A link-oriented comparison of hyperdocuments and programs

Heather Brown, Peter Brown, Les Carr, Wendy Hall, Wendy Milne, Luc Moreau
Department of Computer Science, University of Exeter, Exeter EX4 4PT, UK &
Department of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
{H.Brown,P.J.Brown,W.Milne}@exeter.ac.uk &
{L.A.Carr,W.Hall,L.Moreau}@ecs.soton.ac.uk

Abstract

There are parallels between the construction of programs and the construction of hypertexts, and in particular between the abstractions available to the application programmer and those available to the hypertext author. In this paper we look at the distinctive element of the hypertext medium, the link, and discuss its possible programming language analogs. We go on to examine programming language abstractions that could be usefully employed by hypertext authors to control the complexity of the systems which they are engaged in building.

A link-oriented comparison of hyperdocuments and programs

Abstract

There are parallels between the construction of programs and the construction of hypertexts, and in particular between the abstractions available to the application programmer and those available to the hypertext author. In this paper we look at the distinctive element of the hypertext medium, the link, and discuss its possible programming language analogs. We go on to examine programming language abstractions that could be usefully employed by hypertext authors to control the complexity of the systems which they are engaged in building.

1. Introduction, background and assumptions

Following Dijkstra's famous article ‘Goto considered harmful’ [8], written in 1968, the goto statement has been deprecated in most programming languages. Remaining at the lowest abstraction levels, such as assembly languages, or used for efficiency reasons, it is masked by higher-level abstractions (e.g. selections, procedure calls, method invocations and continuations). On the other hand, the hypertext link, which has been characterized as a goto [7], has remained in use in hypertext. Indeed many people see it as the essence of hypertext; most of the definitions of hypertext originally given by Nielsen [16] centre around linking.

Although programming involves many variations on the goto which are safer for programming-in-the-small (if/then, case, iterations) and -in-the-large (procedures, modules, class libraries), the simple link remains the principal tool for the hypertext engineer. The complexity inherent in the unconstrained use of this primitive construct is one of the main issues in hypertext design.

This paper tries to analyse the apparent clash of practice. It also covers a wider issue: many authors have extended the goto/link analogy by likening the authorship of a hyperdocument to the task of writing programs, or, at a higher level, have mapped out a discipline of hypermedia engineering to match software engineering [13]. These comparisons can be valuable because hyperdocument authoring is a young discipline compared with the discipline of producing programs; if, by drawing parallels with programming, we can gain new insights into hyperdocument authoring, there are big potential gains. We must ensure, however, that the parallels are valid ones, and this paper tries to help. Our main focus, reflected in the title, is at the comparatively low level of links, rather than the higher levels of structuring and engineering.

1.1 Assumptions

In order to make this paper simpler, we shall fix some of the objects we are talking about. We shall assume that the hypertext is represented in HTML and viewed on a web browser. An HTML document may host scripting components, applets and various kinds of dynamic event handlers, but if we discuss hyperdocuments that involve bits of program this will inevitably muddy our discussion. Thus we shall confine our discussion to static hypertext: no pieces of Java, no CGI scripts, no cookies, etc. Our HTML hyperdocument will, of course, consist of a number of pages, and these will in general link to pages outside the current hyperdocument.

Notwithstanding our use of HTML as a basis for example, we will refer to hypertext systems other than the World Wide Web, since many of these are more developed in the abstractions they provide. In particular HTML essentially only offers one type of link, though there is a potentially extremely rich set of link types that hypertext systems may offer [4,14].

1.2 The author and the end-user

If we start on the programming side, the two important parties are the author(s) and the end-user(s). The author creates the program, which may be a module in a much larger program, and the end-user executes the program. The author's world is a long way away from the end-user's. Indeed the end-user is normally unaware of the nature of the source code, and whether it contains any gotos. The goto concept, and indeed all concepts of program structuring, just apply to the author's source code world.

In the hypertext world, the author prepares a document. The end-user reads the document the author has built. However the end-user's world is much closer to the author's than is the case for a program. In particular the links provided by the author are directly visible to the end-user. Thus, reflecting these two levels, we can draw two comparisons:

(a) between gotos in programs and the complete set of all links specified by hypertext *authors*, or

(b) between gotos in programs and the actual set of links used during a browser session by hypertext *end-users*.

