# THE DEXTER HYPERTEXT

**The Dexter Hypertext Reference Model is an attempt to capture, both formally and informally, the important abstractions found in a wide range of existing and future hypertext systems.[1] The goal of the model is to provide a principled basis for comparing systems as well as for developing interchange and interoperability standards. The model is divided into three layers. The storage layer describes the network of nodes and links that is the essence of hypertext. The run-time layer describes mechanisms supporting the user's interaction with the hypertext. The within-component layer covers the content and structures within hypertext nodes. The focus of the model is on the storage layer and the mechanisms of anchoring and presentation specification that form the interfaces between the storage and the storage layer and the within-component and run-time layers, respectively.**

What do hypertext[2] systems such as NoteCards [10], Neptune [4], KMS [1], Intermedia [23] and Augment [6] have in common? How do they differ? In what way do these systems differ from related classes of systems such as multimedia database systems? At a very abstract level, each of these hypertext systems provides its users with the ability to create, manipulate, and/or examine a network of information-containing nodes interconnected by relational links. These systems differ markedly, however, in the specific data models and sets of functionality they provide to their users. Augment, Intermedia, NoteCards, and Neptune, for example, all provide their users with a universe of arbitrary-length documents. KMS and HyperCard, in contrast, are built around a model of a fixed-size canvas onto which items such as text and graphics can be placed. Given these two radically different designs, do these systems have anything in common in their notions of hypertext nodes?

In an attempt to provide a principled basis for answering these questions, this article presents the Dexter Hypertext Reference Model. The model provides a standard hypertext terminology coupled with a formal model of the important abstractions commonly found in a wide range of hypertext systems.[3] Thus, the Dexter model serves as a standard against which to compare and contrast the characteristics and functionality of various hypertext (and nonhypertext) systems. The Dexter model also serves as a principled basis on which to develop standards for interoperability and interchange among hypertext systems.

The Dexter reference model described in this article was initiated as the result of two small workshops on hypertext. The first was held in 1988 at the Dexter Inn in New Hampshire—hence the name of the model. The workshops had representatives from many of the major existing hypertext systems, and a large part of the discussion at these workshops was the elicitation of the abstractions common to the major hypertext systems.[4] The Dexter model is an attempt to capture, further develop, and formalize the results of these discussions.

Another important focus of the workshops was an attempt to find a common terminology for the hypertext field. This was an extremely difficult task due to the absence of an understanding of the common (and differing) abstractions among the various systems. The term "node" turned out to be especially difficult to define given the extreme variation in the use of the term across the various systems. By providing a well-defined set of named abstractions, the Dexter model provides a solution to the hypertext terminology problem. It does so, however, at some cost. In order to avoid confusion, the model does not use contentious terms such as "node," preferring neutral terms like "component" for abstractions in the model.

This article briefly refers to architectural concepts found in a number of existing hypertext systems including Augment [6], Concordia/Document Examiner [22], HyperCard [8], Hyperties [18], IGD [7], Intermedia [23], KMS [1], Neptune/HAM [4], NoteCards [9], the Sun Link Service [17], and Textnet [20]. Appropriate background material on these systems can be found in Conklin [3] and in the proceedings of the Hypertext 87 [11] and Hypertext 89 [12] conferences.

## An Overview of the Model

The Dexter model divides a hypertext system into three layers, the *run-time* layer, the *storage* layer and the *within-component* layer, as illustrated in Figure 1. The main focus of the model is on the storage layer, which models the basic node/link network structure that is the essence of hypertext. The

[1]This article is a revision of the original Halasz and Schwartz paper that appeared in the *Proceedings of the Hypertext Workshop* (National Institute of Standards and Technology, Gaithersburg, Md, Jan. 16–18, 1990, NIST Special Publication 500-178, March 1990, pp. 95–133). Aside from minor editorial modifications, the main change was the removal of the formal Z specification [19]. Those readers interested in a formal presentation of the Dexter model and a precise specification of the interface and integrity constraints are referred to the original article.

[2]The terms *hypertext* and *hypermedia* are often distinguished, with hypertext referring to text-only systems and hypermedia referring to systems that support multiple media. This distinction is not made in this article; the term hypertext is used generically to refer to both text-only and multimedia systems.

[3]See the original NIST paper for a Z-based representation of the formal model.

[4]Participants in the two workshops are listed in the acknowledgments section of this article. Among the systems discussed at the workshops were: Augment, Concordia/Document Examiner, IGD, FRESS, Intermedia, HyperCard, Hyperties, KMS/ZOG, Neptune/HAM, NoteCards, the Sun Link Service, and Textnet.

