The Dexter Hypertext Reference Model

Frank G. Halasz

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304
halasz@xerox.com

Mayer Schwartz

Tektronix Labs
P.O. Box 500, MS 50-662
Beaverton, OR 97077
mayers@tekchips.labs.tek.com

Presented at the NIST Hypertext Standardization Workshop,
Gaithersburg, MD, January 16-18, 1990

Abstract

This paper presents the Dexter hypertext reference model. The Dexter model is an attempt to capture, both formally and informally, the important abstractions found in a wide range of existing and future hypertext systems. 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 runtime 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 as well as on the mechanisms of anchoring and presentation specification that form the interfaces between the storage layer and the within-component and runtime layers, respectively. The model is formalized in the specification language Z [Spiv89], a specification language based on set theory. The paper briefly discusses the issues involved in comparing the characteristics of existing systems against the model.

Introduction

What do hypertext[1] systems such as NoteCards [Hala88], Neptune [Deli86a], KMS [Aksc88a], Intermedia [Yank88a] and Augment [Enge84a] 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. Yet these systems differ markedly in the specific data models and sets of functionality that 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, is there anything common between these systems in their notions of hypertext nodes?

In an attempt to provide a principled basis for answering these questions, this paper 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. Thus, the Dexter model serves as a standard against which to compare and contrast the characteristics and functionality of various hypertext (and non-hypertext) 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 paper was initiated as the result of two small workshops on hypertext. The first workshop was held in October, 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[2]. A large part of the discussion at these workshops was the elicitation of the abstractions common to the major hypertext systems. The Dexter model is an attempt to capture, fill-out, 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 turned out to be an extremely difficult task, especially so in the absence of an understanding of the common (and differing) abstractions among the various systems. The term "node" turned out to be especially difficult 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", prefering neutral terms such as "component" for the abstraction in the model.

In the present paper, the Dexter model is formulated in Z [Spiv89], a formal specification language based on typed set theory. The use of Z provides a rigorous basis for defining the necessary abstractions and for discussing their use and interrelationships. Although an understanding of the Z language is a prerequisite for fully understanding the details of the Dexter model as described in this paper, the paper attempts to provide a complete description of the model in the prose accompanying the formal specification. Readers unfamiliar with Z should be able to gain a full, if not precisely detailed, understanding of the model.

This paper also refers in passing to architectural concepts found in a number of existing hypertext systems including Augment [Enge84a], Concordia/Document Examiner [Walk88a], Hypercard [Good87], Hyperties [Shne89d], IGD [Fein82], Intermedia [Yank88a], KMS [Aksc88a], Neptune/HAM [Deli86a], NoteCards [Hala88], the Sun Link Service [Pear89], and Textnet [Trig86a]. The reader is assumed to be familiar with the general characteristics and functionality of these systems. Appropriate background material on these systems can be found in Conklin [Conk87a] and in the proceedings of the Hypertext 87 [HT87] and Hypertext 89 [HT89] conferences.

This paper is divided in 4 main sections. The first section provides a brief discursive overview of the entire model. The second section describes the storage layer of the model, both formally and informally. The third section describes the runtime layer of the model in a similar manner. The final section discusses issues involved in comparing existing systems against the model.

1. An Overview of the Model

The Dexter model divides a hypertext system into three layers, the *runtime* 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 storage layer describes a ‘database’ that is composed of a hierarchy of data-containing ‘‘components’’ which are 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, etc. that form the basic content in the hypertext network.

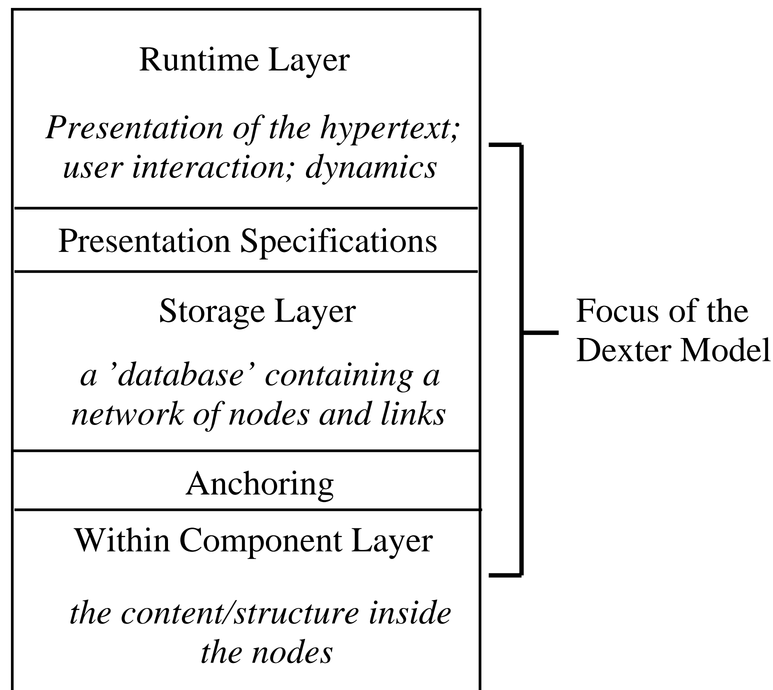


Figure 1: Layers of the Dexter model.

The storage layer focuses on the mechanisms by which the components and links are ‘‘glued together’’ to form hypertext networks. The components are treated in this layer as generic containers of data. No attempt is made to model any structure within the container. Thus, the storage layer makes no differentiation between text components and graphics components. Nor does it 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 open-ended. Text, graphics, animations, simulations, images, and many more types of data have been used as components in existing hypertext systems. It would be folly 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, etc) will be used in conjunction with the Dexter model to capture the entirety of the hypertext, including the with-component content and structure.

An extremely critical piece of the Dexter model, however, is the interface between the hypertext network and the within-component content and structure. The hypertext system requires a mechanism for addressing (refering to) locations or items *within* the content of an individual component. In the Dexter model, this mechanism is know as *anchoring*. The anchoring mechanism is necessary, for example, to support span-to-span links such as are 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. Anchors are a mechanism that provides this functionality while maintaining a clean 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 runtime layer of the model. As in the case of within-component structure, the range of possible tools for accessing, viewing, and manipulating a 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 it 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 runtime 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 runtime layer), but can also be a property of the component itself and/or of the access path (link) taken to that component.