We believe that (a) is the closer comparison, but (b) still deserves attention since the goto is principally happening to the end user. The text does not ‘go’ anywhere, instead there is an intuitive understanding that the user has ‘travelled’, hence the common reference to ‘navigating’ or ‘surfing’ the Web. The role of the author is to specify the complete set of gotos, i.e. to determine the possible navigational choices from which the user chooses the actual set.

2. Alternatives to the goto model

Our first point is that if one wishes to liken a hypertext link to a programming language concept, there are several alternatives to the goto model. We will discuss three of them here: we will call them the *link-is-data-reference* model, the *link-is-procedure-call* model and the *link-is-a-continuation* model.

The first model, the link-is-data-reference model, is simple. In this, a hyperdocument is likened to the data part of a program, not the executable part. Each hyperdocument page is likened to a particular data structure (or to an object in OO technology), and links are just references to other data structures. As an example, if the hypertext page just consists of some text T1, followed by a link L, followed by further text T2, then, using Java notion together with a very simple document object model, this is likened to the programming language data structure:

```
final String T1 = "If you are interested, please";
final Link L = new Link("click here", "http://site.org/data.html");
final String T2 = "for more information";
final Page P = new Page(T1, L, T2);
```

This is a simplistic view of a Web page, the Web Consortium’s DOM standard [19] is a more complete mechanism for treating a hypertext page as just such a simple data structure. The effect of this from our point of view is to reduce a link to an undistinguished component of the data structure, and requires the navigation behaviour (and the rendering activity) to be specified by external semantics in the form of stylesheet data or scripted functions.

Our second model, the link-is-procedure-call model, is closer to the goto model in that it relates to the executable part of a program. Although a Web server is stateless and is therefore unaffected by the user’s choice of link navigation, each browser maintains both a linear history and a stack of previously-visted pages together with a Back button. Given a *Back* facility, a link is arguably a specification of a procedure call, not a goto at all. A model, reflecting the analogy that each hyperdocument page is a procedure that potentially calls other procedures, is that a page can be likened to a procedure with the following body:

```
public static void invoke(String url){
       boolean mustExit=false;
       while(!mustExit){
               renderThisWebPage(url);
```

```
            selectedLink=getLinkChoice();
            if(selectedLink==back)mustExit=true;
            else invoke(selectedLink.getURL());
            }
    }
```

(The use of the *while* loop ensures that if a procedure representing another page is called, and this subsequently returns — as it will if the user selects *Back* — then the original page's procedure is re-executed.)

Obviously this model needs elaboration to cover special cases, one of which is links within a page. (Splitting the procedure for the page into subprocedures may best cover within-page anchors.) Nevertheless we trust that the model captures the essence of the procedural analogy, and shows that a link can be modelled as a procedure call. The model also covers some hypertext systems that have richer types of link properties than HTML (e.g. different data types of links, authorship properties attached to links), since these extras can be added as arguments to the procedure call.

Additionally in some hypertext systems, such as Hyper-G [10] and Microcosm [3], links are two-way. Two-way links provide a rich static mechanism that complements the dynamic facility provided by *Back*. However, given our keep-it-basic approach, we will confine ourselves to simple, one-way, links.

However, the real problem with links-as-procedure-calls is that browsers have “Forward” buttons too, so that ‘returning’ from a Web page is not completely akin to exiting a procedure as the procedure may be re-entered with its state restored. Further complications arise from the browser ability to clone an existing window, thus potentially duplicating the procedure’s activation record in a different context. To resolve this we present a third model, the *link-is-a-continuation* model.

The third model is best introduced via a sample application. The role of links in the presence of a browser forward button can be seen in the example of an educational CD-ROM [17] which we discuss here to show how an extension to the concept of a procedure call can successfully model arbitrary use of forward and back buttons as well as the history list. The CD-ROM contains software, embedded in an HTTP server, which manages complex guided tours of pages, menu choices and arbitrary computations that depend on users' runtime preferences. Consequently, with each page, there is a "computation state" which leads to the selection of the page, and which will also be used to select the next page in the tour.