# REFERENCE

# Model

hyper MEDIA

FRANK HALASZ
MAYER SCHWARTZ

*Edited by*
*Kaj Grønbæk and*
*Randall H. Trigg*

storage layer describes a 'database' that is composed of a hierarchy of data-containing "components" interconnected by relational "links." Components correspond to what is typically thought of as nodes in a hypertext network: *cards* in NoteCards and HyperCard, *frames* in KMS, *documents* in Augment and Intermedia, or *articles* in Hyperties. Components contain the chunks of text, graphics, images, animations, for example, that form the basic content in the hypertext network.[5]

The storage layer focuses on the mechanisms by which link and non-link components are "glued together" to form hypertext networks. The components in this layer are treated as generic containers of data. No attempt is made to model any structure within the container. Thus, the storage layer does not differentiate between text components and graphics components. And it does not provide any mechanisms for dealing with the well-defined structure inherent within a structured document (e.g., an ODA document) component.

In contrast, the within-component layer of the model is specifically concerned with the contents and structure *within* the components of the hypertext network. This layer is purposefully not elaborated within the Dexter model. The range of possible content/structure that can be included in a component is not restricted. Text, graphics, animations, simulations, images, and many more types of data have been used as components in existing hypertext systems. It would not be realistic to attempt a generic model covering all of these data types. Instead, the Dexter model treats within-component structure as being outside of the hypertext model per se. It is assumed that other reference models designed specifically to model the structure of particular applications, documents, or data types (ODA, IGES, for example) will be used in conjunction with the Dexter model to capture the entirety of the hypertext, including the within-component content and structure.

[5]Actually, these are known in Dexter as *atomic* components. As will be shown in the next section, links and *composites*, entities formed by composing components, are also kinds of components.

A critical piece of the Dexter model is the interface between the hypertext network and the within-component content and structure. The hypertext system requires a mechanism for addressing (referring to) locations or items *within* the content of an individual component. In the Dexter model, this mechanism is known as *anchoring*. The anchoring mechanism is necessary, for example, to support span-to-span links as found in Intermedia. In Intermedia, the components are complete structured documents. Links are possible not only between documents, but between spans of characters within one document and spans of characters within another document. Anchoring is a mechanism that provides this functionality while maintaining a separation between the storage and within-component layers.

The storage and within-component layers treat hypertext as an essentially passive data structure. Hypertext systems, however, go far beyond this in the sense that they provide tools for the user to access, view, and manipulate the network structure. This functionality is captured by the run-time layer of the model. As in the case of within-component structure, the range of possible tools for accessing, viewing, and manipulating hypertext networks is far too broad and too diverse to allow a simple, generic model. Hence the Dexter model provides only a bare-bones model of the mechanism for presenting a hypertext to the user for viewing and editing. This presentation mechanism captures the essentials of the dynamic, interactional aspects of hypertext systems, but does not attempt to cover the details of user interaction with the hypertext.

As in the case of anchoring, a critical aspect of the Dexter model is the interface between the storage layer and the run-time layer. In the Dexter model this is accomplished using the notion of *presentation specifications*. Presentation specifications are a mechanism by which information about how a component/network is to be presented to the user can be encoded into the hypertext network at the storage layer. Thus, the way in which a component is presented to the user can be a function not only of the specific hypertext tool that is doing the presentation (i.e., the specific run-time layer), but can also be a property of the component itself and/or of the access path (link) taken to that component.

Figure 2 illustrates the importance of the presentation specification mechanism. In this figure, there is an animation component taken from a computer-based training hypertext. This animation component can be accessed via two link components. When following the standard link, the animation component is brought up as a running animation. In contrast, when following the teacher-owned "edit" link, the animation is brought up in editing mode ready to be altered. To separate these two cases, the run-time layer needs to access presentation information encoded into the links in the network. Presentation specifications are a generic way of doing just this. Like anchoring, they form an interface that allows the storage layer to communicate in a generic way with the run-time layer without violating the separation between the two layers.

Figure 3 attempts to give a flavor of the various layers of the Dexter model as they are embedded within a typical hypertext system. The figure depicts a 4-component hypertext network. The storage layer contains four entities: the four components including the link. The actual contents (text and graphics) of the components are located to the right of the storage layer in the within-component layer. In the run-time layer, the single graphics component is presented to the user. The link emanating from this component is marked by an arrowhead located near the bottom of the component's window on the computer screen.