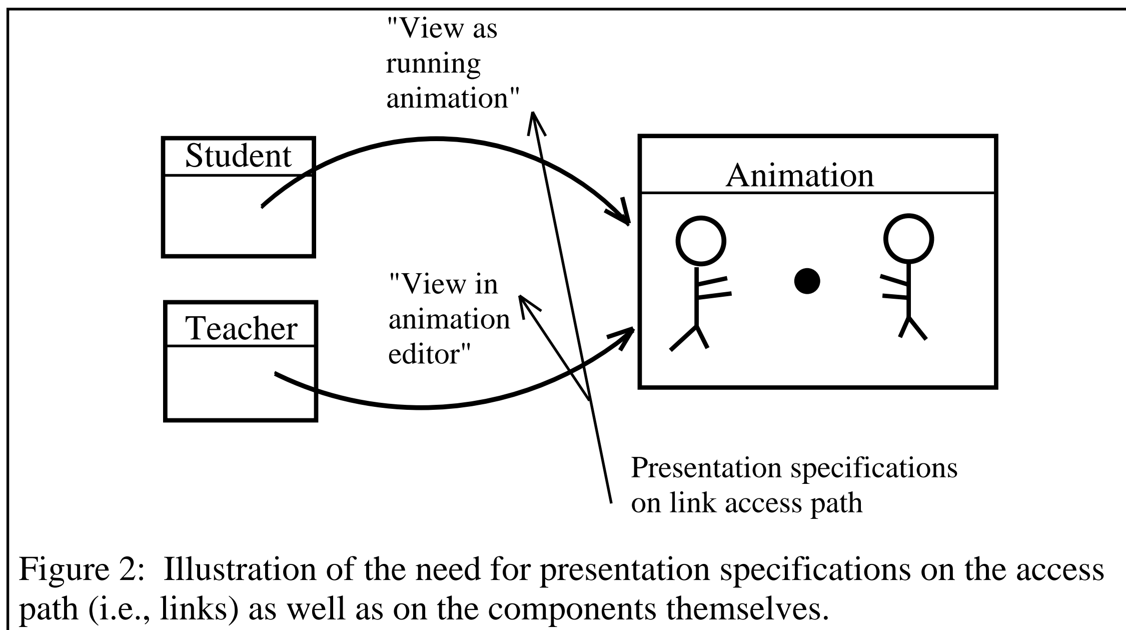


Figure 2: Illustration of the need for presentation specifications on the access path (i.e., links) as well as on the components themselves.

Figure 2 illustrates the importance of the presentation specifications mechanism. In this figure, there is an animation component taken from a computer-based training hypertext. This animation component can be accessed from two other components, a ‘‘teacher’’ component and a ‘‘student’’ component. When following the link from the student component, the animation should be brought up as a running animation. In contast, when coming from the teacher component, the animation should be brought up in editing mode ready to be altered. In order to separate these two cases, the runtime 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, it is an interface that allows the storage layer to communicate in generic way with the runtime 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 an typical hypertext system. The figure depicts a 3 node/1 link hypertext network. The storage layer contains four entities: the three components (i.e., nodes) and the link. The actual contents (text and graphics) for the components are located to the right of the storage layer in the within-components layer. In the runtime layer, the single graphics component is being presented to the user. The link emanating from this node is marked by an arrowhead located near the bottom of the node’s window on the computer screen.

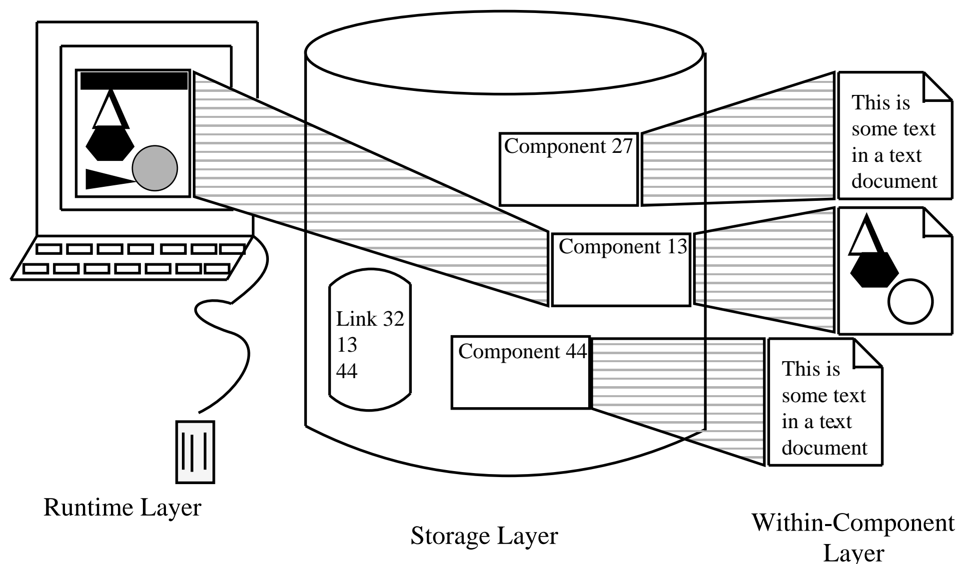


Figure 3: A depiction of the three layers of the Dexter model as embedded in an actual hypertext system.

2. Simple Storage Layer Model

2.1 An Overview of the Storage Layer

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 addressability in the storage layer is the component. A component is either an atom, a link, or a composite entity made up 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 a ‘‘node’’ in a hypertext system, e.g., a *card* in NoteCards, a *frame* in KMS, a *document* in Intermedia, a *statement* in Augment. Links are entities that represent relations between other components. They are basically a sequence of 2 or more ‘‘endpoint specifications’’ each of which refers to (a part of) a component in the hypertext. The structure of links will be detailed below. 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 direct-acyclic graph (DAG), i.e., no composite may contain itself either directly or indirectly. Composite components are relative rare in the current generation of hypertext systems. One exception is the Augment system where 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.

UIDs provide 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 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’’ to a UID (if possible), which then can be used to access the correct component.

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 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 runtime layer), the within-component application will change 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 is, however, will remain constant, providing a fixed referent that can be used to specify a given structure within a component.

The mechanism of the anchor id can be combined with the component specification mechanism to provide a way of specifying the endpoints 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 endpoint 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 a destination. (These are encoded by direction values of FROM, TO, BIDIRECT, and NONE, respectively.) The present specification is a primitive value that forms part of the interface between the storage layer and the runtime layer. The nature and use of present specifications will be discussed in conjunction with the runtime layer below.

Returning to the issue of link components, it is now possible to describe their structure a bit more precisely. In particular, a link is simply a sequence of 2 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.