Let us consider that the user jumps back to a previous page in the history list. The associated server "computation state" should be restored to what it was when the user first visited the page, so that the next page in the guided tour may be selected appropriately. Queinnec observes that the notion of *continuation* exactly embodies the computation state that has to be restored. Continuations are first-class objects in the language Scheme and represent the “rest of the computation”. At any moment, the current continuation, i.e. the rest of the operations remaining to be done, may be captured and stored away in the heap. Vice-versa, a continuation, which stored in the heap, may be re-instated. Queinnec shows that for each document served by the CD-ROM server, there exists an associated continuation representing the current server computation state. When the user navigates the history list, the associated continuations are re-instated in the server. In the terms of the hypertext, the continuation provides both a specification of the page to be viewed and also a ‘history so-far’ of the navigation up to that page.

Can these alternatives to the goto model reverse our thinking about links? In contrast to gotos, the use of procedure calls, continuations and data references are regarded as good programming practise; does this reflect directly on linking? Perhaps instead it shows the weakness of superficial analogies. In particular, to look at the analogies more carefully, we need to examine dynamic nature of programs and hypertexts.

3. Dynamic aspects of programs and in hypertexts

Programs are static entities, commonly expressed as a text, which specify a dynamic execution process. Whatever the programming paradigm, execution may be modelled by a trajectory in an execution space. Structuring is indeed the focus of Dijkstra’s original paper. To quote:

> " ... our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

Thus the argument is that a vital aim in the design of programming languages is to make the program easier to understand and to reason about, particularly in its dynamic behaviour.

On the hypertext side, hyperdocuments are also static, but the user's navigation by link traversal entails a dynamic change of state. In practical terms, this change of state occurs both in the browser (for instance change of history list), or in the server (for instance a change in the "computation state"). Navigation can be paralleled to a trajectory in a hyper-space. The author of a static hyperdocument has also the intellectual burden of visualizing the navigation in the hyper space. There are at least four burdens on the hypertext author caused by the connectivity of the material.

(1) structuring the material in the first place. Correctly structuring a program may be facilitated by a design model and by the underlying abstractions of the programming language. Successfully structuring a hyperdocument is also helped by applying a suitable design model [9, 10, 16], however the unenhanced link remains the underlying abstraction available to the author. This is indeed a challenge (see the evidence collected in Nielsen's book), but is not our focus here.

(2) making the document coherent over all possible paths,

(3) ensuring there are no dangling links, or, the other side of the same coin, no material that is unreachable by a link,

(4) providing navigation alternatives for the end-user.

We discuss (2), (3) and (4) in turn in the three subsections below.

3.1 Coherence and consistency

Item (2) above provides a large intellectual burden on the author. *To write the content of a page, the author potentially needs to know all the possible paths by which an end-user may reach that page.* This is trivial in a catalogue or encyclopaedia, since each page is authored to be self-contained to be read independently of all the other pages. The burden is significant in a tutorial document: for example if concept C2 builds on concept C1 then C1 is a prerequisite for C2, and all the possible paths to a page about C2 should pass though a page about C1 first. (Of course, the author cannot stop the end-user jumping directly to a page about C2 by means of a bookmark or a "find" facility.)

A good design strategy is to produce a “rhetoric of arrival and departure” [11] for each significant page, which must be honoured by all the pages that link to the current page. Thus in general, to make a hyperdocument coherent, the author has to visualize the connectivity of the whole hyperdocument. An obvious consequence of this is that “spaghetti linking” (lots of links with no structure to them) increases the author's burden, especially without tools to assist the process [18].

Research into programming languages has led to the definition of constructs that control the scope of names and data access (blocks, procedures or modules). On the other hand, hyperdocuments have one scope: the page. This presents a significant burden to the author who needs to be aware of all the pages that may link to the current one, either directly or via other pages, and spoils the analogy that a link is a clean procedure call, or, for that matter, a simple data reference.

A similar situation exists in the SGML world, where the primitive linking facilities provided (IDs and IDREFs) also give a single level of scope, but one that is strictly enforced as no reference can be made to IDs in other documents. HyTime added this capability, but further made it possible to reference any item of data (with or without IDs) in any kind of document (encoded in SGML or not), thus overriding and destroying the concept of scope as an authorial tool.

Turning from the author, the cognitive load on the end-user is remembering what decisions were made at previous pages visited, i.e. which link was chosen, and remembering what other information and links were on the page. (There is also the longer-term problem of remembering, perhaps several days later, where information was seen, but this is not our concern here.)

