them in synchronisation or at least gives warnings if they drift apart. Secondly, the dangers are less if calculated mark-up is used. Thirdly it may be possible, if all the parties work together, to use a version control system to control change. Lastly traditional mark-up can be abandoned in favour of the ‘generic’ mark-up found in Microcosm — an instance of what we have called dynamic calculated mark-up above. Unfortunately, however, none of these four palliatives comes close to providing a complete and general solution.
(c) As an extension to the previous point, if mark-up is embedded, anyone editing the content is always aware of the mark-up. If, for example, they add extra material they can decide where to put it relative to the boundaries the mark-up imposes, e.g. whether it is part of some emphasised text or beyond the end. This applies whatever editor is chosen. The advantage is lost if separate mark-up is used.

5 DOES IT MATTER?

The paper so far has presented a dichotomy between two opposing approaches. It is now time to change tack and consider whether, indeed, the approaches are as opposite as they seem. Can you combine the advantages of each?

The first point to make is that, to many users, the difference of approach is an irrelevant detail. How many users (as distinct from authors) of WWW, for example, know what its mark-up is like or whether it is stored separately from the content? In fact it is not. Even authors do not need to be aware of the difference: if authors do their work through a WYSIWYG interface they too are unaware of what the mark-up is like and whether it is embedded. The main constituency that remains are those people who (a) want to add mark-up to existing material that does not belong to them, or (b) want to use the marked-up material with a tool other than the one that created the material. These people need to know whether mark-up is embedded. They are currently a minority, though you could argue that if good facilities were provided for such people their numbers might grow.

6 BEST OF BOTH WORLDS?

We now move on to the question of whether, for those users to whom it *does* matter, the advantages of separate and embedded mark-up can be combined. This can be done by giving such users a choice, when they save a document they have created, of whether to make the mark-up separate. Moreover the user should be able to change their mind and convert from one form to the other, as the need arises. To be realistic the choice must be confined to a choice between embedded mark-up and separate positional mark-up. We must forego the advantages of calculated mark-up, because of the difficulty of generating such mark-up automatically from a lower-level form. To achieve our aim we propose two simple software components, a *Joiner* and a *Splitter*. The Joiner takes two input files and combines them into one, BIBTEX outputting the result. The Splitter is the complement of the Joiner, splitting an input file into two separate output files. Clearly the Splitter needs a *policy* to decide how the split is to be made. A policy of relevance here, when applied to an input file with embedded mark-up, is to output the mark-up in one file and the rest (i.e. the content) in the other, thus producing separate positional mark-up. Each different form of mark-up will need its own policy. The Joiner will likewise need rules for joining its two input files. A simple rule is to regard one input file as pure content and the other as a set of edits to the content. Thus the second file might be a set of edits to insert mark-up into the content. The policy of the Splitter is designed to fit with its complementary Joiner. Thus for the sample Joiner quoted above, the policy of the Splitter would be to extract the mark-up and output it as a set of edits to the content file. Indeed in this case the Joiner might be a simple old-fashioned text editor with commands of the form 'Insert ... at position ... '.

The Joiner and Splitter are conceptually general tools, but, since each form of mark-up will have its own policy, it will have its own version of a Joiner and Splitter. Thus a Splitter designed for SGML mark-up would not work for *troff* mark-up. Obviously the world would be a better place if all tools shared a common style of mark-up, but this is not so now, and probably never will be.

For each style of mark-up the Joiner and Splitter should match in the sense that if you split a file and joined it again you should end up with something identical, at least semantically, with the original.

The Joiner and Splitter should basically be simple, but there is a potential complication illustrated by our Background Point 3: problems associated with content being mistaken for mark-up. Assume that we have some material stored on a CD-ROM and want to add some mark-up to it. We use a Joiner so that the resultant output can be processed by a tool that understands embedded mark-up. What happens if the material on the CD-ROM already contains lines that look like mark-up? Such problems, though introducing tedious extra complexity are, however, fortunately solvable in practice. Ideally the Joiner and Splitter should be simple and fast enough to work on-the-fly, without any need for caching.

Our scheme above involves splitting into two separate files. A further element of complexity is introduced if we want to split into more than two separate files: this would be needed if there were several independent sets of mark-up on the same content, and/or if a single mark-up file brought together several content files. Rather than exploring this complexity, however, we propose a system of *layers*, whereby the output from a Joiner at a top layer can be used as one of the inputs to a Joiner at a lower level, and so on through as many layers as necessary. At the complementary splitting stage the Splitter would likewise feed its output to another Splitter, with a policy as to what is split off at each layer. Figure 1 shows how the joining might be organised.