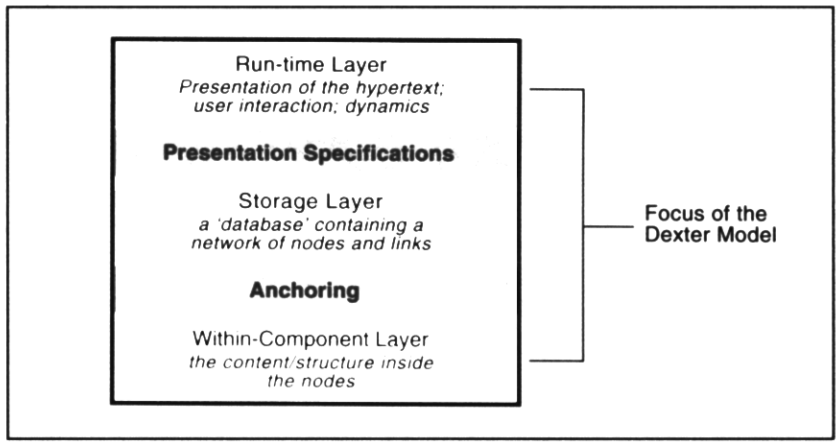
## Simple Storage Layer Model

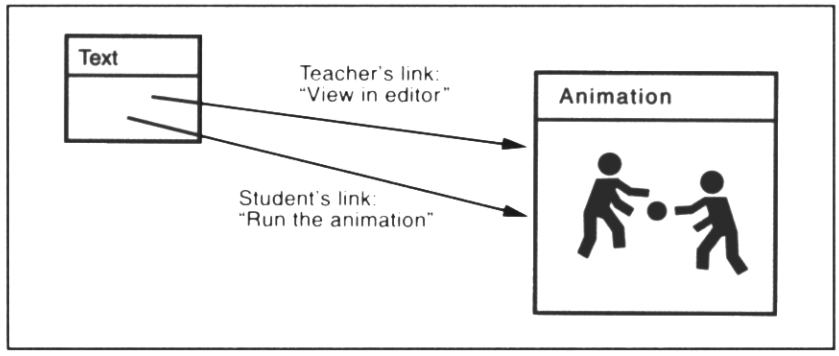
The storage layer describes the structure of a hypertext as a finite set of components together with two functions, a *resolver* function and an *accessor* function. The accessor and resolver functions are jointly responsible for "retrieving" components (i.e., mapping specifications of components into the components themselves).

The fundamental entity and basic unit of addressability in the storage layer is the *component*. A component is either an atom, a link, or a composite entity made from other components. Atomic components are primitive in the (storage layer of the) model. Their substructure is the concern of the within-components layer. Atomic components are what is typically thought of as "nodes" in a hypertext system (e.g., *cards* in NoteCards, *frames* in KMS, *documents* in Intermedia, *statements* in Augment). Links are entities that represent relations between other components. They are basically a sequence of two or more "end-point specifications" each of which refers to (a part of) a component in the hypertext. The structure of links will be detailed later. Composite components are constructed out of other components. The composite component hierarchy created when one composite component contains another composite is restricted to be a directed-acyclic graph (DAG), which means a component can be a subcomponent of multiple composites, and no composite may contain itself either directly or indirectly. Composite components are relatively rare in the current generation of hypertext systems. One exception is the Augment system, in which a document is a tree-structured composition of atomic components called statements.

Every component has a globally unique identity which is captured by its unique identifier (UID). UIDs are primitive in the model, but they are assumed to be uniquely assigned to components across the entire universe of discourse (not just within the context of a single hypertext). The accessor function of the hypertext is responsible for "accessing" a component given its UID (i.e., for mapping a UID into the component "assigned" that UID).

Utilizing UIDs provides a guaranteed mechanism for addressing any component in a hypertext. But the use of UIDs as a basic addressing mechanism in hypertext may be too restrictive. For example, it is possible in the Augment system to create a link to "the statement containing the word 'pollywog.'" The statement specified by this link may not exist or



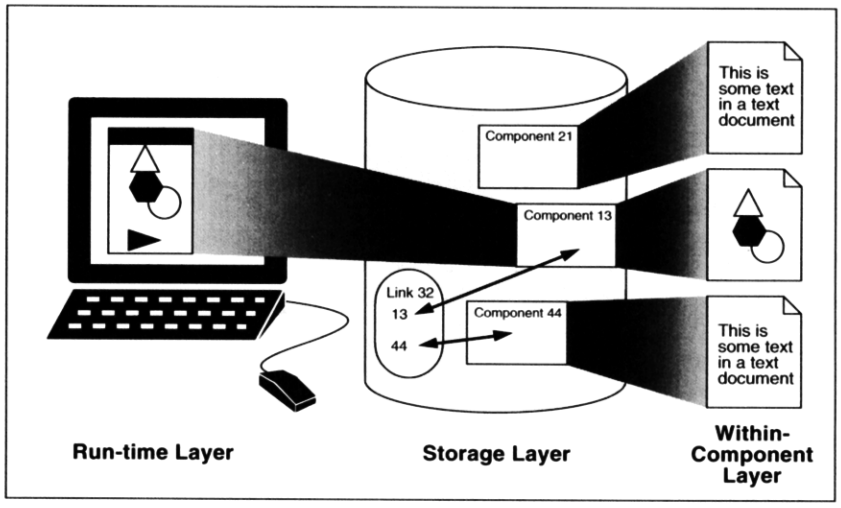




**Figure 1.** Layers of the Dexter model

**Figure 2.** Illustration of the need for presentation specifications on the access path (i.e., links) as well as on the components themselves

**Figure 3.** A depiction of the three layers of the Dexter model as embedded in an actual hypertext system

it may change over time as documents are edited. Therefore, the link cannot rely on a specific statement UID to address the target statement. Rather, when the link is followed, the specification must be "resolved," if possible, to a UID (or set of UIDs) which then can be used to access the correct component(s).