3.2 Dangling links

For regular web users, probably the most obvious failure in hyperdocument authorship is the “dangling link”, the link that tries to reference a non-existent URL. Obviously therefore there is a burden on authors to get all their links right, and this burden proves too great for many authors. The problem can occur either (a) because the author got the link wrong in the first place, or (b) because the link destination has subsequently been changed. Corresponding failures when running programs are comparatively rare, though problems of type (b) can arise in those programming languages that support dynamic casts.

The early history of programming language development made the transition from explicitly enumerated machine addresses to symbolic names, with the rules controlling the resolution of names enshrined in the definition of the programming language’s semantics. In the hyperdocument world, of course, the link is defined in terms of a URL that is a machine address (quite literally). Attempts to introduce symbolic names (URNs) have so far not succeeded, and in fact the reverse is true: in situations where symbolic names are required (for example, to define XML namespaces) a hypothetical machine address is invented. As symbolic names are normally used in programming, there is inevitably a linkage process or link resolution activity in which the identity of the data being named is established.

In the programming world thorough checking is the norm, and programs are continually checked both throughout their development and throughout their usage. There have been great advances in mechanisms to allow authors to detect errors before programs are released to end-users. Some of these mechanisms, like test harnesses, simply involve simulating user behaviour; the more interesting mechanisms concern programming language features that make certain types of error impossible. One of these is strong type checking: this allows a compiler to detect certain types of error, and, if they occur, prevent a program from being run. A second is the concept of strictly defined module interfaces, which allow linkage editors to detect errors when modules are fitted together. A third is information hiding, whereby the author can control who sees what.

The culture in hypertext authoring, on the other hand, is still towards the do-what-you-like end of the spectrum. Thus authors do not generally use some equivalent of a linkage editor, which could detect dangling links within a hyperdocument, or — the converse — pages that could never be reached. Nor do authors provide the equivalent of *imports* and *exports* declarations, which specify which outside links are assumed, and which outside services the hyperdocument provides. It would be better if they did. Finally, it would help if hypertext authoring systems provided for information hiding, if only at the page level (“This page is public and anyone can link to it; this other page is an internal one that might well be changed, and thus other authors should not link to it”).

On a wider scene, these extra mechanisms might be extended to cover user interfaces in general, rather than just the restricted form of interface provided by hypertext linking.

3.3 Navigation Alternatives

A facility that is radically different from programming languages is the concept, present in Microcosm, Hyper-G and HyTime, that the link structure should be separate from the document that it applies to. Indeed several separate and independent link structures could be applied to the same document, each author thereby providing an interpretation of the structure of the underlying document. Moreover the separate links might be what DeRose calls "Intensional" links [4], such as a generic link from every occurrence of a certain word — wherever it may occur — to a dictionary entry that explains that word.

4. How connectivity has been tamed in programming languages

Our conclusions so far are that there are, not surprisingly, big burdens on the hyperdocument author caused by connectivity. Hence we can gain by looking at the programming mechanisms that help tame connectivity. We will now do this, and we will then relate these mechanisms to hyperdocument authors and end-users.

4.1 Aggregation of choices

Aggregation of choices in programs is provided by the *if-then-else* statement, and a generalisation of this, the *case* or *switch* statement. This represents the abstraction of "choose one from many". It allows nesting and thus a tree-structured choice. Aggregation of links is common in hypertext systems outside the Web either in the form of the "choose-one-from-many" pattern found in programming languages, or in the form of the trail, a sequence of links followed in turn. In the former case, the aggregation can simply be a matter of recommended style [12], perhaps using HTML's list as an existing data aggregation mechanism; alternatively in HyTime [15] and the seminal Intermedia hypermedia system [18], aggregations of links ("fat", one-to-many links) are a built-in feature. In the latter case, the trail is provided by the author to guide the end-user through a predetermined sequence of pages, designed to give the end-user an understanding of a certain subset of the information represented by the overall hyperdocument. Usually there can be any number of separate trails through a hyperdocument, and these are totally independent of each other: for example they may cross and overlap one another in arbitrary ways. The path concept can be carried a stage further by allowing choices within a path, and more generally by providing a script that takes paths though a document and performs various actions as it goes [21]. It is not clear that this abstraction is found in programming languages.