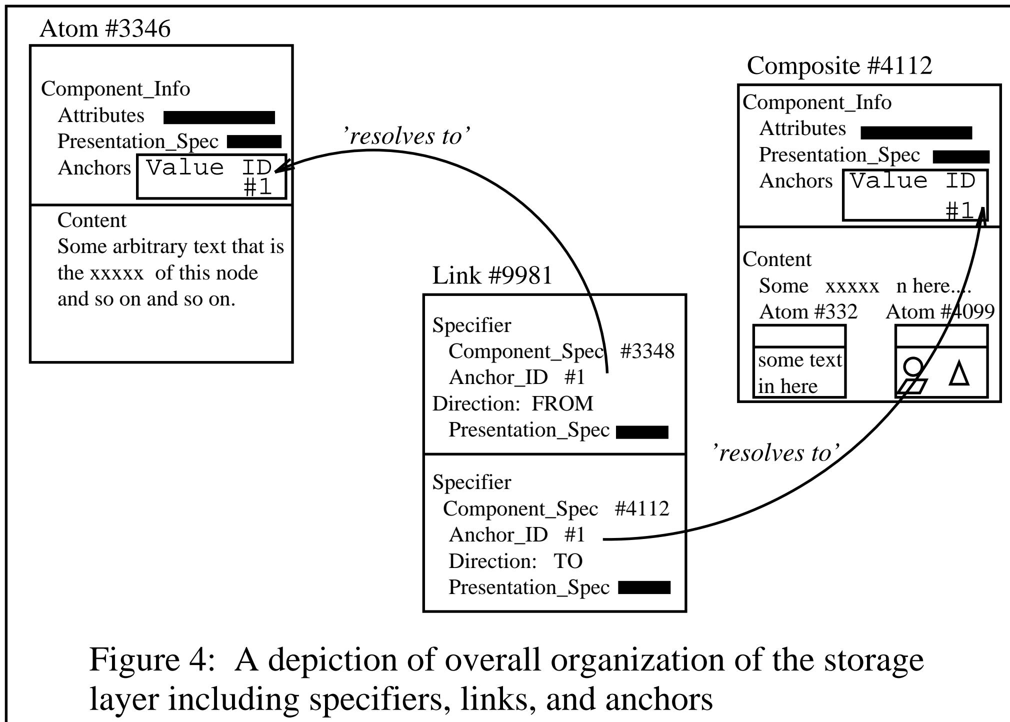


Figure 4: A depiction of overall organization of the storage layer including specifiers, links, and anchors

Figure 4 depicts the overall organization of the storage layer including specifiers, links, and anchors. The figure depicts 5 components including 3 atomic components, 1 composite component (that constructed from two of the atomic components plus some text), and 1 link component that represents a connection from the anchor (i.e., span) within an atomic component (#3346) to the anchor (span) in the composite component (#4112).

In the foregoing discussion, components were described as being either an atom, a link, or a composition 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'. Specifically, the component information contains a sequence of anchors that index into the component, a present specification that contains information for the runtime 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 mutiple '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.

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 such a way as to maintain the invariants of the hypertext, e.g., the fact that the composition hierarchy of components/sub-components 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 ancilliary 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 is one operation needed for determining the interconnectivity of the network structure. This operation, *linksToAnchor*, returns the set of links that refer to an anchor when given the anchor and its containing component.

2.2 Formalization of the Storage Layer

As described, we envision a hypertext system consisting of a set of components, each of which has a UID from the given set *UID*.

[UID]

Retrieving a component involves finding its UID and then using that UID to get hold of the actual component; this is accomplished by means of an *accessor* function which returns a component given its UID. UIDs are normally not meant to be visible to clients of a hypertext system. Given a component specification, it may be possible to find the UID to which the component specification refers, by means of a *resolver* function. Component specifications arise from the given set *COMPONENT _SPEC*. We also have a description for the visual presentation (present spec) of a component, which as part of a component is used in the run-time layer but not in the storage layer; these visual descriptions come from the given set *PRESENT _SPEC*.

[COMPONENT _SPEC, PRESENT _SPEC]

Links are an important kind of component and are supported in every hypertext system. Directionality is sometimes important for links, while at other times is immaterial. We introduce *DIRECTION* as a free type to model respectively the end of a link as a source, as a destination, as both a source and destination, or as neither.

DIRECTION ::= FROM | TO | BYDIRECT | NONE

The schema type *SPECIFIER* essentially takes the form of the description of one end of a "link." This description is sometimes sufficient to determine the UID of the component at one end of a link. As described in the overview, anchoring plays an important part in the model. Anchors are identified by means of a unique (to a component) anchor id from the given set *ANCHOR_ID*. Anchor values come from the given set ANCHOR_VALUE. Anchors are then just pairs of anchor id and associated anchor value.

[ANCHOR_ID, ANCHOR_VALUE]

ANCHOR == ANCHOR_ID x ANCHOR_VALUE

A value of type *SPECIFIER* describes a single end of a link. We include the variable *presentSpec* in the *SPECIFIER* schema so we can model different ways of visually showing links as we follow them (based on the specifier used), as illustrated in the example shown in Figure 2.

```
┌─SPECIFIER──────────────────────────
│ componentSpec: COMPONENT _SPEC
│ anchorSpec: ANCHOR _ID
│ presentSpec: PRESENT _SPEC
│ direction: DIRECTION
└────────────────────────────────────
```

Links must include at least two specifiers. What apprear 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 have a *DIRECTION* with the value *TO*. The two *specifiers* link constraint simplifies the hypertext model. On the other hand there is no reason not to have multi-way links, and also the model accomodates them. In the most general model, duplicate specifiers are allowed. The only constraint is that at least one specifier have a direction of *TO*.

```
┌─LINK───────────────────────────────────────
│ specifiers: seq SPECIFIER
├──────────────────
│ #specifiers ≥ 2
│ ∃ s : ran specifiers · s.direction = TO
└────────────────────────────────────────────
```

A base component (a generalization of the traditional "node" or "link") of a hypertext can either be

- an atomic element which is modeled by the given type *ATOM*,

[*ATOM*]

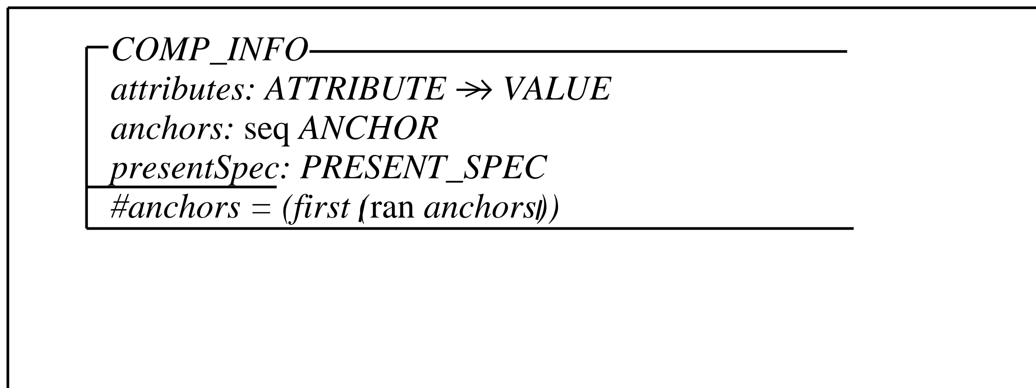
models a "node" of a typical hypertext system but with the internal detail omitted.

- a *link* which is modeled by the *LINK* schema given above, or
- a *composite* which can be described recursively as a sequence of base components.