This kind of indirect addressing is supported in the storage layer, using *component specifications* together with the resolver function. The resolver function is responsible for "resolving" a component specification into a UID, which can then be fed to the accessor function to retrieve the specified component. Note, however, that the resolver function is only a partial function. A given specification may not be resolvable into a UID (i.e., the component being specified may not exist). However, it is the case that for every component there is at least one specification that will resolve to the UID for that component. In particular, the UID itself may be used as a specifier, in which case the resolver function is the identity function.

Implementing span-to-span links (e.g., in Intermedia) requires more than simply specifying entire components. Span-to-span linking depends on a mechanism for specifying substructure within components. But in order to preserve the boundary between the hypertext network *per se* and the content/structure within the components, this mechanism cannot depend in any way on knowledge about the internal structure of (atomic) components. In the Dexter model, this is accomplished by an indirect addressing entity called an *anchor*. An anchor has two parts: an *anchor id* and an *anchor value*. The anchor value is an arbitrary value that specifies some location, region, item, or substructure within a component. This anchor value is interpretable only by the applications responsible for handling the content/structure of the component. It is primitive and unrestricted from the viewpoint of the storage layer. The anchor id is an identifier which uniquely identifies its anchor within the scope of its component. Anchors can therefore be uniquely identified across the whole universe by a component UID-anchor id pair.

The two-part composition of an anchor is designed to provide a fixed point of reference for use by the storage layer, the anchor id, combined with a variable field for use by the within-component layer, the anchor value. As a component changes over time (e.g., when it is edited within the run-time layer), the within-component application changes the anchor value to reflect changes to the internal structure of the component or to reflect within-component movement of the point, region, or items to which the anchor is conceptually attached. The anchor id, however, remains constant, providing a fixed referent that can be used to specify a given structure within a component.[6]

The mechanism of the anchor id can be combined with the component specification mechanism to provide a way of specifying the end points of a link. In the model, this is captured by an entity called a *specifier*, which consists of a component specification, an anchor id, and two additional fields: a *direction* and a *presentation specification*. A specifier specifies a component and an anchor 'point' within a component that can serve as the endpoint of a link. The direction encodes whether the specified end point is to be considered a source of a link, a destination of a link, both a source and a destination, or neither a source nor destination. (These are encoded by direction values of FROM, TO, BIDIRECT, and NONE, respectively.) The presentation specification is a primitive value that forms part of the interface between the storage layer and the run-time layer. The nature and use of presentation specifications will be discussed in conjunction with the run-time layer.

It is now possible to describe the structure of link components a bit more precisely. In particular, a link is simply a sequence of two or more specifiers. Note that this provides for links of arbitrary arity, despite the fact that binary links are standard in existing hypertext systems. Directional links, also standard in existing systems, are handled using the direction field in the specifier. In this way, what appear to be one-way links, such as HyperCard buttons, can be modeled as two-way links with the button end having a DIRECTION with value NONE and the other end having a DIRECTION with value TO. In the most general model, duplicate specifiers are allowed. The only constraint is that at least one specifier have a direction of TO or BIDIRECT.

In the foregoing discussion, components were described as being either atoms, links, or compositions of other components. In actuality, this describes what the model calls a *base component*. In contrast, *components* in the model are complex entities that contain a base component together with some associated *component information*. The component information describes the properties of the component other than its 'content.'[7] Specifically, the component information contains a sequence of anchors that index into the component, a presentation specification that contains information for the run-time layer about how the component should be presented to the user, and a set of arbitrary attribute/value pairs. The attribute/value pairs can be used to attach any arbitrary property (and its value) to a component. For example, keywords can be attached to a component using multiple 'keyword' attributes. Similarly, a component type system can be implemented in the model by adding to each component a 'type' attribute with an appropriate type specification as its value.[8]

In addition to a data model, the storage layer defines a small set of operations that can be used to access and/or modify a hypertext. All of these operations are defined in a way that will maintain the invariants of the hypertext (e.g., the fact that the

[6]Because a link is a kind of component, this discussion suggests the possibility of modeling not just links whose end points are other links, but a link end point anchored *within* another link. If we consider a link's content to be its specifiers, this suggests the idea of anchoring, say, a textual annotation of a link in one or more of its specifiers. This might be useful in collaborative situations in which part of the link structure itself becomes the focus of a hypermedia-based discussion.

[7]The *Component-Info* block for Link #9981 is not shown in Figure 4.

[8]For example, link "types" like those supported in NoteCards can be implemented in this way.

