Are hyperdocuments like programs and are links really gotos?

Heather Brown, Peter Brown, Les Carr, Wendy Hall, Wendy Milne,
Luc Moreau

Department of Computer Science, University of Exeter, Exeter EX4 4PT, UK
and
Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK

{H.Brown,P.J.Brown,W.Milne}@exeter.ac.uk
and
{L.A.Carr,W.Hall,L.A.V.Moreau}@ecs.soton.ac.uk

ABSTRACT

Todo

1. Introduction, background and assumptions

Following Dijkstra's famous article '*Goto considered harmful*'[4], written in 1968, the goto statement has been deprecated in most programming languages, and only remains at the lowest abstraction levels, such as assembly languages, which are used increasingly rarely. On the other hand the link, which many authors have characterized as a goto [3 + other refs todo], has prospered in hypertext. Indeed many people see it as the essence of hypertext; for example most of the definitions of hypertext cited by Nielsen [12] centre round linking.

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 [9]. 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 the programming language is a simple imperative one, and we shall assume the hypertext is represented in HTML, and viewed on a web browser.

Nowadays, "dynamic HTML" is a focus of attention, 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, just as a program module will interact with other modules.

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 [5].

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 (gotos) provided by the author are directly visible to the end-user. Thus, reflecting these two levels, we can draw parallels between:

(a) gotos in programs and links used by hypertext *authors*.

(b) gotos in programs and links used by hypertext *end-users*.

We believe that (a) is the closer parallel, but (b) still deserves some attention.

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 two of them here: we will call them the *link-is-data-reference* model and the *link-is-procedure-call* model. *<<add a third: continuations?; link-is-a-variable?>>*

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, this is likened to the programming language data structure:

```
final String T1 = "...";
final Link L = new Link(...);
final String T2 = "...";
```

<<Think about this more: is it a problem that hypertext has the notion of sequential reading interspersed with jumps, and this is much closer to program than data?>>

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. Its motivation is that almost every hyperdocument browser provides a *Back* facility. (Additionally in some hypertext systems, such as Hyper-G [6] and Microcosm [2], all 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.) Given a *Back* facility, a link is arguably 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:

```
boolean mustExit = false;
while (!mustExit) {
  Call the browser, with the HTML of the
  current page as parameter. We assume the
  browser displays the page, and, when the
  end-user has selected the next action, re-
  turns either a URL or an indication to go
  Back.
  if (back was selected)
    mustExit = true;
  else
    Call the procedure corresponding to the
    URL
}
```

(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. (Within-page anchors may best be covered by splitting the procedure for the page into subprocedures.) 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.

<<TODO: more fundamental material related to Moreau and Hall [10].>>

Can these two alternatives to the goto model reverse our thinking about links? Since both the use of procedure calls and the use of data references, unlike the use of gotos, are regarded as good in programming languages, does this suddenly making linking good too? The answer is more that it shows the weakness of superficial analogies. In particular, to look at the parallels more carefully, we need to examine the notion of state.

3. State in programs and in hypertext

In a imperative program state is represented *inter alia* by the values of all the current variables. The dynamic changing of state is a key property of any program, and this indeed is 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, with our assumption of static hypertext, there is no concept of dynamic state. The author of a hyperdocument therefore does not have the huge intellectual burden of, to use Dijkstra's words, "visualizing processes evolving in time". There are, however, at least four burdens on the hypertext author caused by the connectivity of the material:

(1) structuring the material in the first place. This is indeed a challenge (see the evidence collected in Nielsen's book [12]), but is not our focus here.

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

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

(4) providing navigation for the end-user. This is needed because, unlike with programs, the end-user sees the structure prepared by the author.