Components can have ancillary information associated with them, such as attribute/value pairs, anchors, or presentation information. Most hypertext systems allow for attributes of components. These attributes can be thought of as attribute/value pairs which can be modeled as a partial function mapping attributes to values. We thus introduce two additional given sets, one for the set of attribute names and the other for the set of possible values:

[*ATTRIBUTE, VALUE*]

The additional information associated with a base component, which was mentioned above, can be captured in the following schema. We include the invariant that anchor ids are unique within a given component, i.e., the number of anchors within a component is equal to the size of the set of (different) anchors within the component.



Note that a *presentSpec* always has some value. We introduce the function *minInfo* which returns an instance of this schema with "minimal information," that is, no attributes, no anchors and a *presentSpec* which is given as an argument.

$$
\begin{array}{l}
minInfo: PRESENT_SPEC \rightarrow COMP_INFO \\
\hline
\forall ps: PRESENT_SPEC \bullet \\
\quad minInfo(pa) = (\mu\ info : COMP_INFO | \\
\qquad info.attribute = \emptyset \wedge \\
\qquad info.anchors = <> \wedge \\
\qquad info.presentSpec = ps)
\end{array}
$$

We use the recursive type, *BASE_COMPONENT*, to describe the base components of a hypertext system.

$$
\begin{array}{l}
BASE_COMPONENT ::= atom \langle\langle ATOM \rangle\rangle \\
\qquad\qquad\qquad\qquad | \ link \langle\langle LINK \rangle\rangle \\
\qquad\qquad\qquad\qquad | \ composite \langle\langle seq\ BASE_COMPONENT \rangle\rangle
\end{array}
$$

Finally, the schema *COMPONENT* represents a base component along with its associated information.

$$
\begin{array}{l}
COMPONENT \\
\hline
compBase: BASE_COMPONENT \\
compInfo: COMP_INFO \\
\hline
\end{array}
$$

The functions defined in the remainder of this section are there just to make the specification of the model easier to read and understand -- they are not meant to have any particular significance in their own right. The following function builds a component given its base component and associated information.

$$
\begin{array}{l}
component : BASE_COMPONENT \times COMP_INFO \\
\qquad\qquad \rightarrow COMPONENT \\
\hline
component = (\lambda b: BASE_COMPONENT; i : COMP_INFO \bullet \\
\quad (\mu c: COMPONENT | \\
\qquad c.compBase = b \wedge \\
\qquad c.compInfo = i))
\end{array}
$$

The following two functions extract respectively the base component and associated information of a component.

$$
\begin{array}{|l}
base: COMPONENT \to BASE_COMPONENT \\
info: COMPONENT \to COMP_INFO \\
\hline
\forall c: COMPONENT \bullet \\
\quad base(c) = c.compBase \land \\
\quad info(c) = c.compInfo
\end{array}
$$

We introduce three predicates (prefix relations) which are respectively true iff a component is an atom, a link, or a composite.

$$
\begin{array}{|l}
\mathrm{isAtom_} : \mathbb{P}\ COMPONENT \\
\mathrm{isLink_} : \mathbb{P}\ COMPONENT \\
\mathrm{isComposite_} : \mathbb{P}\ COMPONENT \\
\hline
\forall c : COMPONENT \bullet \\
\quad \mathrm{isAtom}\ c \Leftrightarrow base(c) \in \mathrm{ran}\ atom \land \\
\quad \mathrm{isLink}\ c \Leftrightarrow base(c) \in \mathrm{ran}\ link \land \\
\quad \mathrm{isComposite}\ c \Leftrightarrow base(c) \in \mathrm{ran}\ composite
\end{array}
$$

We also define a "type" consistency relationship between components -- that is, two components are "type consistent" is they are both atoms, both links, or both composites.

$$
\begin{array}{|l}
\mathrm{_typeConsistent_}: COMPONENT \leftrightarrow COMPONENT \\
\hline
\forall c_1, c_2 : COMPONENT \bullet \\
\quad c_1\ \mathrm{typeConsistent}\ c_2 \Leftrightarrow \\
\quad (\mathrm{isAtom}\ c_1 \land \mathrm{isAtom}\ c_2) \lor \\
\quad (\mathrm{isLink}\ c_1 \land \mathrm{isLink}\ c_2) \lor \\
\quad (\mathrm{isComposite}\ c_1 \land \mathrm{isComposite}\ c_2)
\end{array}
$$

Because link components are referred to quite frequently in what follows, we introduce the schema *LinkComp* so we can define variables of that type.

$$
\begin{array}{|l}
\hline
\textit{LinkComp} \\
\hline
\textit{COMPONENT} \\
\hline
\textit{compBase} \in \operatorname{ran} \textit{link} \\
\hline
\end{array}
$$

We also introduce some helpful functions to extract the various parts that make up a base component type. The first two functions are only defined for link components and return respectively the set of component specs for the link and the set of anchor ids for the link.

$$
\begin{array}{|l}
componentSpecs : LinkComp \twoheadrightarrow \mathbb{F}\ COMPONENT_SPEC \\
anchorSpecs : LinkComp \twoheadrightarrow \mathbb{F}\ ANCHOR_ID \\
\hline
\forall c : LinkComp \bullet \\
\quad componentSpecs(c) = \{cs : COMPONENT_SPEC \mid \\
\qquad \exists s : \operatorname{ran}(link\sim(base(c))).specifiers \bullet \\
\qquad\quad cs = s.componentSpec\} \land \\
\quad anchorSpecs(c) = \{as : ANCHOR_ID \mid \\
\qquad\quad \exists s : \operatorname{ran}(link\sim(base(c))).specifiers \bullet \\
\qquad\qquad as = s.anchorSpec\}
\end{array}
$$

The next two functions are defined for any component and return respectively its attributes and its anchors.

$$
\begin{array}{|l}
attributes: COMPONENT \rightarrow (ATTRIBUTE \twoheadrightarrow VALUE) \\
anchors : COMPONENT \rightarrow \mathbb{F}\ ANCHOR \\
\hline
\forall c : COMPONENT \bullet \\
\quad attributes(c) = (info(c)).attributes \land \\
\quad anchors(c) = \operatorname{ran}(info(c)).anchors
\end{array}
$$

Finally, we introduce a function which given a component returns a component just like the given one except that the attributes function is (possibly) overwritten with a new value for a given attribute.

$$
\begin{array}{l}
modifyAttribute: COMPONENT \times ATTRIBUTE \times VALUE \\
\qquad\qquad\qquad\qquad \rightarrow COMPONENT \\
\hline
modifyAttribute = (\lambda c: COMPONENT; a: ATTRIBUTE; \\
\qquad\qquad\qquad v : VALUE \bullet \\
\quad (\mu c' : COMPONENT \mid \exists\, i,i' : COMP_INFO \mid \\
\qquad\qquad i = info(c) \bullet \\
\qquad i'.attributes = i.attributes \oplus \{a \mapsto v\} \wedge \\
\qquad i'.anchors = i.anchors \wedge \\
\qquad i'.presentSpec = i.presentSpec \wedge \\
\qquad c' = component(base(c), i')))
\end{array}
$$

