Design issues for a Dexter-based hypermedia system[†]

Kaj Grønbæk
Computer Science Department, Aarhus University, Ny Munkegade 116, DK 8000 Aarhus C, Denmark.

Randall H. Trigg
Xerox PARC
3333 Coyote Hill Rd., Palo Alto, CA 94304, USA

Abstract

This paper discusses experiences and lessons learned from the design of an open hypermedia system, one that integrates applications and data not "owned" by the hypermedia. The Dexter Hypertext Reference Model [8] was used as the basis for the design. Though our experiences were generally positive, we found the model constraining in certain ways and underdeveloped in others. For instance, Dexter argues against dangling links, but we found several situations where permitting and supporting dangling links was advisable. In Dexter, the data objects making up a component's contents are encapsulated in the component; in practice, references to objects stored apart from the hypermedia structure should be allowed. We elaborate Dexter's notion of composite component to include composites that "contain" other components and composites with structured contents, among others. The paper also includes a critique of Dexter's notion of link directionality, proposes a distinction between marked and unmarked anchors, and discusses anchoring within a composite.

1. Introduction

The hypermedia work discussed here is part of the DeVise project at the Computer Science Department, Aarhus University, Denmark [4]. The DeVise project is developing general tools to support experimental system development and cooperative design in a variety of application areas including large engineering projects. These use settings are characterized by group work distributed over time, space and hardware platforms. The requirements this intensely collaborative, open-ended work makes on hypermedia include: a shared database, access from multiple platforms, portability, extensibility and tailorability. For a detailed discussion of our engineering project use setting and its CSCW and hypermedia requirements, see [5].

To our knowledge, no hypermedia system met these requirements on the platforms we needed to support. Having to build our own, we nonetheless wanted to benefit from the experience and expertise of past and present hypermedia

[†] In Proceedings of the European Conference on Hypertext '92 (ECHT '92), Milano, Italy, December 1.-4. 1992

designers. It was for this reason that we decided to use the Dexter Hypertext Reference Model [8] (called “Dexter” in the rest of this paper) as our platform. Dexter is an attempt to capture the best design ideas from a group of “classic” hypertext systems, in a single overarching data and process model. Although these systems have differing design goals and address a variety of application areas, Dexter managed to combine and generalize many of their best features.

We took the Dexter reference model as the starting point and turned it into an object-oriented design and prototype implementation (called DeVise Hypermedia, or just "DHM"), running on the Apple Macintosh. As programming environment, we chose the Mjølner Beta System supporting the object oriented programming language BETA [12]. The Mjølner Beta System includes an object-oriented database [10], in which our hypermedia structures are stored.

Among the media supported by DHM are text, graphics, and video, using a styled text editor, a simple drawing editor, and a Quicktime movie player, respectively. DHM also supports link and node browser composites and a composite to capture screen configurations of open components (modelled on the NoteCards *TableTop* [18]). In addition to traversing links (including multi-headed ones), users can edit link endpoints using a graphical interface. Components can also be retrieved and presented via title search. Dexter's model of anchoring is extended to include a distinction between *marked* and *unmarked* anchors. Finally, in contrast to Dexter, DHM explicitly supports dangling links.

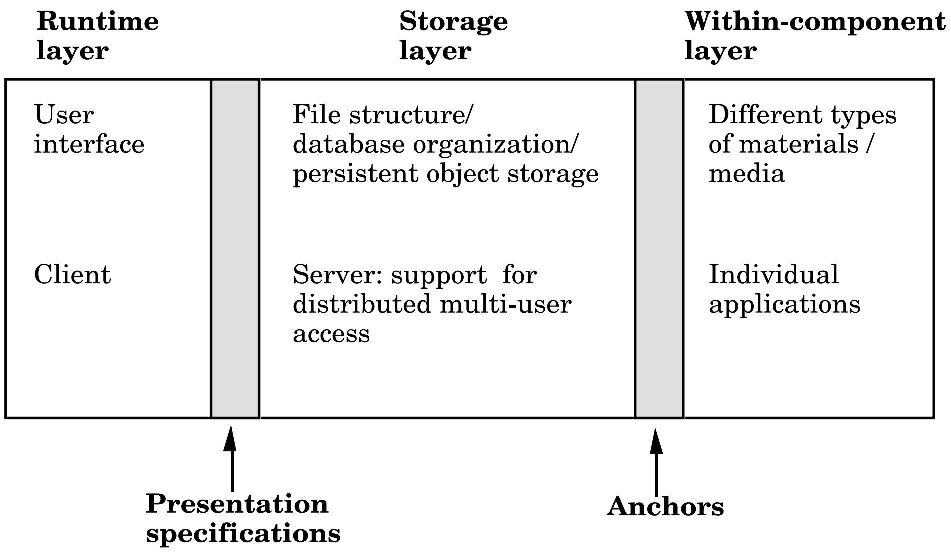
In short, our attempt to directly “implement” Dexter was largely successful. We were surprised at the robustness of the resulting design - it met several of our goals not explicitly identified in the Dexter paper. At the same time, we uncovered holes in the model, areas where further development is needed. For some of these, we now feel prepared to offer proposals for other hypermedia designers. Those involving the overall architecture, details of the data and process model, and tailorability are described in [3]

This paper reviews the Dexter model before discussing our experiences in applying it. Our focus here is on links, anchors, composites and cross-layer interfaces. For each of these, we comment on the utility and applicability of Dexter, identify the implementation choices made in our prototype, and make recommendations for designers of future systems and standards. We close with research issues and open questions.

2. The Dexter Model

The *Dexter Hypertext Reference Model* [8] separates a hypertext system into three layers having well defined interfaces as shown in Figure 1.
The *Storage layer* captures the persistent, storable objects making up the hypertext. The basic object provided in the Storage layer is the *component*. As shown in Figure 2, components are divided into *contents*, corresponding to the component’s data, and *component information*. The component information includes a general purpose set of *attributes*, a *presentation specification* and a set of *anchors*. The *atomic component* is an abstraction replacing the widely used but weakly defined concept of ‘nodes’ in a hypertext network. *Composite components* provide a hierarchical structuring mechanism. The contents of a *link component*

is a list of *specifiers*, each including a presentation specification as well as component and anchor identifiers. A *hypertext* is simply a set of components.