4.2 Assert statements

Checking that a property holds at run time is an important feature both in programming languages and hypertext. Although assert statements in programming languages are normally used for dynamic states, they can also be used in conjunction with the staticly-determined states in hyperdocuments. The author could use a similar facility to verify that a condition holds for the user's navigation. To return to our example of concept C2 depending on concept C1, the page for concept C2 might say: `<assert> C1-covered</assert>` where `C1-covered` is a Boolean variable set by all pages that cover C1. Thus one such page may say: `<set> C1-covered </set>` (Obviously `<set>` and `<assert>` are concerned with checking, and do not cause anything to appear on the screen: they are therefore likely to be coded along with other metadata in the HEAD section on the hypertext page.) Given these declarations, a checker, run every time the hyperdocument is modified, can ensure that no path through the hyperdocument can reach the `<assert>` statement for `C0-covered` without passing through a corresponding `<set>` statement. A mechanism that could be used for this facility exists in the form of cookies (named values that can stored in the browser), but this is at too low a level to be immediately useful to the hypertext author.

4.3 Pre- and Post-Conditions

Java's `finally` clause guarantees that a post-action will be executed whenever control leaves a block for any reason (natural termination, a return statement, breaking out of a loop or exception throwing). More generally, Scheme's `dynamic-wind` provides both pre- and post-actions for a block. Such a mechanism could be used to ensure that a particular condition holds (see previous subsection) or to enforce the arrival and departure paradigm (see section 3.1) when linking into and out of the body of a hyperdocument [6].

4.4 Procedures

Procedures have been described above as models for the behaviour of a link. They also stand as useful metaphors for hypertext construction: the seminal Guide system [2] encouraged users to think of navigation in terms of embedded, nested pages which were opened (unfolded with contents visible) or closed (folded with contents hidden), corresponding very well to the activation of a procedure.

4.5 Exception Handling

The ability to handle exceptional error situations has become an important feature of programming languages. Several HTML constructs (for example the `<OBJECT>` and `<FRAMESET>` elements), can declare alternative hypertext page fragments to present to the user if the browser cannot correctly process the required elements. Similarly, Web servers and proxies can be configured to present alternative information to the user if the requested page cannot be delivered.

Most hypertext servers, when unable to resolve a URL, just present a page that contains an error message, and leave it to the user to decide what to do, i.e. the server provides its own default exception mechanism. It would be of more benefit to expose the exception handling mechanism to the hypertext author to allow better control, especially over links to remote pages over which the author has no control.

4.6 Interfaces

Interfaces (made popular through Java) have the potential to provide a useful abstraction that can be applied to a set of hyperdocument pages in the same way that a Java interface provides a useful abstraction that captures the key features of a set of data objects. By way of example, assume that author X has produced a hyperdocument that discusses geometric shapes. X recognises that different end-users, who will have different displays, will want to view these shapes in different ways. Hence, following programming practice, X decides that all the hypertext pages concerned with displaying shapes should be in a separate module. X provides one possible instantiation of this module, but wants to allow other authors to provide other ones. In order to aid this X would like to provide an interface specification, which guarantees that any module that discusses geometric shapes and gives similar kinds of information would be an acceptable replacement. Any set of Web pages (perhaps discovered by a search engine) which contains an explanation about squares, circles and triangles and taught the user how to find the perimeter and area of each shape irrespective of the tuition method or rendering technology may be an equivalent module as far as the purpose of this hyperdocument is concerned. Interfaces model an expected set of facilities and may be about the data or the links that are to be used across the boundary of the hyperdocument.

5. Concluding comments

Programming can at many points be usefully compared to hyperdocument authorship. Both program execution and hyperdocument navigation are concerned with dynamic state. However the connectivity structure is visible to the end-user of a hyperdocument, but not to the end-user of a program. It is often the hypertext author’s responsibility to add extra connectivity to aid user navigation, whereas a programmer’s aim might be to minimize connectivity in order to reduce complexity or to decrease the code’s ‘footprint’.

The hyperdocument author needs to be aware of the incoming links to each page, whereas the programmer need not be aware of the context in which a procedure or module is to be used. Further, hyperdocuments normally have links to external hyperdocuments over which the author has no control and for which no checking or validation is performed.

Some of the abstractions in hypertext have no obvious parallels in programming: for example we have discussed trails and separating link structure from hyperdocuments. Hence likening hypertext links to programming constructs, if done superficially, is unlikely to yield valid insights. To return to the is-a-link-a-goto question, the answer is that it is an analogy that is half right, half wrong. Analogies of gotos with procedure calls or data references are just as right — and just as wrong.