Components can have sub-components and the same component may be a sub-component to more than one component. This relationship will be denoted by _subcomp_ and is defined below.

$$
\begin{array}{l}
\hline
subcomp: COMPONENT \leftrightarrow COMPONENT \\
\forall c_1, c_2 : COMPONENT \bullet \\
\quad c_1 \text{ Subcomp } c_2 \Leftrightarrow \\
\qquad base(c_1) \in \mathrm{ran}(composite\sim(base(c_2)))
\end{array}
$$

A hypertext system, modeled by the schema *PROTO_HYPERTEXT*, has three parts. (1) The set of *components* represents the traditional "nodes" and "links" of a hypertext system. (2) A partial function termed the *resolver* returns the UID for a given component specifier. Note that more than one specifier may return the same UID. (3) To actually get hold of a component, we introduce an *accessor* function which given a UID returns a component. Note that this function while partial, is invertible.

$$
\begin{array}{l}
PROTO_HYPERTEXT \\
\hline
components: \mathbb{F}\; COMPONENT \\
resolver: COMPONENT_SPEC \twoheadrightarrow UID \\
accessor: UID \rightarrowtail\!\!\!\!\!+\; COMPONENT \\
\hline
\end{array}
$$

To identify those links resolving to a given component, we introduce the function

linksTo which, given a hypertext system and the UID of a component in the system, returns the UIDs of links resolving to that component.

$$
\begin{array}{l}
linksTo: PROTO_HYPERTEXT \times UID \rightarrow \mathbb{F}\ UID \\
\hline
linksTo = (\lambda H : PROTO_HYPERTEXT;\ u: UID \bullet \{uid: UID \mid \\
\quad (\exists comp: LINKCOMP \mid comp \in H.components \bullet \\
\quad\quad uid = H.accessor\sim(comp) \land \\
\quad\quad (\exists s: COMPONENT_SPEC \mid \\
\quad\quad\quad s \in componentSpecs(comp) \bullet \\
\quad\quad\quad u = H.resolver(s)))\})
\end{array}
$$

There are four constraints which must be satisfied by an instance of the schema *PROTO_HYPERTEXT* before we can call it a *HYPERTEXT*.

- The *accessor* function must yield a value for every component. Because this function is invertible, every component must then 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 component specifiers of the links resolving to the component.

$$
\begin{array}{l}
HYPERTEXT \\
\hline
PROTO_HYPERTEXT \\
\hline
\forall c: components \bullet c \in \mathrm{ran}\ accessor \\
\mathrm{ran}\ resolver = \mathrm{dom}\ accessor \\
\forall c: components \bullet (c,c) \notin (_\mathrm{subcomp}_^{*} \\
\forall c: components \bullet \exists\ lids : \mathbb{F}\ UID \mid \\
\quad lids = linksTo(\theta PROTOTYPE_HYPERTEXT, accessor\sim(c)) \bullet \\
\quad first(\!| anchors(c) |\!) = \\
\quad\quad \bigcup((anchorSpecs \circ accessor)(\!| lids |\!))
\end{array}
$$

2.3 Adding New Components

In this section the model is adding a new component to a hypertext. The last function defined in this section, *CreateNewComponent*, is the function actually called from the run-time layer and is also part of the external view of the model. (See the section on conformance with the reference model for more about this external view.)

Adding a new component to the hypertext is given by the following function. It 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.

$$
\begin{array}{l}
createComponent: HYPERTEXT \times COMPONENT \\
\qquad\qquad\qquad\qquad \rightarrow HYPERTEXT \\
\hline
\forall H: HYPERTEXT; c: COMPONENT \bullet \\
\quad \exists H': HYPERTEXT \mid \\
\quad\quad H'.components = H.components \cup \{c\} \land \\
\quad\quad (\exists_1\ uid: UID \bullet \\
\quad\quad\quad (\exists\ compositeSpec: COMPONENT_SPEC \bullet \\
\quad\quad\quad\quad H'.accessor = H.accessor \cup \{uid \mapsto c\} \land \\
\quad\quad\quad\quad H'.resolver = H.resolver \cup \\
\quad\quad\quad\quad\quad \{componentSpec \mapsto uid\})) \bullet \\
\quad\quad createComponent(H,c) = H'
\end{array}
$$

The functions for creating a new node, link, and composite respectively are given below. They use the function *createComponent* described above.

$$
\begin{array}{l}
createAtomicComponent: HYPERTEXT \times ATOM \\
\quad \times PRESENT_SPEC \rightarrow HYPERTEXT \times COMPONENT \\
\hline
\forall H:HYPERTEXT; a: ATOM; ps: PRESENT_SPEC \bullet \\
\quad \exists c: COMPONENT \mid c = component(atom(a), minInfo(ps)) \bullet \\
\quad\quad createAtomicComponent(H, a, ps) = \\
\quad\quad\quad (createComponent(H, c), c)
\end{array}
$$

In creating a link, we must ensure that all of its component specifiers resolve to existing components. To test for such consistency among links we introduce the following link consistency predicate as a prefix relation.

$$
\begin{array}{l}
\mathit{linkConsistent_}: \mathbb{P}\ \mathit{HYPERTEXT} \\
\hline
\forall H: \mathit{HYPERTEXT} \bullet \\
\quad \mathrm{linkConsistent}\ H \Leftrightarrow \\
\qquad (\forall l: \mathit{LINK};\ s: \mathit{SPECIFIER} \mid \\
\qquad\quad (\exists cl: \mathit{LinkComp} \mid cl \in H.\mathit{components} \bullet \\
\qquad\qquad l = \mathit{link}{\sim}(\mathit{base}(cl))) \wedge \\
\qquad\quad s \in \hat{I}\ \mathrm{ran}\ l.\mathit{specifiers} \bullet \\
\qquad\quad (\exists\ c : \mathit{COMPONENT} \mid c \in H.\mathit{components} \bullet \\
\qquad\qquad (H.\mathit{accessor} \circ H.\mathit{resolver})(s.\mathit{componentSpec}) = c))
\end{array}
$$

$$
\begin{array}{l}
\mathit{createLinkComponent}: \mathit{HYPERTEXT} \times \mathit{LINK} \times \mathit{PRESENT_SPEC} \\
\qquad \rightarrow \mathit{HYPERTEXT} \times \mathit{COMPONENT} \\
\hline
\forall H: \mathit{HYPERTEXT};\ l: \mathit{LINK};\ ps: \mathit{PRESENT_SPEC} \bullet \\
\quad \exists H': \mathit{HYPERRTEXT};\ c: \mathit{COMPONENT} \\
\qquad c = \mathit{component}(\mathit{link}(l), \mathit{minInfo}(ps)) \wedge \\
\qquad H' = \mathit{createComponent}(H,c) \wedge \\
\qquad \mathit{createLinkComponent}(H, l, ps) = (H', c) \bullet \\
\qquad \mathrm{linkConsistent}\ H'
\end{array}
$$