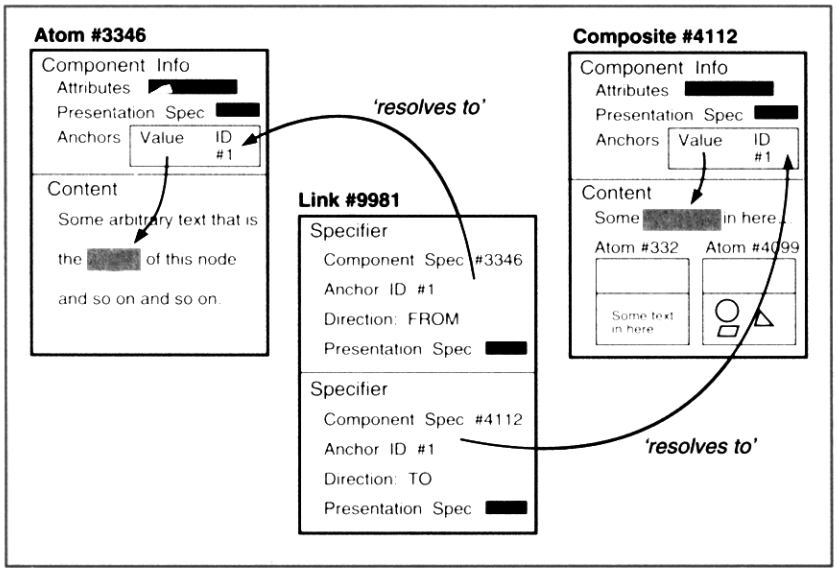



**Figure 4.** Overall organization of the storage layer including specifiers, links, and anchors. The five components shown include three atomic components, one composite component (constructed from two of the atomic components plus some text), and one link component representing a connection from the anchor in atomic component #3346 to the anchor in composite component #4112.

composition hierarchy of components/subcomponents is acyclic). The operations defined in the model include adding a component (atomic, link or composite) to a hypertext, deleting a component from the hypertext, and modifying the contents or ancillary information (e.g., anchors or attributes) of a component. There are also operations for retrieving a component given its UID or any specifier that can be resolved to its UID. Finally, there are operations that help determine the interconnectivity of the network structure. *LinksTo*, given a hypertext and the UID of a component in the hypertext, returns the UIDs of links resolving to that component. *LinksToAnchor* returns the set of link UIDs that refer to an anchor when given the anchor identifier, its containing component's UID, and a hypertext.

Maintaining the invariants of the hypertext requires enforcing the following constraints:

- The *accessor* function must be an invertible mapping from UIDs to components. This implies that every component must have a UID.
- The *resolver* function must be able to produce all possible valid UIDs.
- There are no cycles in the component/subcomponent relationship, that is, no component may be a subcomponent (directly or transitively) of itself.
- The anchor ids of a component must be the same as the anchor ids of the link specifiers resolving to the component.
- The hypertext must be *link-consistent:* that is, the component specifiers of every link specifier must resolve to extant components. (Link-consistency requires, for example, that

```
<hypertext>
    <component>
        <type> text </type>
        <uid> 21 </uid>
        <data> This is some text … </data>
        <anchor>
            <id> 1 </id>
            <location> d13 </location>
        </anchor>
    </component>
    <component>
        <type> text </type>
        <uid> 777 </uid>
        <data> This is some other text … </data>
        <anchor>
            <id> 1 </id>
            <location> 13-19 </location>
        </anchor>
    </component>
    <component>
        <type> link </type>
        <uid> 881 </uid>
        <specifier>
            <component_uid> 21 </component_uid>
            <anchor_id> 1 </anchor_id>
            <direction> FROM </direction>
        <\specifier>
        <specifier>
            <component_uid> 777 </component_uid>
            <anchor_id> 1 </anchor_id>
            <direction> TO </direction>
        <\specifier>
    </component>
</hypertext>
```

**Figure 5.** Example of a simple interchange format derived from the model

**Table 1.** Storage layer functions and run-time layer operations

| Storage layer functions | |
|---|---|
| CreateComponent | Creates a new component and adds it to the hypertext. Ensures that the range of the accessor function is extended to include the new component. The resolver function is also extended so that there is at least one specifier for the new component's corresponding UID. |
| CreateAtomicComponent | Takes an atom and a presentation specification and uses CreateComponent to create a new atomic component. |
| CreateLinkComponent | Takes a link and a presentation specification and uses CreateComponent to create a new link component. |
| CreateCompositeComponent | Takes a collection of base components and a presentation specification and uses CreateComponent to create a new composite component. |
| CreateNewComponent | Invoked from the run-time layer, calls one of CreateAtomicComponent, CreateLinkComponent, or CreateCompositeComponent. |
| DeleteComponent | Deletes a component, ensuring that any links whose specifiers resolve to that component are removed. |
| ModifyComponent | Modifies a component, ensuring that its associated information remains unchanged, that its type (atom, link, or composite) remains unchanged, and that the resulting hypertext remains link consistent. |
| GetComponent | Takes a UID and uses the accessor function to return a component. If the UID represents a link component, returns either a source or destination specifier for that component. |
| Attribute Value | Takes a component UID and an attribute and returns the value of the attribute. |
| SetAttribute Value | Takes a component UID, a value and an attribute, and sets the value of the attribute. |
| All Attributes | Returns the set of all component attributes. |
| LinksToAnchor | Takes an anchor and its containing component, and returns the set of links that refer to the anchor. |
| LinksTo | Takes a hypertext and a component UID, and returns the UIDs of links resolving to that component. |