Runtime layer
Storage layer
Within-component layer
User interface
File structure/ database organization/ persistent object storage
Different types of materials / media
Client
Server: support for distributed multi-user access
Individual applications
Presentation specifications
Anchors

Figure 1: The Dexter model layers and interfaces.[1]

The *Within-component layer* corresponds to individual applications. The applications are responsible, for example, for supporting content selections for link anchoring.

The interface between the storage and within-component layers is based on the notion of *anchors*. Anchors consist of an *identifier* that can be referred to by links and a *value* that picks out the anchored part of the material.

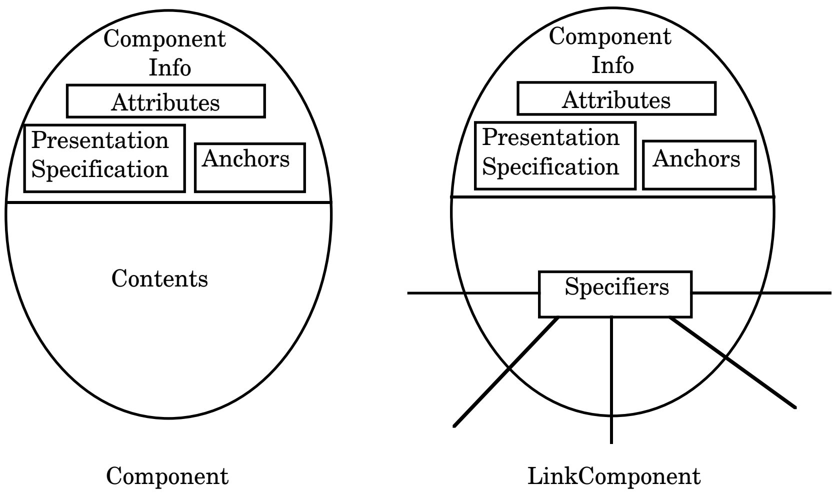


Figure 2: Component structure in the Storage Layer.

The Runtime layer is responsible for handling links, anchors, and components at runtime. Objects in the runtime layer include *sessions*, managing interaction with particular hypertexts, and *instantiations*, managing interaction with particular components. The runtime layer provides tool independent user

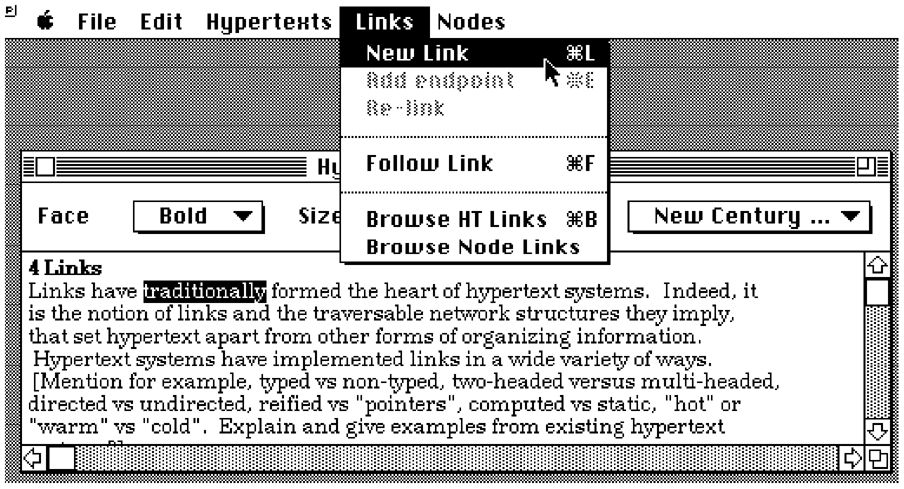
[1]Some of the text appearing in the figure is our own.

interface facilities through operations like NewComponent, AddLinkEndpoints, and FollowLink.

The interface between the Storage layer and the Runtime layer includes *presentation specifications* which determine how components are presented at runtime. Presentation specifications might include information on screen location and size of a presentation window, as well as a “mode” for presenting a component. Halasz and Schwartz [8] use the example of an animation component that can be opened in either run mode or edit mode.

3. Links

Links have traditionally formed the heart of hypertext systems. Indeed, the traversable network structures formed by links distinguish hypertext from other means of organizing information. Hypertext systems have implemented links in a number of ways, many of which are unified by Dexter’s notion of *link component*. In addition to the typical source/destination links, Dexter can model multi-headed links. Furthermore, because links are components, they can be the endpoints of other links. Through the use of specifiers, the Dexter model supports computed as well as static links. Simple “typing” can be supported by adding attributes to the link component. But DHM also supports full-fledged typing of links due to its object oriented component design.



4 Links
Links have traditionally formed the heart of hypertext systems. Indeed, it is the notion of links and the traversable network structures they imply, that set hypertext apart from other forms of organizing information. Hypertext systems have implemented links in a wide variety of ways. [Mention for example, typed vs non-typed, two-headed versus multi-headed, directed vs undirected, reified vs "pointers", computed vs static, "hot" or "warm" vs "cold". Explain and give examples from existing hypertext

Figure 3: Creating a link with one endpoint.

Though in principle a Dexter link could have fewer than two endpoints, this is expressly forbidden by the model’s semantics. In DHM, we have relaxed this constraint; that is, "dangling" links having zero or one endpoint are perfectly legal. This means that we can avoid the modal "start link / end link" link creation style of many hypertext interfaces. In DHM's user interface, links can be created in two ways: (1) a “New Link” operation creating a link having one endpoint based on the current selection in the active editor (Figure 3); in this case no instantiation or link editor is opened. And, (2) via a new node operation creating a link with an open instantiation and link editor; in this case the link has no endpoints (see the ‘Link 6’ link instantiation in Figure 4). Endpoints can be added to the link at any time and as shown in Figure 4, links can have other links as an endpoints.