In creating a composite we must ensure that any subcomponents of the new composite are already in the hypertext.

$$
\begin{array}{l}
\mathit{createCompositeComponent}: \\
\qquad \mathit{HYPERTEXT} \times \mathrm{seq}\ \mathit{BASE_COMPONENT} \\
\qquad\quad \times \mathit{PRESENT_SPEC} \rightarrow \mathit{HYPERTEXT} \times \mathit{COMPONENT} \\
\hline
\forall H: \mathit{HYPERTEXT};\ s: \mathrm{seq}\ \mathit{BASE_COMPONENT}; \\
\qquad\qquad ps: \mathit{PRESENT_SPEC} \bullet \\
\quad \exists \mathit{newComp}: \mathit{COMPONENT} \mid \\
\qquad\quad \mathit{newComp} = \mathit{component}(\mathit{composite}(s), \mathit{minInfo}(ps)) \bullet \\
\qquad \mathit{createCompositeComponent}(H, s, ps) = \\
\qquad\quad (\mathit{createComponent}(H,\mathit{newComp}), \mathit{newComp}) \wedge \\
\qquad (\forall c: \mathit{COMPONENT} \mid \mathit{base}(c) \in \mathrm{ran}\ s \bullet \\
\qquad\quad c \in H.\mathit{components})
\end{array}
$$

We package creating a new component with the following function. This is the function which will ultimately be invoked from the run-time layer.

$$
\begin{array}{|l}
CreateNewComponent: HYPERTEXT \times BASE_COMPONENT \\
\quad \times PRESENT_SPEC \rightarrow HYPERTEXT \times COMPONENT \\
\hline
\forall H: HYPERTEXT; bc: BASE_COMPONENT; \\
\qquad ps: PRESENT_SPEC \bullet \\
\quad ((\exists a: ATOM \bullet bc = atom(a)) \Rightarrow \\
\qquad CreateNewComponent(H, bc, ps) = \\
\qquad\quad createAtomicComponent(H, atom\sim(bc), ps)) \wedge \\
\quad ((\exists l: LINK \bullet bc = link\ (l)) \Rightarrow \\
\qquad CreateNewComponent(H, bc, ps) = \\
\qquad\quad createLinkComponent(H, link\sim(bc), ps)) \wedge \\
\quad ((\exists s: \mathrm{seq}\ BASE_COMPONENT \bullet bc = composite(s)) \Rightarrow \\
\qquad CreateNewComponent(H, bc, ps) = \\
\qquad\quad createCompositeComponent\ (H, composite\sim(bc), ps))
\end{array}
$$

2.4 Deleting A Component

In deleting a component we must ensure that we remove any links whose specifiers resolves to that component.

$$
\begin{array}{|l}
DeleteComponent: HYPERTEXT \times UID \rightarrow HYPERTEXT \\
\hline
DeleteComponent = (\lambda\ H: HYPERTEXT; uid: UID \bullet \\
\quad (\mu H': HYPERTEXT \mid \exists\ uids: \mathbb{F}\ UID \mid \\
\qquad uids = \{uid\} \cup linksTo(H, uid) \bullet \\
\qquad\quad H'.components = H.components \setminus H.accessor(uids) \wedge \\
\qquad\quad H'.accessor = uids \ntriangleleft H.accessor \wedge \\
\qquad\quad H'.resolver = H.resolver \mathbin{\rhd\!\!\!\!-} uids))
\end{array}
$$

2.5 Modifying Components

In modifying a component we require that its associated information remain unchanged, that its type (atom, link, or composite) remain unchanged, and that the resulting hypertext remains link consistent.

$$
\begin{array}{|l}
\textit{ModifyComponent: HYPERTEXT x UID x COMPONENT} \\
\qquad\qquad\qquad\qquad \rightarrow \textit{HYPERTEXT} \\
\hline
\forall H\text{: } HYPERTEXT;\ uid\text{: } UID;\ c'\text{: } COMPONENT \bullet \\
\quad \exists c\text{: } COMPONENT;\ H'\text{: } HYPERTEXT \mid \\
\qquad c = H.accessor(uid) \land \\
\qquad H'.components = H.components \setminus \{c\} \cup \{c'\} \land \\
\qquad H'.accessor = H.accessor \oplus \{uid \mapsto c'\} \land \\
\qquad H'.resolver = H.resolver \land \\
\qquad info(c') = info(c) \land \\
\qquad c \text{ typeConsistent } c' \land \\
\qquad \text{linkConsistent } H' \bullet \\
\quad ModifyComponent(H, uid, x) = H'
\end{array}
$$

2.6 Retrieving A Component

To retrieve a component, given its UID, means just to have the returned value of the *accessor* function.

$$
\begin{array}{|l}
getComponent\text{: } HYPERTEXT \text{ x } UID \rightarrow COMPONENT \\
\hline
\forall H\text{: } HYPERTEXT;\ uid\text{: } UID \bullet \\
\quad getComponent\ (H, uid) = H.accessor(uid)
\end{array}
$$

Given a UID which happens to represent a link, there exist operations which return either a source or destination specifier for that component.

2.7 Attributes

We introduce functions to both get and set the value of a given attribute (if it exists) for a given component.

$$
\begin{array}{|l}
\underline{\textit{Attribute Value}}: \textit{HYPERTEXT} \times \textit{UID} \times \textit{ATTRIBUTE} \rightarrow \textit{VALUE} \\
\forall H: \textit{HYPERTEXT}; uid: \textit{UID}; a: \textit{ATTRIBUTE} \bullet \\
\quad (\exists c: \textit{COMPONENT} \mid c = H.accessor(uid) \bullet \\
\quad\quad \textit{Attribute Value}\ (H, uid, a) = attributes(c)(a))
\end{array}
$$

$$
\begin{array}{|l}
\textit{SetAttributeValue}: \textit{HYPERTEXT} \times \textit{UID} \times \textit{ATTRIBUTE} \times \textit{VALUE} \\
\qquad\qquad\qquad\qquad \rightarrow \textit{HYPERTEXT} \\
\hline
\textit{SetAttributeValue} = \\
\quad (\lambda H: \textit{HYPERTEXT}; uid: \textit{UID}; a: \textit{ATTRIBUTE}; \\
\quad\quad v: \textit{VALUE} \bullet \\
\quad (\mu H': \textit{HYPERTEXT} \mid \exists\ c, c': \textit{COMPONENT} \bullet \\
\quad\quad c = H.accessor(uid) \wedge \\
\quad\quad c' = modifyAttribute(c, a, v) \wedge \\
\quad\quad H'.components = H.components \setminus \{c\} \cup \{c'\} \wedge \\
\quad\quad H'.accessor = H.accessor \oplus \{uid \rightarrow c'\} \wedge \\
\quad\quad H'.resolver = H.resolver))
\end{array}
$$