| Run-time layer operations | |
|---|---|
| openSession | Creates a session for a given hypertext. A session begins with no instantiations. |
| openComponents | Opens a set of new instantiations on a given set of components. |
| presentComponent | Takes a specifier and a presentation specification and creates an instantiation for the associated component. |
| followLink | Uses openComponents to present any components referred to by the "TO" specifiers of any links with anchors represented by a given link marker. |
| newComponent | Opens a new instantiation on a newly created component. |
| unPresent | Removes an instantiation. |
| editInstantiation | Edits an instantiation. A call to realizeEdits is required to save the changes. |
| realizeEdits | Saves an instantiation's editing changes to the corresponding component by calling ModifyComponent. |
| deleteComponent | Deletes the component associated with a given instantiation. Also removes any other instantiations for that component. |
| closeSession | Closes a given session. Note that by default, pending changes to instantiations are not saved. |

when deleting a component, any links whose specifiers resolve to that component must also be deleted.)

### Simple Run-time Layer Model

The fundamental concept in the run-time layer is the *instantiation* of a component. An instantiation is a presentation of the component to the user. Operationally, an instantiation should be thought of as a kind of run-time cache for the component. A 'copy' of the component is cached in the instantiation, the user views and/or edits this instantiation, and the altered cache is then 'written' back into the storage layer. Note that there can be more than one simultaneous instantiation for any given component. Each instantiation is assigned a unique (within-session, see the following description) instantiation identifier (IID).

Instantiation of a component also results in instantiation of its anchors. An instantiated anchor is known as a *link marker*. This terminology is congruent with that used in Intermedia, where the term "anchor" refers to an attachment point or region and the term "link marker" refers to the visible manifestation of that anchor in a displayed document. In order to accommodate the link marker notion within the model, an instantiation is actually a complex entity containing a *base instantiation* together with a sequence of link markers and a function mapping link markers to the anchors they instantiate. A base instantiation is a primitive in the model that represents some sort of presentation of the component to the user.

At any given moment, the user of a hypertext can be viewing and/or editing any number of component instantiations. The run-time layer includes an entity called a *session* which serves to keep track of the moment-by-moment mapping between components and their instantiations. Specifically, when a user wants to access a hypertext, he or she opens a session on that hypertext. The user creates instantiations of components in the hypertext by an action called *presenting* the component. The user can edit such an instantiation, modify the component based on accumulated edits to the instantiation (an action known as *realizing* the edits), and finally destroy the instantiation (called *unpresenting* the component). If the user deletes a component through one of its instantiations, then all other instantiations are automatically removed. When the user is finished interacting with the hypertext, the session is closed.

In the model, the session entity contains the hypertext being accessed, a mapping from the IIDs of the sessions's current instantiations to their corresponding components in the hypertext, a history, a run-time resolver function, an instantiator function, and a realizer function. At any given moment, the history is a sequence of all operations carried out since the last open session operation. In the present version of the model, this history is used only in defining the notion of a read-only session. It is intended to be available, however, to any operation that needs to be conditioned on preceding operations.

The session's run-time resolver function is the run-time version of the storage layer's resolver function. Like the resolver, it maps specifiers into component UIDs. The run-time resolver, however, can use information about the current session, including its history, in the resolution process. The storage resolver layer has no access to such run-time information. For example, a specifier may refer to "the most recently accessed component named 'xyzzy'." The run-time resolver is responsible for mapping this specifier into the UID matching this specification. The storage layer resolver would not be able to handle this specification.[9] The run-time resolver is restricted to be a superset of the storage layer resolver function; any specifier that the storage layer resolver can resolve to a UID must be resolved to the same UID by the run-time resolver.

At the heart of the run-time model is the session's *instantiator* function. Input to the instantiator consists of a component (UID) and a presentation specification. The instantiator returns an instantiation of the component as part of the session. The presentation specification is primitive in the model, but is intended to contain information specifying how the component being instantiated is to be "presented" by the system during this instantiation. Note that the component itself has a presentation specification from the storage layer of the model. This presentation specification is meant to contain information about the component's own notion of how it should be presented. It is the responsibility of the instantiator function to adjudicate (by selection or combination or otherwise) between the presentation specification passed to the instantiator and the presentation specification attached to the component being instantiated. The model in its current form does not make this adjudication explicit.

The instantiator function is the core of the *present component* operation. Present component takes a component specifier (together with a session and a presentation specification) and calls the instantiator using the component UID derived from resolving the specifier.[10] Present component in turn is the core of the *follow link* operation. Follow link takes (the IID of) an instantiation together with a link marker contained within that instantiation. It then presents the component(s) that are at the destination end points (say, end points whose specifiers have direction TO or BIDIRECT) of all link(s) that have as an end point the anchor represented by the given link marker. In the case in which all links are binary, this is equivalent to following a link from the link marker for its source. The result of following the link is a presentation of its destination component and anchor.

The instantiator function also has an "inverse" function called the realizer function, which takes an instantiation and returns a (new) component that "reflects" the current state of the instantiation (i.e., including recent edits to the instantiation). This is the basic mechanism for "writing back the cache" after an instantiation has been edited. The component produced by the realizer

[9]This means, for example, that the LinksTo function will not find links whose component specifiers are resolvable only at run time—such links must be captured in the run-time layer.