BIBTEX

Figure 1: two layers of mark-up

We will now proceed to give some practical details.

7 IMPLEMENTATION WITHIN THE Guide HYPERTEXT SYSTEM

A Joiner and Splitter have been implemented within the UNIX version of the Guide hypertext system [2]. Guide uses embedded mark-up and the purpose of its Joiner and Splitter is to remove some of the disadvantages of this. From its inception, Guide has had facilities for generated mark-up: for example UNIX manual pages and other *troff* documents can be automatically converted into Guide form on the fly, with titles converted into buttons, 'SEE ALSO' items into links, etc. Hence the concept of having a program send its output into Guide is well provided for.

Guide users create hypertext facilities in a WYSIWYG way, and the authors (and users) are unaware of the underlying mark-up; the use of the Joiner and Splitter was therefore designed to be behind the scenes so that the authors and users can retain their blissful ignorance. All the author needs to do is to select an

BIBTEXoption, when it is needed, to save mark-up separately. Behind the scenes this involves Guide's Splitter, which has the policy of producing a separate (positional) mark-up file, which we call the *joiner-edit-file*.

The joiner-edit-file has a special comment line on the front, so that, if it is fed to Guide, Guide knows that it should call up its Joiner. The initial comment line of the joiner-edit-file also identifies the content file to be used (in fact there can be multiple content files, but we will suppress such detail here). The rest of the lines have the general form:

BIBTEX

```
Read N1 bytes from the content file
Insert the following: ...
Read N2 bytes from the content file
Insert the following: ...
```

(Thus the mark-up is, unfortunately, in terms of low-level byte offsets — something of a necessity, given that the content can be anything at all. If Guide were geared towards a special form of content, which had known structure, it might be possible to go a little way towards calculated mark-up.) The content can involve media other than text, but currently mark-up can only be inserted within the textual parts.

The Joiner is a crude editor that processes edits of the above form, and as a result converts back to the original form, with its embedded mark-up, and outputs this into Guide.

In fact the joiner-edit-file also contains a designator giving the policy by which it was created (we will soon see that several possible policies may be available). When a user comes to save a file, Guide by default saves it as embedded mark-up if the original input file used embedded mark-up and with separate (positional) mark-up otherwise. In the latter case the same policy is, by default, used as was found in the original input. Again, the ordinary user does not need to know whether she is using embedded or separate mark-up, but the user who cares can change the policy that is to be used on saving.

The joining and splitting described above apply both to files that are loaded when Guide starts up and to files loaded as a result of following links.

There are many cases when the user wants to go beyond the simple policy of just splitting off the mark-up. In our CD-ROM example, the user may wish to add content to the CD-ROM: extra text, button-names, icons, etc. Guide has a facility for the user to define logical objects in the source file. These might be objects such as Title, Part-number, Program-listing (when a piece of program is embedded in a document), etc. A separate table specifies for each logical object, how it is to be displayed (e.g. using X font and Y colours). These logical objects are, in fact, called *contexts*.

When saving a file, Guide uses the system of layers mentioned earlier. The default policy on saving is to have everything in the same layer (i.e. content with embedded mark-up), but other possible policies are to have a content layer and a mark-up layer above it, or to go a stage further and to have many separate layers. It is the contexts that provide the flexibility in assigning objects to layers, on top of a basic facility to specify whether mark-up is, by default, in a separate layer or not.

To provide this flexibility, each context has an attribute (changeable at any time by the user) which specifies what layer it and its content belong to. For example the user could create a context called *Annotation* which had the attribute that everything within an *Annotation* was saved at a mark-up layer, i.e. an *Annotation* is an object at the mark-up layer. Given this, the author can load one or more CD-ROM files, add some *Annotation*s (and perhaps some ordinary Guide buttons too) and save the result so that everything lying within an *Annotation* context plus the Guide mark-up go into a separate joiner-edit-file that can be applied to the CD-ROM content.

As a refinement, another context could be defined, called, say, *MetaAnnotation*, which allowed further annotations to be added and saved at a layer above the layer with the original annotations. We then have three layers:

(1) a content file.

BIBTEX (2) a set of *Annotations* and other mark-up on top of (1).
(3) a set of *MetaAnnotations* on top of (2).

Any of the above three can be used independently: i.e. the user can ask for just the content, the content with *Annotation*s, etc., or the content with both *Annotation*s and *MetaAnnotation*s.