There is also a function which returns the set of all component attributes.

$$
\begin{array}{|l}
\textit{AllAttributes}: \textit{HYPERTEXT} \rightarrow \mathrm{F}\ \textit{ATTRIBUTE} \\
\hline
\forall H: \textit{HYPERTEXT} \bullet \\
\quad AllAttributes(H) = \{a: \textit{ATTRIBUTE} \mid \exists\ c: \textit{COMPONENT} \bullet \\
\quad\quad a \in \mathrm{dom}(attributes(c))\}
\end{array}
$$

2.8 Anchors

It is sometimes useful to know the link components which are associated with a particular anchor. The function *LinksToAnchor* returns the set of link component uids associated with a particular anchor id for a particular component id.

$$
\begin{array}{|l}
\textit{LinksToAnchor}: \textit{HYPERTEXT} \times \textit{UID} \times \textit{ANCHOR_ID} \rightarrow \mathrm{F}\ \textit{UID} \\
\hline
\textit{LinksToAnchor} = \\
\quad (\lambda H: \textit{HYPERTEXT}; u: \textit{UID}; aid: \textit{ANCHOR_ID} \bullet \\
\quad\quad \{lid: \textit{UID} \mid \exists\ lids : \mathrm{F}\ \textit{UID} \mid \\
\quad\quad\quad lids = linksTo\ (H, u) \wedge lid \in lids \bullet \\
\quad\quad\quad aid \in (anchorSpecs \circ H.accessor)(lid)\})
\end{array}
$$

3. Simple Runtime Layer Model

3.1 An Overview of the Runtime Model

The fundamental concept in the runtime 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 runtime 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 below) 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 accomodate 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 runtime 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 sessio on that hypertext. The user can then create instantiations of components in the hypertext (an action known as ''presenting'' the component). The user can edit these instantiations, can modify the component based on the accumulated edits to the instantiation (an action known as ''realizing'' the edits), and finally can destroy the instantiation (an action known as ''unpresenting'' a component). 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 session's current instantiations to their corresponding components in the hypertext, a history, a runtime resolver function, an instantiator function, and a realizer function. At any given moment, the history is a sequence of all operations

carried 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 conditionalized on preceeding operations.

The session's runtime resolver function is the runtime version of the storage layer's resolver function. Like the resolver, it maps specifiers into component UIDs. The runtime 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 runtime information. For example, a specifier may refer to ‘‘the most recently accessed component named ‘xyzzy’’’. The runtime resolver is responsible for mapping this specifier into the UID matching this specification. The storage layer resolver would not be able handle this specification. The runtime 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 runtime resolver.

At the heart of the runtime 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) among 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 a 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. 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 endpoints (i.e., endpoints whose specifier has direction of TO) of all link(s) that have as an endpoint the anchor represented by the given link marker. In the case where 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 is used as an argument to the storage layer modify composite operation to replace the component with the edited component. This operation is wrapped in the function called *realize edits* in the runtime layer.

3.2 Formalization of the Runtime Layer

The runtime model depends on the notion of an *instantation* which is the visual representation of some component. Each instantiation has a unique instantiation id from the given set [*IID*].

[*IID*]

An instantiation consists of a *base instantiation* which ‘‘represents’’ a component, a sequence of *link markers* which ‘‘represents’’ the anchors of the component, and a function mapping link markers to anchor ids.

[BASE_INSTANTIATION, LINK_MARKER]

INSTANTIATION
base: BASE_INSTANTIATION
links: seq *LINK_MARKER*
linkAnchor: $LINK_MARKER \rightarrow ANCHOR_ID$
dom *linkAnchor* = ran *links*

A user manipulates instantiations, so that there must be a way of mapping from instantiations to components. The function variable *instants* in the *SESSION* schema defined below maps an instantiation id to a pair consisting of an instantiation and the UID of its corresponding component. The *accessor* function in the *HYPERTEXT* schema then maps these UIDs to components. More than one instantiation may be associated with the same UID and hence with the same component.

A hypertext is manipulated in a session which is modelled by the *SESSION* schema. The *OPERATION* free type names the various operations a user can perform during a hypertext session.

OPERATION ::= *OPEN* | *CLOSE*
| *PRESENT* | *UNPRESENT*
| *CREATE* | *EDIT* | *SAVE* | *DELETE*

During a session, a user opens up one or more instantiations of hypertext components through which the hypertext may be modified. We use the term *presents* to denote opening up an instantiation on a component because the component is presented to the user by means of the instantiation. Instantiations are not only a function of the component which they represent, and two presentation specifiers --- one implicitly from the component's *compInfo* and the other explicitly, either user given or from a link specifier --- but also implicitly of the ‘‘current’’ set of instantiations. The function *instantiator* which is part of the schema *SESSION* captures this relationship. In saving the result of a series of edits, the reverse of the *instantiator* function is needed; we call this function a *realizer* function. It takes an instantiation and returns a component based on the current session.

There are some component specifiers which can only be resolved at run-time. An example of such a specifier is ‘‘the last node visited.’’ The storage layer should be independent of such component specifiers. We introduce the notion of a run-time resolver which is just an extension of the regular resolver function. Note that the invariants on anchors given in the schema for *HYPERTEXT* only apply to those component specifiers which are in the domain of *H.resolver*. Also the *LinksToAnchor* function will not give those links with component specifiers resolvable only at run-time (not in the domain of *H.resolver*) --- these additional links must be captured in the run-time layer.

```
┌ SESSION ─────────────────────────────────────
│ H: HYPERTEXT
│ history: seq OPERATION
│ instants: IID ⤔ (INSTANTIATION x UID)
│ instantiator: UID x PRESENT_SPEC → INSTANTIATION
│ realizer: INSTANTIATION → COMPONENT
│ runTimeResolver : COMPONENT_SPEC ⇸ UID
├──────────────
│ head(history) = OPEN
│ ∀uid: UID; ps: PRESENT_SPEC |
│    uid ∈ dom H.accessor •
│       realizer(instantiator(uid,ps)) = H.accessor(uid) ∧
│ H.resolver ⊆ runTimeResolver
└──────────────────────────────────────────────
```

```
┌ ΔSESSION ─────────────────────────
│ SESSION
│ SESSION'
├──────────────
│ #history' = #history + 1
│ instantiator' = instantiator
│ realizer' = realizer
└───────────────────────────────────
```

A session begins with an existing hypertext (storage system) and a clean instantiation slate.

```
┌ openSession ──────────────────────
│ SESSION
│ hypertext?: HYPERTEXT
├──────────────
│ H = hypertext?
│ history = ⟨OPEN⟩
│ instants = ∅
└───────────────────────────────────
```

Because there are several operations which can open up a new instantiation, we introduce the following function which opens up a set of new instantiations on an existing set of components.