[10]Because presentComponent has access to the session, the resulting instantiation can additionally depend on the current set of instantiations.

is used as an argument to the storage layer modify-component operation to replace the component with the edited component. This operation is wrapped in the function called *realize edits* in the run-time layer.[11]

## Conformance with the Reference Model

One reason to have a reference model for hypertext is to try to ascertain whether a purported hypertext system actually warrants being called a hypertext system. So, given an actual hypertext system how do we show that it meets, or is conformant with the model? The best guidance for answering this question comes from the VDM experience under the heading of data reification as described, for example, in Cliff Jones's book [13, Chapter 8] on software development using VDM. First, we must exhibit total functions, called *retrieve* functions which map the actual types and functions from the given (actual) hypertext system to each of the types and functions of the model. We must also demonstrate *adequacy*—that there is at least one actual representation for each abstract value. Obviously, the retrieve functions must satisfy the invariants which are given for the data types and functions. An informal way of saying this is that everything which is expressible or realizable in the model must be expressible or realizable in the actual system.

In actuality our model is much more powerful than necessary. In particular

- By admitting multiway links and links to links in the model, we put a fairly heavy burden on any implementation.
- Many hypertext systems do not have the notion of composites.
- Some hypertext systems, such as KMS, do not have links with both an explicit source and destination. Thus requiring discrimination among all the values of type DIRECTION is too much.

We are currently working on a "minimal" model that addresses these and other concerns.

[11]RealizeEdits serves as the hypertext's Save operation.

## Conclusion

Development of the Dexter model is still in its very early stages. The model as currently stated is far more powerful than any existing hypertext system. The provisions for *n*-ary links and for composite components, for example, are intended to accommodate the design of future hypertext systems. No existing system that we have examined includes both *n*-ary links and composite components. The result is that no existing system 'conforms to' the model in the sense that it supports all of the mechanisms the model supports. The solution to this problem is to make some mechanisms 'optional,' resulting in a family of interrelated models that support differing sets of optional mechanisms. The weakest model, for example, would have no composites and only binary links. The strongest model would be the Dexter model in the present form. Conformance to the model could then be conditioned on the exact set of mechanisms supported. Systems would be compared on the basis of the set of mechanisms they do support.

A related issue involves a number of consistency restrictions imposed by the present model. For example, when creating a link the model requires that all of its specifiers resolve to existing components. This restriction prevents the creation of links that are 'dangling' from the outset, and adequately represents the consistency guarantee of KMS. But it is overly restrictive for Augment, which allows creation of initially dangling links. Restrictions of this sort will have to be made optional in the model, just as they will with mechanisms. Conformance to the model can then be conditioned on appropriate choices of restrictions. Systems can be compared on the basis of the set of restrictions they enforce, in the same way mechanisms can.

The model has yet to be compared in detail to the hypertext systems it is designed to represent. Clearly, a necessary step in the development of the model is to formally specify (in Z) the architecture and operation of a number of 'reference' hypertext systems using the constructs from the Dexter model. These reference systems should be chosen to represent a broad spectrum of designs, intended application domains, implementation platforms, and so forth. This enterprise would provide valuable feedback regarding the adequacy and completeness of the model. In particular, it will help assess whether the model provides sufficient mechanisms for representing the important (common) abstractions found in the reference systems. It will also provide feedback on the 'naturalness' of the model (i.e., on whether the specification of the reference systems in Dexter terms feels 'natural' or whether the abstractions found in certain systems must be excessively massaged to fit into the Dexter abstractions).

Despite its early stages of development, the model has already been useful in developing hypertext interchange standards. As described in the panel on interchanging hypertexts at the Hypertext '89 Conference [16], a number of efforts have been started to operationalize the abstractions of the Dexter model in the form of interchange formats. Figure 5 illustrates one such format. This format was used for experimenting with the interchange of hypertexts between NoteCards and HyperCard. The format is a fairly straightforward rendering of the entities found in the Dexter model into a SGML-like syntax. This format is by no means a well-developed interchange standard. But it does suggest that the Dexter model provides a good basis from which to develop such standards. In fact, because the model is an attempt to provide a well-defined and comprehensive model, it is an ideal basis for developing a comprehensive standard for interchanging hypertexts between widely differing systems. ■

**Acknowledgments**

The model described in this article grew out of a series of workshops on hypertext. The following people attended the workshops and were instrumental in the development of the model: Rob Akscyn, Doug Engelbart, Steve Feiner, Frank Halasz, John Leggett, Don McCracken, Norm Meyrowitz, Tim Oren, Amy Pearl, Catherine Plaisant, Mayer Schwartz, Randall Trigg, Jan Walker, and Bill

Weiland. The workshops were organized by Jan Walker and John Leggett. Guest editors Kaj Grønbæk and Randall Trigg are grateful to John Leggett and Mayer Schwartz for their comments on earlier drafts of this revision.