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

4. Making hyperdocuments coherent and consistent

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 burden is trivial in a document such as a catalogue, where each page is self-contained and can be read independently of all the other pages. The burden is, however, significant in, say, a tutorial document: for example if to understand concept C1 you need to understand concept C0, then all paths to a page about C1 should have passed though a page about C0 first. (If the end-user, by means of a bookmark, a "find" facility, or whatever, jumps directly to a page about C1, then that is his risk, and not something the author can control.) More generally, a good author will probably produce a "rhetoric of arrival" [7] for each page, and this must relate to 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 hyperdocument. An obvious consequence of this is that "spaghetti linking" (lots of links with no structure to them) increases the author's burden. It also relates to our analogy of a link as a procedure call. In good programming practice, procedures are largely self-contained, and the use of global variables, etc., is deprecated. In hyperdocuments, on the other hand, the author needs to be aware of all the pages that may link to the current one, either directly or via other pages. This is not quite as bad as global variables used in procedures in programs, since it relates to the static structure of the hyperdocument rather than its dynamic structure. Nevertheless it is a significant burden, and spoils the analogy that a link is a nice procedure call, or, for that matter, a simple data reference.

<<Todo: redo the rest of this Section:>> As we have said, links are also visible to the hypertext end-user, and we also need to look at the cognitive load here. The user has a simpler problem than the author in that, following the model of most WWW browsers, she has a stack of current links.

If a link is regarded as a procedure call, this stack gives the currently open procedure calls. A use of the *Back* command returns to the calling procedure. In Netscape — and similar features are found in other browsers — *Back* is complemented by a more general *Go* command, which can return to a procedure call higher up the stack: this is equivalent to doing a sequence of several procedure returns.

The cognitive load on the end-user is remembering what decisions where 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.) *<<The paper needs more work on the end-user's view of links.>>*

5. 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 selection of procedures, e.g. dynamic casts.

We do not, however, believe that this difference between programs and hyperdocuments is fundamental: instead it is largely cultural. 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 way out 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.

6. Navigation and other features confined to hypertext

In order to show that analogies between linking and gotos (or other program constructs) need to be viewed with care, we will now discuss some extended link facilities that have no obvious parallel in programming languages.

The first relates to navigation. This is an authorship concept that is visible to end-users. If we assume the end-user sees a single page (i.e., in HTML terms, there are no frames), there must be a convenient and logically sensible path for the end-user to get from any page to any other. This concept has no parallel in programming languages, either in data or the executable code. <<*Todo: develop this further? Make point that navigation links may be semi-automated using templates*>>

A second facility that is radically different from current 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 [5], such as a generic link from every occurrence of a certain word — wherever it may occur — to a dictionary entry that explains that word.

7. 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. We will ignore programming constructs such as *do* and *while* statements, since these are geared to dynamic states. All the mechanisms relate to the executable part of a program rather than the data, and thus they relate best to the model of a link as a goto or procedure call.

Among them are:

(1) Aggregation of choices, as 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.

(2) Assert statements. These relate to states assumed by the author to apply, and may be checked at compile-time, or, more usually, at run-time.

(3) The procedure or method call. This is a highly-developed programming concept and procedures have a large number of desirable properties: one of these is that they have a well-defined entry point (the start) and exit point (the end, or perhaps a *return* statement).

(4) Exceptions. These relate to dynamic behaviour, and can relate to the states of variables — often these variables relate to some state of the computer, its devices, or the user's actions.

(5) Interfaces and information hiding.

8. Abstractions from hypertext links

<<This is a weak Section: more work and insights needed.>> The corresponding abstractions in hypertext are:

(1) Aggregation of choices. Aggregation of links is common in hypertext. It can (a) follow the "choose-one-from-many" pattern found in programming languages, or (b) it can be used to provide a concept unfamiliar in programming languages: the *path*, or sequence of links followed in turn. In case (a), the aggregation can simply be a matter of recommended style [8], perhaps using HTML's existing aggregation mechanism, the list; alternatively in some hypertext systems, such as HyTime [11] and the pioneering Intermedia hypermedia system[13], aggregates of links ("fat", one-to-many, links) are a built-in feature. In case (b), the path is provided by the author to guide the end-user through a coherent 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 paths 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 [14]. It is not clear that this abstraction relates to programming languages at all. <<TODO: structure separate from the document; *Generic links* and *Compute links*. Hierarchical structures with up-next-back links. All the material in this Section needs bringing together.>>