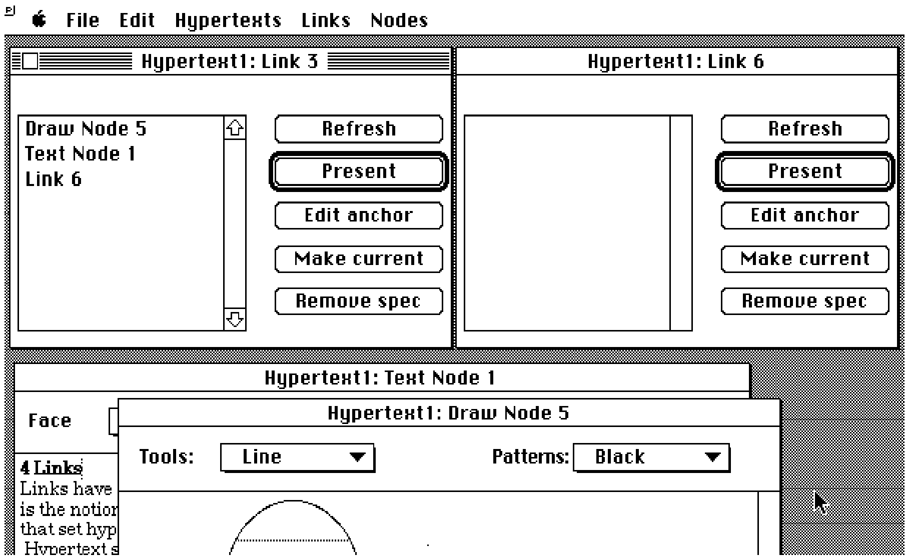


Figure 4: Two open link instantiations.

In our implementation of links, we confronted two problems with Dexter's model: 1) its aversion to dangling links and 2) its notion of link directionality.

Dangling links

In spite of Dexter's explicit aversion to dangling links, we chose to support them for several reasons. First, they allow lazy updating and garbage collection following node and anchor deletion. This is useful when the link to be deleted (or modified) lives on another machine or is currently locked by another user. A second, related situation involves data objects outside the control of the hypermedia, for example, files with component data needing to be moved or deleted. Third, the dangling endpoint can be "re-linked" or re-connected to another node or anchor without having to rebuild the entire link (especially useful for multi-headed links). Finally, dangling links can be created intentionally as placeholders when the desired endpoint node or anchor does not yet exist.[2] The presence of such dangling links could be monitored by the system either on command or automatically. Users could then be prompted to reconnect "missing" link endpoints.

We imagine four different dangling link situations arising in an integrated Dexter-based hypermedia system: 1) the endpoint's component has been deleted, 2) its anchor has been deleted, 3) relevant data objects referred to by the component's contents are unavailable, and 4) the anchor value is invalid. In the first two cases, the deletion operation modifies the objects so that later calls to followLink raise exceptions. Component deletion is implemented by clearing the

[2]An anonymous reviewer mentioned an example from asynchronous collaborative writing: When sharing parts of a hypertext, the links should dangle while being shared, but be re-attached when returned.

anchors list and component contents, and setting a “deleted” flag. Anchor deletion is carried out similarly.

Cases 3 and 4 usually result from actions outside the control of the hypermedia. For example, data objects making up a component's contents can become unavailable if the contents is a file identifier, and the file has been moved or deleted independent of the hypermedia (case 3). In this case the followLink operation should catch the file system exception and pass it along as a dangling exception to the user.

In case 4, the data specified by the anchor value becomes invalid when relevant parts of the component's contents are modified with editors outside the hypermedia. This situation is impossible to detect in general during a CreateLinkMarker or a FollowLink operation, since the lookup/computation of anchor value may not raise an exception. An example is when the anchor value is still legal but out of date, as a result of "unauthorized" editing of the surrounding text. Currently in DHM, we have implemented detection and re-link options for case 1.

Link directionality

The Dexter model includes only minimal motivation for its notion of link directionality. We are told that each link specifier indicates a directionality using one of the constants FROM, TO, BIDIRECT, or NONE, depending on whether the endpoint is to be interpreted as a source, destination, both source and destination or neither, respectively. Furthermore, every link must have at least one TO specifier.[3] Such directionality constants are used to model the link semantics of existing hypermedia systems. For example, Intermedia links are modelled with BIDIRECT directionality on all specifiers. This is because the endpoints of an Intermedia link are directionally interchangeable [6].[4] In NoteCards, on the other hand, links have a definite source and destination [9].

However, this scheme seems insufficient to model the ways people interpret link direction in practice. Consider the following three notions of directionality:

Semantic direction: This concerns the semantic relationship between the components captured by the link. For example, a “Support” link connecting components A and B has a direction in which it normally “reads”; the argument in Component A “supports” the claim in Component B [17, Ch. 4].

Creation direction: This direction corresponds to the order in which the link endpoints were created: the source of the link is the first endpoint created while the destination is the last.

Traversal direction: This direction specifies how the link can be traversed. HyperCard links, for example, can only be traversed from source to destination.[5]

[3]An anonymous reviewer informs us that the wording of the constraint should have been, "at least one TO or BIDIRECT specifier."

[4]Intermedia anchor attributes can, however, be notated with directionality information.

[5]This is because HyperCard links are implemented as “Go” statements in a script in the link’s source component. This also means that link's cannot normally be seen from their destination cards.

NoteCards links can be traversed in both directions, although the interface style is different. When moving from source to destination, one clicks on the source anchor’s icon. To move from destination to source, a menu of “back-links” is opened in the destination component and the appropriate link icon is chosen.