Guide's Joiner and Splitter can be used in UNIX pipes when Guide files need to be communicated to other UNIX tools. As a refinement it is possible, in these cases, to adjust the Splitter's policy so that different logical objects can be selected. For example assume the user wants to extract the content layer, but minus any occurrences of a *Comment* context that occurs at this layer, but instead including all occurrences of the *Annotation* context, which normally belong at a higher layer. This can be done by temporarily adjusting the layer to which the *Comment* and *Annotation* contexts belong, and then doing a save, which passes the output to the next component in a pipe. (Indeed *Comment*s could be assigned to a null layer in this case.) This goes towards the kind of flexibility we mentioned when discussing literate programming.

8 LESSONS FROM THE Guide IMPLEMENTATION

The lessons from the Guide implementation are:

- the Joiner and Splitter can be used to combine many of the advantages of separate and embedded mark-up, though not the flexibility of calculated mark-up.
- it helps, however, if the machinations of the Joiner and the Splitter are disguised from the user. In particular it is an advantage that Guide automatically recognises whether an input file (a) involves embedded mark-up or (b) is a separate mark-up file, and in case (b) automatically invokes the Joiner. We call this *automatic mode recognition*. (If Guide's internal workings had been based on separate rather than embedded mark-up, its automatic mode recognition would have called the Splitter whenever it found an input file with embedded mark-up, i.e. a special tool is needed if the input style does not match what the hypertext system normally expects.)
- further flexibility is gained when the author can create, using a WYSIWYG interface, logical objects that relate to the Splitter's policy, e.g. *Annotation*s that belong in the mark-up layer.

9 PIPES AT WORK

We will now return to the general case and look at how marked-up files can be used by different systems, and how the Joiner and Splitter can help. Our discussion is based on a UNIX environment where tools (ideally simple uni-purpose ones) are connected together using pipes. The discussion applies equally well, however, to any operating system environment where tools can be connected together.

The simplest tool, as regards its input/output, is one where input goes in one end and output comes out the other. Most UNIX tools are designed like this, so that they can be fitted into pipes and more generally can use all UNIX's input/output redirection mechanisms. With such tools it is a trivial matter to incorporate Joiners and Splitters: the pipe mechanism is made for it.

With more complex tools the input file can link to other input files. We call this an *embedded-file-link*. Embedded-file-links occur extremely frequently in hypertext systems — indeed some people argue that they are the essence of hypertext — but they can also occur with many formatters (a document that contains a request to include another), programs (e.g. *#include* directives in C), etc.

BibTeX10 ANALYSIS OF EMBEDDED-FILE-LINKS

Since embedded-file-links are at the core of many hypertext systems we will discuss them in more detail. Let us assume that the user wishes to add mark-up to an existing file F, which is referenced by embedded-file-links. We will also assume that the added mark-up is for a hypertext system and that this hypertext system is based on embedded mark-up — this is likely to be the most common situation in practice.

Two properties of F are relevant:

(a) is F writable by the user?
(b) does the user require embedded-file-links that reference F to be interpreted as references to:
  (b1) the original F
  (b2) or to the marked-up version of F?

As examples of (b):

- the answer is likely to be (b1) if the reference to F is to include it in a C program. If the author has a separate mark-up file for F, to be used by a hypertext system, then that is irrelevant to C.
- the answer is likely to be (b2) if the reference to F is within a system that uses the extra mark-up. A particular example would be when the author has defined extra mark-up to be applied to an existing WWW file F; if another WWW page linked to F, the user should see the newly marked-up version of F: that is the whole point of the extra mark-up file.

(There may be cases where some embedded-file-links to F are to be interpreted as (b1) in some environments and as (b2) in others, but we will continue our policy of ignoring complications such as this in this paper.)

Case (b1) is easy. If, for example, a C includes a file called *mylib.h*, the author can create a separate mark-up file called, say, *mylib.mark*. When the user wants to view the file in its marked-up form he calls it *mylib.mark*. Ideally, if the hypertext system works on embedded mark-up, it will have an automatic mode recognition capability that automatically calls its Joiner behind the scenes to combine the two files *mylib.mark* and *mylib.h*. It is unlikely that there would be any existing hypertext links to *mylib.h*, but if there were they would lead to the original, unmarked-up, *mylib.h*.

Case (b2) is easy only when F is writable. When it is writable, the original file F can be moved to a new file FC, which will be treated as a content file, and a mark-up file, which applies to FC, can be put in place of F. Thus F is now a mark-up file, which leads indirectly to FC, the original content. If the hypertext system that deals with the mark-up has automatic mode recognition, then, on seeing any reference to F it will treat is as a mark-up file for FC.