It is better to look at the overall concept of connectivity, which produces similar issues in programming and in hyperdocuments. The development of programming has been dominated by introducing more disciplined ways of working. Some disciplines that can help hypermedia are:

(a) assertions, pre- and post-conditions

(b) linkage editors to check connections

(c) exception handling *and*

(d) interfaces to hyperdocuments.

Overall, an important requirement of hyperdocument authors is for new mechanisms in the authoring language or environment that are designed to prevent linking errors. The XLink proposal [5] provides a suitable container architecture for expressing some of the abstractions listed in this paper; further work is planned to build authoring support systems on this basis.

Acknowledgements

Peter Brown would like to thank the Leverhulme Trust for support.

References

1. Brown, P.J. “Do we need maps to navigate round hypertext systems?”, *EP-odd*, **2**, *2*, pp. 91-100, 1989.

2. Brown, P.J. “Dynamic Documentation”, Software Practise and Experience, 16, 3, (March 1986), 291-299.

3. Davis, H.C., Hall, W., Heath, I., Hill, G.J. and Wilkins, R.J. `Towards an integrated environment with open hypermedia systems', *Proceedings of the ACM Conference on Hypertext: ECHT92*, ACM Press, pp. 181-190, 1992.

4. DeRose, S.J. “Expanding the notion of links”, *Hypertext'89 Proceedings*, ACM Press, pp. 249-257, 1989.

5. DeRose, S.J. “XML Linking Language (XLink)”, W3C Working Draft 21-February-2000, http://www.w3.org/TR/xlink/ . 2000.

6. De Roure D. “The Role of Distributed Lisp” in Open Hypermedia Information Systems, In *Proceedings of Parallel Symbolic Languages and Systems*, 1996.

7. De Young, L. “Linking considered harmful”, Hypertext: Concepts, Systems and Applications, *Proceedings of the European Conference on Hypertext*, INRIA, France, Cambridge University Press, pp. 238-249, 1990.

8. Dijkstra, E.W. `Goto considered harmful', *Comm. ACM*, **11**(3), pp. 147-8, 1968.

9. Garzotto, F., Paolini, P. and Schwabe, D. “HDM – A Model-based approach to Hypermedia Application design”, *ACM Transactions on Information Systems*, **11**(1), 1-26, 1993.

10. Kappe, F., Maurer, H., and Sherbakov, N. `Hyper-G: a universal hypermedia system', *Journal of Educational Multimedia and Hypermedia*, **2**, 1, pp. 39-66, 1993.

11. Landow, G.P., `The rhetoric of hypertext: some rules for authors', *Journal of Computing in Higher Education*, **1**, 1, pp. 39-64, 1989.

12. Lemay, L., *Teach yourself web publishing in a week*, Sams Publishing, Indianapolis, Third Edition, 1996.

13. Lowe, D. and Hall, W., *Hypermedia & the web: an engineering approach*, John Wiley, Chichester, 1999.

14. Moreau, L. and Hall, W. “On the expressiveness of links in hypertext systems”, *Computer Journal*, **41**, 7, pp. 459-473, 1998.

15. Newcomb, S.R., Kipp, N.A. and Newcomb, V.T., “The HyTime hypermedia/time-based document structuring language”, *Comm. ACM*, **34**(11), pp. 67-83, 1991.

16. Nielsen, J. *Multimedia and hypermedia: the internet and beyond*, Academic Press, San Diego, Ca., 1995.

17. Christian Queinnec, “The Influence of Browsers on Evaluators.” University Paris 6 — Pierre et Marie Curie. Submitted for publication, 2000.

18. Thimbleby, H. *Int. J. Human-Computer Studies* (1997) 47, 139-168, http://ijhcs.open.ac.uk/thimbleby/thimbleby-01.html

19. Wood, L. Level 1 Document Object Model Specification, W3C Recommendation, http://www.w3.org/TR/WD-DOM/

20. Yankelovich, N., Meyrowitz, N. and van Dam, A., “Reading and writing the electronic book”, *IEEE Computer*, **18**, 10, pp. 15-30, 1985.

21. Zellweger, P.T. `Active paths through multi-media documents', in van Vliet (Ed.), *Document manipulation and typography*, Cambridge University Press, pp. 1-18, 1988.