(2) Assert statements. 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. To return to our example of concept C1 depending on concept C0, the page for concept C1 might say:

```
<assert> C0-covered
</assert>
```

where `C0-covered` is a Boolean variable set by all pages that cover C0. Thus one such page may say:

```
<set> C0-covered
</set>
```

(Obviously `<set>` and `<assert>` are concerned with checking, and do not cause anything to appear on the screen: they therefore relate to a page header.) 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.

(3) The procedure or method call. This is most manifest, if one accepts the link-is-a-procedure-call model, in the *Back* button. *<<TODO: note and usage buttons in Guide; Nielsen's experiments; multiple entry points to a page.>>*

(4) Exceptions. Since exceptions represent highly specialised behaviour that relates

to the dynamic behaviour of programs, it is not surprising that no corresponding hypertext facility exists. (Perhaps the nearest equivalent is the HTML `<OBJECT>` tag, which can provide a series of fall-back positions if loading fails.) Paradoxically, a bugbear of the hyperdocument user is, as we have already said, the dangling link. This is just the sort of problem that exceptions cater for in programming languages. Currently in hypertext the browser, when unable to follow a link, just presents a page that contains an error message, and leaves it to the user to decide what to do, i.e. the browser provides its own default exception mechanism. It may be of benefit to allow the author to specify an exception mechanism that overrides the browser's default. This applies especially to outside links that the author has no control over.

(5) Interfaces and information hiding. These mechanisms have the potential to provide a useful abstraction that can be applied to a set of hyperdocument pages. It may help to give a concrete 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 supporting this interface will be acceptable.

9. Remarks about higher-level concepts and fundamentals

Our focus here has been on low-level programming concepts, but in looking at these we have become increasingly conscious of high level issues concerning the relationship of software engineering and hyperdocument design. From a separate perspective, we also believe that a more rigorous analysis of underlying concepts is needed. We plan to discuss these in separate papers.

10. Draft conclusions

Imperative programming is in many ways different from hyperdocument authorship. (Other styles of programming, such as functional programming are even further from current hyperdocument authorship, though there might be interesting future research in bringing them closer together.) Key differences are:

(1) programs are concerned with dynamic states, but hyperdocuments (in their basic form) are not.

(2) the connectivity structure is visible to the end-user of a hyperdocument, but not to the end-user of a program.

(3) Following on from (2), the hypertext author often needs to add extra connectivity, whereas a programmer's aim might be to minimize it.

(4) The hyperdocument author needs to be aware of the incoming links to each page.

(5) Hyperdocuments normally have links to outside hyperdocuments over which the author has no control.

(6) Some of the abstractions in hypertext have no obvious parallels in programming: for example we have discussed paths and separating link structure from documents.

Hence likening of 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) assert statements.

(b) linkage editors to check connections.

(c) exceptions; in the hyperdocument case this may apply to failing links.

(d) new mechanisms, designed to prevent errors, in the authoring language.

11. Acknowledgements

Several of our colleagues have contributed greatly to this work: we would especially like to mention todo. In addition Peter Brown would like to thank the Leverhulme Trust for support.

12. References

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

[2] 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.

[3] 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.

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

[5] DeRose, S.J. 'Expanding the notion of links', *Hypertext'89 Proceedings*, ACM Press, pp. 249-257, 1989.

[6] 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.

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

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

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

[10] Moreau, L. and Hall, W. 'On the expressiveness of links in hypertext systems', *Computer Journal*, **41**, 7, pp. 459-473, 1998.

[11] 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.

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

[13] Yankelovich, N., Meyrowitz, N. and van Dam, A., 'Reading and writing the electronic book', *IEEE Computer*, **18**, 10, pp. 15-30, 1985.

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