The difficult case is case (b2) when F is not writable, because the trick of replacing F is not applicable. What is needed, either within the hypertext system or within the operating system it runs under, is a general aliasing system. In its simplest form this takes the form ‘use X in place of Y’, but an ideal, which is simple and more flexible, is to be able to say to the hypertext system ‘whenever you load a file X, call the program P (which can be specified by the user) with X as argument’. In the latter case the default version of P will simply use the file X unchanged, but P could be a Joiner with an associated table with entries of form ‘if the file is F, apply the Joiner to it using mark-up file FM, and output the result’. Moreover this approach opens the way to a generic form of annotation: program P could for example add an extra hypertext link on the front of *all* pages it loaded.

If none of the forms of aliasing that we have mentioned is available in practice, system hackers may be able to provide the equivalent by installing personal modifications to the I/O libraries that open files, particularly if such libraries are dynamically linked.

11 AN EXAMPLE WITH A SEPARATE MARK-UP HYPERTEXT SYSTEM

BIBTEXFinally, let us balance our emphasis on hypertext systems that have embedded mark-up with an example that assumes a hypertext system with separate mark-up.

Let us assume, therefore, that a hyperdocument is represented by one content file with separate mark-up.

We wish it to do some complicated edits on this, and decide to use a specialised editor, not the hypertext system, to do the job. Assuming we have a Joiner and Splitter that cater for the mark-up used in the hyperdocument, we apply the Joiner and send the resultant output, which is the hyperdocument with embedded mark-up, into the editor. We then do the necessary editing, and when we come to save the result, we send the editor's output to the Splitter, thus getting back to the separate form we had before.

A few points about this exercise:

- writing the necessary Joiner and Splitter can range from easy to hard depending on the nature of the mark-up.
- while we are using the editor the embedded mark-up will be evident — generally an advantage, given that the existence of this mark-up would not otherwise be apparent.
- in the final saved version the separate mark-up would automatically have been adjusted to take care of the change in the content, thus removing one of the problems associated with separate mark-up. There would, however, be problems if there were other separate mark-up files based on the original content.
- obviously the Joiner and Splitter are more complex if a single mark-up file can reference several different content files.
- the technique is just as applicable to, say, sound and video files as to text files. When editing a video file, for example, it might be useful to have some clue of the existence of mark-up on top of it.

12 CONCLUSION

The paper has set out to prove that the issue of separate versus embedded mark-up is not a central issue in hypertext. Most hypertext systems today are based on embedded mark-up, but they can be made to work with separate positional mark-up by using the Joiner and Splitter technology. Likewise systems based on separate positional mark-up can be made to support an embedded form if this is required, e.g. for exporting to some other tool that would like to see an embedded mark-up.

The big issue is not the nature of the mark-up but the facilities for integrating software components. In any real-world situation the hypertext system needs to work with other systems to provide a solution to a problem. The closer these systems fit together the better the solution. Our Joiner/Splitter technology requires a relatively small degree of integration. The real challenge is to proceed to a degree of integration where we never need to distinguish between documents that are hyperdocuments and those that are not, but where components work together to make hypertext ubiquitous, with any underlying tools to change the formats of the information being invisible to users.

ACKNOWLEDGEMENTS

I am grateful to the anonymous (?) referees, especially to Wendy, for helpful comments.

REFERENCES

1 D.F. Brailsford, 'CD-ROM Acrobat journals using networks', *Conference on Digital Media and Electronic Publishing*, Leeds Univ., 1994.

2 P.J. Brown, 'A hypertext system for UNIX', *Computing Systems*, **2**(1), 37-53, (1989).

BIBTEX3 R. Carr and P. Shafer, *The power of PenPoint*, Addison-Wesley, Reading, Mass., 1991.

4 H.C. Davis, W. Hall. I. Heath, G.J. Hill and R.J. Wilkins. 'Towards an integrated environment with open

hypermedia systems', *Proceedings of the ACM Conference on Hypertext: ECHT92*, ACM Press, 1992, 181-190.

5 F. Kappe, H. Maurer, and N.Sherbakov. 'Hyper-G - A universal hypermedia system', *Journal of Educational Multimedia and Hypermedia*, **2**, 1, 39-66, (1993).

6 D.E. Knuth, 'Literate programming', *Computer Journal*, **27**(2), 97-111, (1984).

7 G.P. Landow, *Hypertext: the convergence of contemporary critical theory and technology*, Johns Hopkins Press, Baltimore, Md., 1992.

8 S.R. Newcomb, N.A. Kipp, V.T Newcomb, 'The HyTime hypermedia/time-based document structuring language', *Comm. ACM*, **34**(11), 67-83, (1991).

9 N. Yankelovich, N. Meyrowitz and A. van Dam, 'Reading and writing the electronic book', *IEEE Computer*, **18**(10), 15-30, (1985).

BIBTEX