These senses of link direction are in principle orthogonal. For example, the directions in which one can physically traverse a link in a particular system need not depend on the link’s semantic direction. Nonetheless, many systems enforce dependencies. In NoteCards, for example, the creation direction corresponds to the traversal direction.

The question for the Dexter model is, which sense of directionality is being modelled by a particular choice of directionality constant in a link specifier? Like Dexter, we have not explicitly addressed this question in DHM; by default, all specifiers start with a BIDIRECT constant which users can later modify. And by default, the direction attribute is interpreted as a traversal direction, i.e. following a link means bringing up the TO and BIDIRECT endpoints. However, we plan to allow tailoring of the followLink operation, e.g. by specializing it to support different behavior.

4. Anchors

One of Dexter’s major contributions is its explicit identification of anchors as the “glue” connecting network structures to the contents of particular components. Anchors are a controlled means of referring into the “within-component” layer. Without them, links connect only whole components.

Dexter's anchors are defined relative to a component and have an id that is unique within that component. Link specifiers must identify both the component id and the anchor id. Explicit mention of the ids can be avoided, however, by use of the resolver function. Thus the component appearing at a link’s endpoint can be computed dynamically at run-time.

The biggest problem with Dexter’s model of anchors is that they are not properly related to composites. That is, although the contents of a composite (a list of baseComponents) is “visible” (i.e. explicitly represented) in Dexter, no mention is made of how anchors should refer to baseComponents within a parent composite. In DHM we allow composites to include full-fledged components (see Section 5), adding further problems. For example, can an anchor in the parent composite be tied to an anchor in one of its components? That is, can a link “indirect” through a composite’s anchor, to an enclosed component’s anchor?

There are other anchor-related issues not discussed in the Dexter model. Consider, for example, links to whole components. Should they have an empty Anchor reference in the specifier or should there be a “whole-component” anchor type? In that case, should all whole-component links share a single whole-component anchor, or should there be one anchor for each link endpoint? Indeed the general issue of sharing versus multiplying anchors is left open in Dexter. When creating a new link, should one always try to reuse any existing applicable anchor? Suppose there is more than one?

DHM extends Dexter's model of anchors in several ways. First, we use dynamic references ("pointers") instead of anchor ids.[6] This means that link specifiers point directly at component anchors avoiding the need for an accessor function. Similar benefits accrue from our block-structured type definitions. In the same way that a component need not keep a "back-pointer" to its enclosing hypertext, an anchor need not keep a "back-pointer" to its enclosing component.

DHM distinguishes three high-level anchor types which are independent of the type of the enclosing component. *Whole-component* anchors support the degenerate case of link endpoints not anchored within a component's contents.[7]

A *marked* anchor is one for which an object is directly embedded in the component's contents. This object is called a *link marker* in Dexter. It may or may not be visible – indeed, some link markers (e.g. an Emacs "mark") may never be made visible as such. Link markers can be implemented in a variety of ways depending on the medium and the application. Visible icons inserted in text or graphic windows can serve as link markers (e.g. NoteCards link icons). But a link marker can also correspond to what Meyrowitz [14] calls a "permanent tie." Such an object can "track" editing changes to the component's contents including changes to the selection itself. The instantiation may or may not choose to make the link marker visible (see e.g. Intermedia's arrow icon registering the presence of a permanent selection). DHM supports link markers in text components by maintaining outlined regions around the anchored text selections. A command-click within the link marker region invokes a follow on the corresponding marked anchor's links.

Unmarked anchors have no link markers; normally their location within a component must be computed. Text components in DHM support a specialization of unmarked anchor called *keyword* anchor, resembling the endpoints of HyperTies text links [16]. Creating an anchor requires saving a copy of the text string corresponding to the current selection. Following a link requires checking whether the current selection's text matches a saved keyword anchor and if so, following that anchor's links.

That which sets apart a marked anchor from an unmarked one is the ability to retrieve the anchor directly from a selection in the component's editor. If a link marker is currently selected (or clicked on) in an open instantiation, then the instantiation should be able to directly access the corresponding marked anchor. This is in contrast to unmarked anchors, where a search is required. In general, each unmarked anchor must be asked whether it is currently "selected" (or perhaps more descriptively, "applicable"). The operation of following a link from a marked anchor should take constant time, whereas following a link from an unmarked anchor requires in the worst case time proportional to the total number of unmarked anchors in the component.[8]

[6]Utilizing an OODB makes our pointers persistent. We nonetheless maintain component and anchor ids in order to be able to generate transportable interchange formats for the hypermedia structure.

[7]In DHM, all links with whole-component endpoints in a component share a single whole-component anchor.

[8]This can be improved using hash tables and the like.

5. Composites

The notion of structure (usually hierarchical) has been a part of most hypertext systems since the time of NLS/Augment in the 60's [2]. To name just one example, in KMS (as well as its ancestor ZOG), a hierarchical structuring capability is built in to every node [1]. That is, all nodes (called "frames" in ZOG/KMS) can act as containers for other nodes. Usually, however, hierarchical structuring (and on rare occasions, non-hierarchical structuring), is supported through separate mechanisms.

In his landmark "Seven Issues" paper, Halasz proposed that the *composite* be elevated to peer status with atomic nodes and links [7]. Composites would provide a means of capturing non-link based organizations of information, making structuring beyond pure networks an explicit part of hypertext functionality.[9] Halasz also argued for the related notions of computed and virtual composites. The contents of a computed composite might be, say, the result of a structural query over the hypertext returning sets of nodes and links as "hits." A virtual composite is created on demand at runtime, but not saved in the database. Later, in Aquanet [13], the composite idea was used to capture slot-based structures consisting of nodes and *relations*, multi-headed variants of links.

Halasz [7] also criticized purely link-based structures arguing that they lack a single node capturing the overall structure. The Dexter model's composite addresses this critique. As an aggregation of base components, it acts both as a full-fledged node in the network, and as container for the structure. In particular, such a composite can contain link components (in addition to atomic nodes and other composites) and thus capture complex non-hierarchical network structures (like Aquanet relations). Furthermore, because of Dexter's clean separation of storage and runtime environments, virtual composites are a simple variant.