**References**


1. Akscyn, R., McCracken, D.L., and Yoder, E.A. KMS: A distributed hypertext for managing knowledge in organizations. *Commun. ACM 31,* 7 (July 1988), 820–835.
2. Campbell, B. and Goodman, J.M. HAM: A general purpose hypertext abstract machine. *Commun ACM 31,* 7 (July 1988), 856–861.
3. Conklin, J. Hypertext: A survey and introduction. *IEEE Comput. 20,* 9 (1987), 17–41.
4. Delisle, N. and Schwartz, M. Neptune: A hypertext system for CAD applications. In *Proceedings of ACM SIGMOD '86* (Washington, D.C., May 28–30, 1986), pp. 132–142.
5. Englebart, D.C. Authorship provisions in Augment. In *Proceedings of the IEEE COMPCON* (Spring 1984), pp. 465–472.
6. Englebart, D.C. Collaboration support provisions in Augment. In *OAC Digest: Proceedings of the 1984 AFIPS Office Automation Conference* (Los Angeles, Calif. Feb. 20–22, 1984), pp. 51–58.
7. Feiner, S., Nagy, S. and van Dam, A. An experimental system for creating and presenting interactive graphical documents. *ACM Trans. Graph. 1,* 1 (1982), pp. 59–77.
8. Goodman, D. *The Complete HyperCard Handbook.* Bantam Books, New York, 1987.
9. Halasz, F.G. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Commun. ACM 31,* 7 (July 1988), 836–855.
10. Halasz, F.G., Moran, T.P. and Trigg, R.H. NoteCards in a nutshell. In *Proceedings of the 1987 ACM Conference of Human Factors in Computer Systems (CHI+GI '87)* (Toronto, Ontario, Apr. 5–9, 1987), pp. 45–52.
11. *Proceedings of Hypertext 87,* Chapel Hill, N.C., Nov. 13–15, 1987. Available from ACM Press, order number 608892.
12. *Proceedings of Hypertext 89,* Pittsburgh, Pa, Nov. 5–8, 1989. Available from ACM Press, order number 608891.
13. Jones, C.B. *Systematic Software Development Using VDM.* Prentice-Hall International, Hertfordshire, England, 1986.
14. Lange, D.B. A formal approach to hypertext using post-prototype formal specification. Dept. of Computing Science, Technical University of Denmark, Oct. 31, 1989.
15. Meyrowitz, N. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '86)* (Portland, Ore., Sept. 29–Oct. 2, 1986), pp. 186–201.
16. Oren, T. Panel: Interchanging hypertexts. In *Proceedings of Hypertext 89,* (Pittsburgh, Pa, Nov. 5–8, 1989), pp. 379–381.
17. Pearl, A. Sun's Link Service: A protocol for open linking. In *Proceedings of Hypertext 89* (Pittsburgh, Pa, Nov. 5–8, 1989), 137–146.
18. Shneiderman, B. *Hypertext on Hypertext.* Addison-Wesley, New York, 1989.
19. Spivey, J.M. *The Z Notation.* Prentice-Hall International, Hertfordshire, England, 1989.
20. Trigg, R.H. and Weiser, M. TEXTNET: A network-based approach to text handling. *ACM Trans. Off. Inf. Syst. 4,* 1 (1986), 1–23.
21. Walker, J. Document Examiner: Delivery interface to hypertext documents. In *Proceedings of Hypertext 87,* Chapel Hill, N.C., Nov. 13–15, 1987, pp. 307–323.
22. Walker, J. Supporting document development with Concordia. *IEEE Comput. 21,* 1 (1988), 41–59.
23. Yankelovich, N., Haan, B., Meyrowitz, N., and Drucker, S. Intermedia: The concept and the construction of a seamless information environment. *IEEE Comput. 21,* 1 (1988), 81–96.


**CR Categories and Subject Descriptors:** E.1 [**Data Structures**]: Hypertext; H.1.1 [**Models and Principles**]: General Systems Theory; H.1.2 [**Models and Principles**]: User/Machine Systems—*human information processing*; H.2.1 [**Database Management**]: Logical Design—*hypertext*; H.3.2 [**Information Storage and Retrieval**]: Information storage—*hypertext*; H.4.2 [**Information Systems Applications**]: Types of Systems—*hypermedia*

**General Terms:** Design

**Additional Key Words and Phrases:** Composites, data and process model, hypermedia, hypertext, open hypermedia

**About the Authors:**

**FRANK HALASZ** is vice president of software and systems at Group Communications, the Xerox Document Conferencing Unit. Current research interests include computer-supported cooperative work and hypermedia. **Author's Present Address:** Group Communications, The Xerox Document Conferencing Unit, 2225 East Bayshore Rd., Suite 2000, Palo Alto, CA 94303; email: halasz@parc.xerox.com

**MAYER SCHWARTZ** is a principal engineer at Tektronix Laboratories. Current research interests include networking and operating systems support for distributed digital video applications. **Author's Present Address:** Tektronix, Inc., Mail Stop 50-370, P.O. Box 500, Beaverton, OR 97077; email: Mayer.Schwartz@tek.com