Though Dexter's notion of composite is a significant step forward, it is only one point in a spectrum of possible designs, each having certain advantages and meeting certain needs. Our prototype opens Dexter's notion of composite to tailoring for particular applications. Users adding a new composite type to our prototype make choices along several dimensions:

Virtual / non-virtual components

Any component type (not just composites) can be made virtual by the setting of a flag. Such components resemble normal components, but are usually not saved in the database. If however, another component (say, a link) points at the virtual, then it is indeed saved. Virtual components resemble objects in a dynamic programming environment; if they are not pointed at, then garbage collection reclaims them.

Computed / static components

Any component type (again, not just composites) can be the result of a computation rather than manually created by the user. A typical example is a component created on the basis of executing a query. An attribute contains the

[9]A similar appeal was made by van Dam in his attack on links as "go to" statements [19].

information used to perform the computation. The component's contents can later be recomputed, either on demand or automatically. Some computed components (like browsers) reflect the contents or structure of parts of the network. In such cases, recomputation can be based on periodic checks of the relevant sub-net, or be forced by changes to the relevant components or structures.

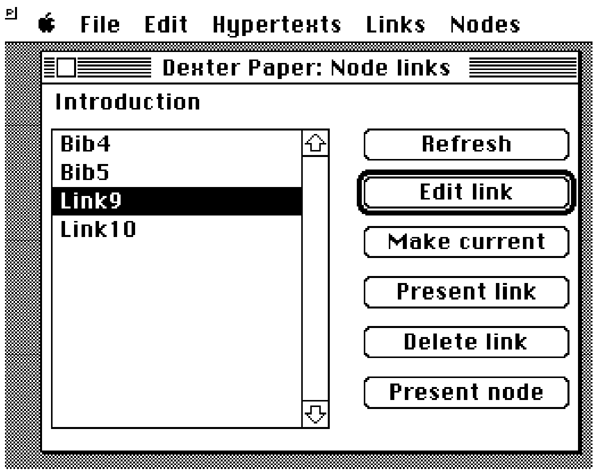


Figure 5: A link browser composite in DHM; lists all links to and from the 'Introduction' node.

Component contents

Typically, the contents of a component in a hypermedia system is not simply a flat set of enclosed data objects as suggested by the Dexter model. The contents are often structured and can include external data objects or references to other components

Figure 5 shows an example of a composite type supporting link browsing in DHM. The Link browsers are implemented as virtual, computed composites with contents consisting of lists of references to LinkComponents. Though not anticipated by the Dexter model, this kind of component was fairly easy to implement using the framework described above.

6. Integration and component contents

The phenomenon of system developers "owning the world" is becoming increasingly rare. Today, most practical computer environments consist of a number of third-party applications, perhaps customized for particular work settings by local programmers or user "tailors." Unfortunately, the application's inner workings and structures are rarely open to the developer trying to integrate them into a larger environment. The problem is exacerbated if the environment includes a variety of platforms.

In the last few years, researchers and developers have tried to use hypermedia to address this integration problem [6,11,14,15]. They argue that rather than build a hypermedia system that includes all the applications needed in the work setting, one should employ hypermedia as a linking architecture, "connecting" the world rather than "owning" it.

The Dexter reference model makes certain important contributions to this effort. At the architectural level, Dexter distinguishes between objects belonging to the hypermedia (both runtime and storage), and the "within-component layer" belonging to an application. In addition to describing the hypermedia data model, Dexter offers two important concepts that help cross the boundary: anchors and presentation specifications (or "pspecs"). Anchors support linking to and from points within the contents of an application document. Pspecs provide a means of storing with a Dexter component information on how to start and configure the appropriate application.

In this way, Dexter opens the possibility of integrating third-party applications into a linked hypermedia environment. But it leaves unaddressed at least two important integration-related questions. First, Dexter does not distinguish between components whose contents are managed (in particular, stored) by the hypermedia and those whose contents are managed by third-party applications.

The second problem involves application documents having internal structure. Such documents can be integrated as a single unit into the hypermedia using a component "wrapper," but often the document's internal structure needs to be "exposed" for link anchoring. Dexter suggests using composite components, but says almost nothing about how to anchor within the subcomponents of a composite. Nor does it discuss whether or how a composite component's structure should model the internal structure of an application document.

In what follows, we discuss various possibilities for storing and structuring component contents.

Atomic components

Figure 6 shows two possible relations between an atomic component and an anchored data object.

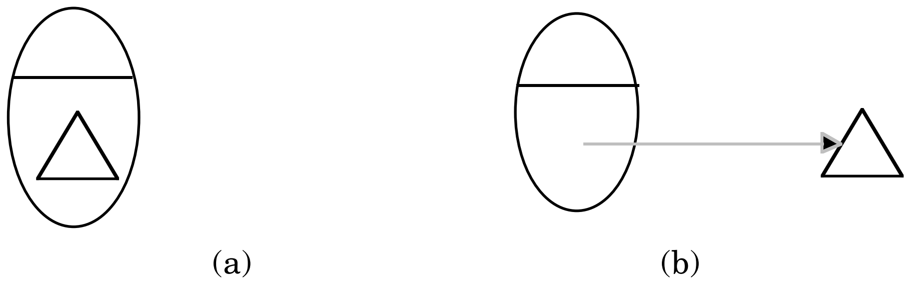


Figure 6: Data is either part of an atomic component's contents or referenced by it. (Dotted arrows denote references out of the hypermedia structure, e.g. file identifiers.)

Figure 6a shows the traditional situation where an application and its data objects are built into the hypermedia system. DrawComponents in DHM, for example, encapsulate lists of graphical objects stored in the OODB together with the components.

In Figure 6b, on the other hand, data objects wrapped by a component are stored separately and only referenced by the contents of the component. In DHM such a component/data object relationship characterizes *FileComponents* and *MovieComponents*. FileComponents are used to wrap arbitrary files in the file system, using file ids stored in the component contents. In this way, DHM supports linking (using WholeComponent anchors) to, say, Microsoft Word or Excel documents. The followLink operation launches the appropriate applications on the files as if they had been double-clicked in the Finder.

MovieComponents "wrap" Quicktime movies,[10] large multimedia data objects (from 5 to several hundred MB) too complex to be easily stored in the hypermedia's OODB. Hence, they are better handled using MovieFiles referred to by the component contents.[11] In this case, the component contents is also a file identification object.

Typically, an atomic data object belongs to exactly one atomic component. But there are cases where two or more components need to share data. Here the components could have different types and/or different sets of anchors. Such multiple "views" can be supported by the containment style shown in Figure 6b.

Composite components

With regard to more complex structures of components and data objects, we found Dexter's notion of composite too narrow. According to Dexter, a composite may only contain encapsulated data objects (see for example, the bottom left composite in Figure 7). As noted by Halasz & Schwartz [8] this kind of composite can model structures like graphical canvases. For other applications, however, composites need to refer to external data objects or other components. In the following we discuss examples of such composite types.

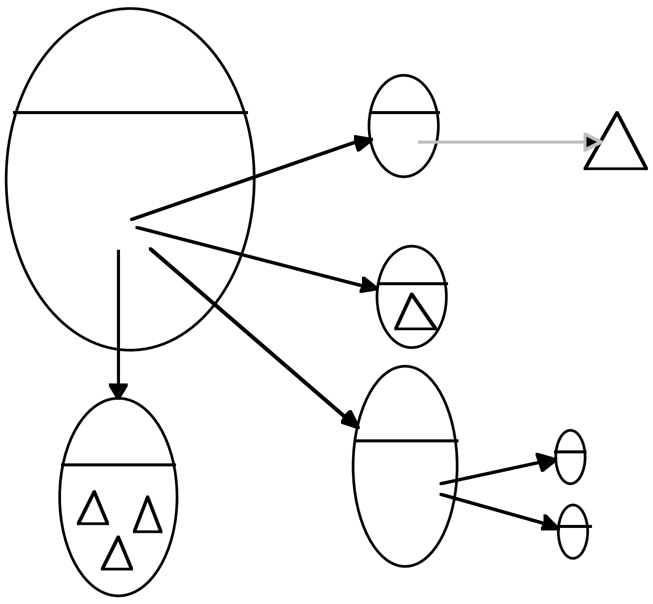


Figure 7: A composite referring to components of arbitrary type. (Solid arrows denote internal pointers to hypermedia components.)

[10]QuickTime™ by Apple Computer Inc. implements a format for storing/compressing digitalized video.

[11]In the current version of DHM, we support only one movie per MovieFile (and thus, one per component).

Composites "containing" components

We first consider composites that refer to other *components* as shown in Figure 7. One example is the TableTopComposite used to save configurations of components presented together on the screen [18].

The contents of a TableTopComposite in DHM is a list of “pointers” to components of arbitrary type (including other links and composites); the composite does not directly contain or wrap the data objects.

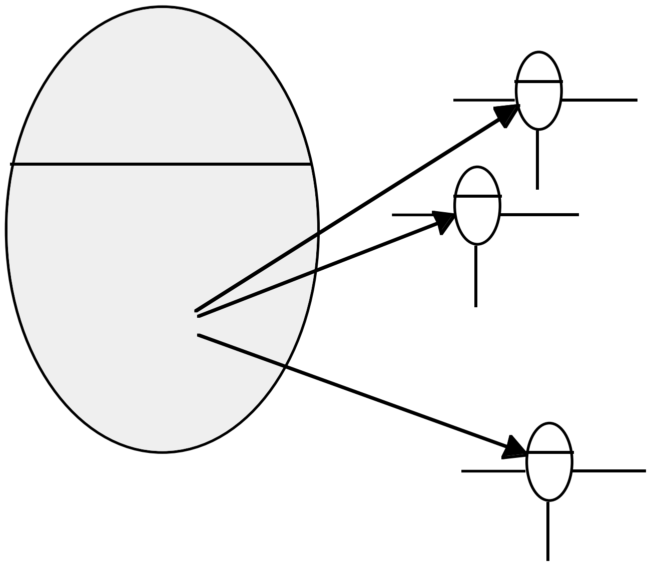


Figure 8: A virtual composite restricted to refer to LinkComponents. (The shading indicates that the composite is virtual).

Another example is a *search* composite. Here the contents is a list of components (again of arbitrary type) resulting from a title search or a query over component attributes. In DHM, such search composites are implemented as virtuals (see Section 5).

Figure 8 shows a slightly different kind of composite also used to group components. In this case, the composite is both virtual and has contents restricted to certain component types.

The VirtualLinkComposite shown in Figure 8 is used in DHM to implement a variety of link browsers. VirtualLinkComposites are “computed” composites; their creation requires collecting a set of links for an entire hypertext, a specific component, or a specific anchor, depending on the kind of link browser.

When appropriate, restricting the component types pointed at by a composite allows customization of the composite's interface. For example, the VirtualLinkComposite interface supports inspecting individual link specifiers. A non-typed composite would require runtime checking of the types of contained objects.

Encapsulated data objects

Up to this point we have focused on composites referring to other components; we now turn to composites referring directly to data objects. In Figure 9, the data objects depicted as triangles are encapsulated in a "container" object (drawn as a rectangle). In this case, the internal structure of the rectangular object is visible to the hypermedia system. Hence the composite and its nested components can refer both to the enclosing object and to its internal structure.[12]

[12]A nested component is one whose definition lies within the block structure of the parent component and thus can only exist in the context of the parent component.

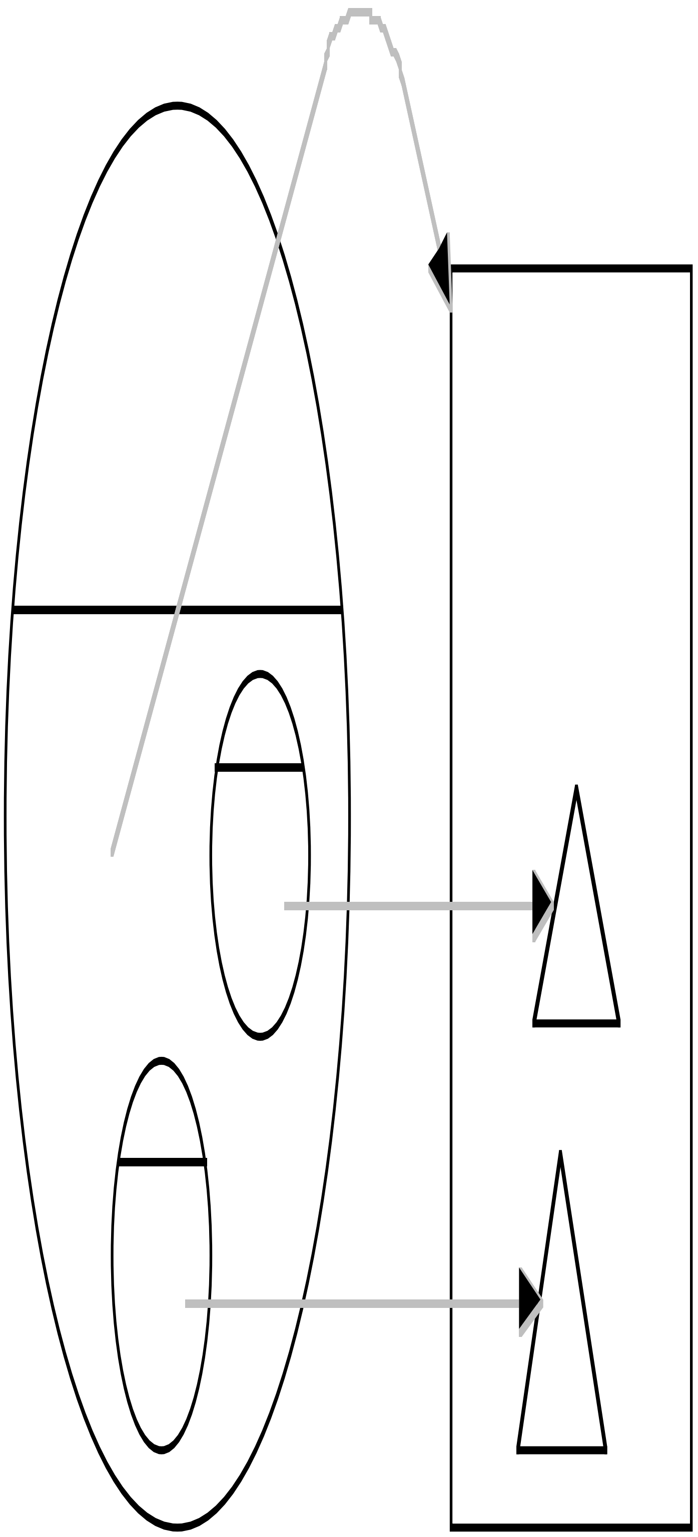


Figure 9: Typed composite with nested components points at encapsulated data objects.

An example of such a composite is used to represent modules in the Mjølner Beta programming environment [12]. Mjølner supports fine grained modularization of programs using atomic modules called ‘fragments’ contained in parent 'fragment groups'. Each fragment group is stored on a file. To represent such structures in DHM, we use a FragmentGroupComposite whose contents includes a reference to a fragment group file and a list of references to atomic FragmentComponents, declared inside the block structure of the FragmentGroupComposite. The nested structure of the ‘real world’ data objects (fragments and fragment groups) is mapped directly onto the nested structure of the representing components. Hence, we can link both to the composite and to the nested atomic components representing individual fragments.[13]

We provide anchors at the FragmentGroupComposite level, to comments made at the group level, and at the FragmentComponent level, to comments and source code belonging to individual fragments.

Structured composites

An Aquanet *relation* [13], is an example of a hypermedia composite with structured contents. A fundamental feature of an Aquanet relation is that it resembles a multi-headed link with named endpoints.

We suggest implementing such relations as composites with contents consisting of a keyed table of component references (see Figure 10). Such a composite can refer to basic objects (atomic components) as well as to other relations (structured composite components). In addition, instantiations of such composites can support link-like "endpoint" presentation. Here, "endpoints" refer to the components pointed at by the composite's encapsulated structure.

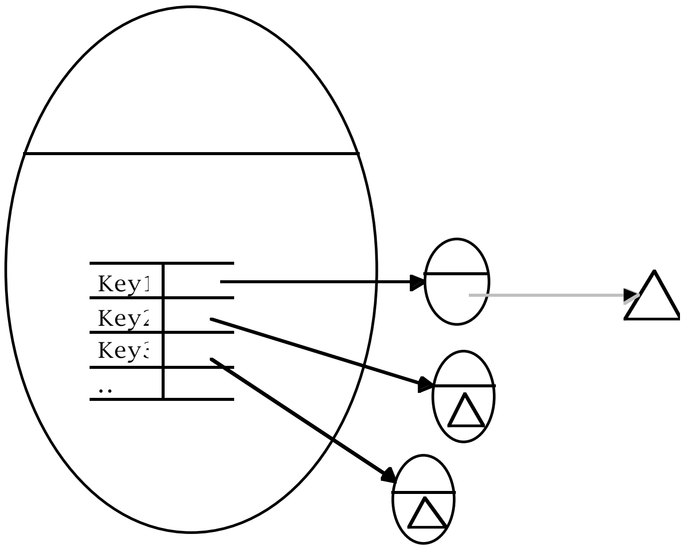


Figure 10: A composite with structured contents.

Summary

The above examples show the need to support a broad view of component contents when developing open Dexter-based hypermedia systems. Integrating components of the Storage layer with data objects of the within-component layer

[13]Note that our block structured environment obviates the need for a backwards reference from atoms representing fragments to composites representing the fragment group.

is one important aspect (as illustrated by the difference between a DrawComponent and a MovieComponent). Another is the internal integration of components and composites as illustrated by the cases of TableTopComposite and VirtualLinkComposite. Table 1 summarizes the discussion in this section using three feature dimensions.

Structure of contents	Location/type of contents	Location of definition
• Atomic • Unstructured collection • Structured collection - sorted list - keyed table - tree - ...	• Data objects - within component - outside component • Components - restricted types - unrestricted	• Nested in this component • Resides outside this component

Table 1: Features of component contents.

7. Concluding remarks

This paper discussed experiences from Dexter-based hypermedia development in the DeVise project at Aarhus University. The work has lead to clarifications and extensions to the Dexter model. Those treated in this paper are concerned with integration issues and the design of central object classes like links, anchors, and composites.

The open, extensible architecture we are developing will provide a basis for developing multi-user hypermedia applications in a variety of domains. In a forthcoming paper [3], we focus on architecture and tailorability issues for the development of an open, extensible hypermedia system based on Dexter. We discuss integrating the Runtime layer with external applications and introduce a fourth Presentation layer to the Dexter architecture.

In addition, our work on Dexter based hypermedia will be part of the Esprit III project, EuroCode, aimed at developing a CSCW application development “shell." One of the issues will be to extend the Dexter based hypermedia architecture to support concurrency control, e.g. via an event mechanism as described in [20].

Acknowledgements

This work has been supported by the Danish Research Programme for Informatics, grant number 5.26.18.19. Our thanks also go to the members of the DeVise project at Aarhus University and four anonymous reviewers.

References

1. Akscyn, R., McCracken D., & Yoder, E. 1988. KMS: A distributed hypermedia system for managing knowledge in organizations. CACM, 31, 7, (July), 820-835.
2. Engelbart, D. C. 1984. Authorship provisions in AUGMENT. *Proceedings of the 1984 COMPCON Conference, COMPCON '84 Digest*, (San Francisco, Calif., February), pp. 465-472.
3. Grønbæk, K & Trigg, R.H. An open, extensible hypermedia system based on the Dexter Hypertext Reference Model. Computer Science Department, Aarhus University, Denmark, in preparation.
4. Grønbæk, K. & Knudsen, J. L. Tools and Techniques for Experimental System Development. In Systä, K., Kellomäki, P., & Mäkinen, R.(eds.) *Proceedings of the Nordic Workshop on Programming Environment Research*, Tampere, Finland, January 8-10, 1992.
5. Grønbæk, K., Kyng, M., & Mogensen, P. CSCW challenges in large-scale technical projects – a case study. To appear in *Proceedings of Conference on Computer Supported Cooperative Work '92, Toronto, Ontario*, November, 1992.
6. Haan, B.J., Kahn, P., Riley, V.A., Coombs, J.H., & Meyrowitz, N.K. IRIS Hypermedia Services. *Communications of the ACM* 35(1), January 1992, pp.36-51.
7. Halasz, F. 1988. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Commun. ACM*, 31, 7, (July), 836-852.
8. Halasz, F., & Schwartz, M. 1990. The Dexter hypertext reference. *Proceedings of the Hypertext Standardization Workshop*, (Gaithersburg, Md., January), pp. 95-133.
9. Halasz, F., Moran, T., & Trigg, R. 1987. NoteCards in a nutshell. Proceedings of the CHI '87 Conference, (Toronto, Canada, April), pp. 45-52.
10. Hem, J.A., Madsen, O.L., Møller, K.J., Nørgaard, C., & Sloth, L. Object Oriented Database Interface. Deliverable D5.2, *ESPRIT project 5305 EuroCoOp IT Support for Distributed Cooperative Work*, December 1991.
11. Kacmar, C.J. & Leggett, J.J. PROXHY: A Process-Oriented Extensible Hypertext Architecture. *ACM Transactions on Information Systems* 9(4), October 1991, pp. 399-419.
12. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., & Nygaard, K: *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, forthcoming (1992).
13. Marshall, C.C., Halasz, F.G., Rogers, R.A., & Janssen, W.C. Aquanet: a hypertext tool to hold your knowledge in place. *Proceedings of Hypertext '91*, ACM New York, December 1991, pp. 261-275.
14. Meyrowitz, N. The Missing Link: Why We're All Doing Hypertext Wrong. In Barrett (ed.) *The Society of Text*. MIT Press, Cambridge Massachusetts, 1989, pp. 107-114.

15. Pearl, Amy. 1989. Sun’s link service: A protocol for open linking. *Proceedings of the Hypertext ‘89 Conference*, (Pittsburgh, Pa., November), pp. 137-146.

16. Shneiderman, B. 1987. User interface design for the HyperTIES electronic encyclopedia. *Proceedings of the Hypertext '87 Conference*, (Chapel Hill, November), pp. 189-194.

17. Trigg, R. 1983. A network-based approach to text handling for the online scientific community. Ph.D. dissertation. University of Maryland (University MicroFilms #8429934), College Park, Md.

18. Trigg, Randall. 1988. Guided tours and tabletops: Tools for communicating in a hypertext environment. *ACM Trans. Off. Inf. Syst.*, 6,4, (October), 398-414.

19. van Dam, A. 1988. Hypertext '87: Keynote Address. *CACM*, 31, 7, (July), 887-895.

20. Wiil, U. K. Using events as Support for Data Sharing In Collaborative Work. In *Gorlin, K. & Sattler, C. (eds.) Proceedings of the International workshop on CSCW*, Berlin April, 1991. Institut für Informatik und Rechentechnik, Berlin.