$$
\begin{array}{l}
openComponents: \\
\quad SESSION \times \mathbb{F}\,(SPECIFIER \times PRESENT_SPEC) \\
\qquad\qquad\qquad \rightarrow SESSION \\
\hline
\forall S: SESSION;\ specs : \mathbb{F}\,(SPECIFIER \times PRESENT_SPEC) \bullet \\
\quad \exists S': SESSION;\ iids: \mathbb{F}\ IID; \\
\qquad\quad newInstants: IID \overset{\shortmid}{\rightarrow} (INSTANTIATION \times UID) \mid \\
\qquad S'.H = S.H \wedge \\
\qquad S'.runTimeResolver = S.runTimeResolver \wedge \\
\qquad S'.history = S.history \mathbin{\char`^} \langle PRESENT \rangle \wedge \\
\qquad S'.instants = S.instants \oplus newInstants \wedge \\
\qquad \#iids = \#specs \wedge iids \cap \mathrm{dom}\ S.instants = \varnothing \wedge \\
\qquad \mathrm{dom}\ newInstants = iids \wedge \\
\qquad (\forall s: specs \bullet \\
\qquad\quad \exists iid: iids;\ uid: UID; \\
\qquad\qquad cs: COMPONENT_SPEC; \\
\qquad\qquad ps: PRESENT_SPEC; \\
\qquad\qquad inst: INSTANTIATION \mid \\
\qquad\quad cs = (first(s)).componentSpec \wedge \\
\qquad\quad ps = second(s) \wedge \\
\qquad\quad uid = S.runTimeResolver(cs) \wedge \\
\qquad\quad inst = S.instantiator\ (uid, ps) \bullet \\
\qquad\quad newInstants(iid) = (inst, uid)) \bullet \\
\qquad openComponents(S, specs) = S'
\end{array}
$$

$$
\begin{array}{l}
presentComponent \\
\Delta SESSION \\
spec?: SPECIFIER \\
presentSpec?: PRESENT_SPEC \\
\hline
\theta SESSION' = \\
\qquad openComponents(\theta SESSION, \{(spec?, presentSpec?)\})
\end{array}
$$

We can also follow a link from a given link marker in a given instantiation and present all the components for which the associated link(s) has(have) specifiers with a "TO" direction. There may be more than one link involved because there may be more than one link associated with a particular anchor.

$$
\begin{array}{|l}
\hline
\mathit{followLink} \\
\Delta \mathit{SESSION} \\
\mathit{iid?} : \mathit{IID} \\
\mathit{linkMarker?} : \mathit{LINK_MARKER} \\
\hline
\exists \mathit{aid}: \mathit{ANCHOR_ID};\ \mathit{links}: \mathbb{F}\ \mathit{LinkComp}; \\
\qquad \mathit{specs}: \mathbb{F}(\mathit{SPECIFIER} \times \mathit{PRESENT_SPEC}) \mid \\
\quad \mathit{aid} = (\mathit{first}(\mathit{instants}(\mathit{iid?}))).\mathit{linkAnchor}(\mathit{linkMarker?}) \land \\
\quad \mathit{links} = \mathit{H.accessor} (\!| \mathit{LinksToAnchor}(H, \\
\qquad\qquad\qquad \mathit{second}(\mathit{instants}(\mathit{iid?})), \mathit{aid}) |\!) \land \\
\quad \mathit{first} (\!| \mathit{specs} |\!) = \{ s : \mathit{SPECIFIER} \mid \exists\ \mathit{linkc}: \mathit{LinkComp} \mid \\
\qquad \mathit{linkc} \in \mathit{links} \bullet s \in \mathrm{ran}\ (\mathit{link}{\sim}(\mathit{base}(\mathit{linkc}))).\mathit{specifiers} \} \land \\
\quad (\forall s: \mathit{specs} \bullet (\mathit{first}(s)).\mathit{direction} = \mathit{TO} \land \\
\qquad\qquad \mathit{second}(s) = (\mathit{first}(s)).\mathit{presentSpec}) \bullet \\
\qquad \theta \mathit{SESSION}' = \\
\qquad\qquad \mathit{openComponents}(\theta \mathit{SESSION}, \mathit{specs}) \\
\hline
\end{array}
$$

Opening up a new instantiation on a newly created component is modeled by the *newComponent* schema.

$$
\begin{array}{|l}
\hline
\mathit{newComponent} \\
\Delta \mathit{SESSION} \\
\mathit{component}: \mathit{COMPONENT} \\
\mathit{baseComp?}: \mathit{BASE_COMPONENT} \\
\mathit{ps?}: \mathit{PRESENT_SPEC} \\
\mathit{presentSpec?}: \mathit{PRESENT_SPEC} \\
\hline
\mathit{history}' = \mathit{history} \frown \langle \mathit{CREATE} \rangle \\
(H', \mathit{component}) = \mathit{CreateNewComponent}(H, \mathit{baseComp?}, \mathit{ps?}) \\
\exists \mathit{uid}: \mathit{UID};\ \mathit{inst}: \mathit{INSTANTIATION};\ \mathit{iid}: \mathit{IID} \mid \\
\quad \mathit{iid} \notin \mathrm{dom}\ \mathit{instants} \bullet \\
\quad \mathit{inst} = \mathit{instantiator}\ (\mathit{uid}.\mathit{presentSpec?}) \land \\
\quad \mathit{uid} = H'.\mathit{accessor}{\sim}(\mathit{component}) \land \\
\quad \mathit{instants}' = \mathit{instants} \oplus \{ \mathit{iid} \mapsto (\mathit{inst}, \mathit{uid}) \} \\
\hline
\end{array}
$$

The schema *unPresent* models the removal of an instantiation.

```
┌ unPresent ─────────────────────────────
│ ΔSESSION
│ iid?: IID
│ H' = H
│ history' = history ⁀ ⟨UNPRESENT⟩
│ instants' = {iid?} ⩤ instants
└────────────────────────────────────────
```

Instantiations can be modified by editing them. Editing an instantiation does not cause a change in its correcponding component. An explicit save operation is required to save the result of an edit (or many edits).

```
┌ editInstantiation ─────────────────────
│ ΔSESSION
│ instantiation?: INSTANTIATION
│ iid?: IID
├──────────
│ H' = H
│ history' = history ⁀ ⟨EDIT⟩
│ iid? ∈ dom instants
│ instants' = instants ⊕
│     {iid? ↦ (instantiation?, second(instants(iid?)))}
└────────────────────────────────────────
```

```
┌ realizeEdits ──────────────────────────
│ ΔSESSION
│ iid?: IID
├──────────
│ history' = history⁀ ⟨SAVE⟩
│ instants' = instants
│ ∃c: COMPONENT; inst: INSTANTIATION; uid: UID |
│         inst = first(instants(iid?)) ∧
│         uid = second(instants(iid?)) ∧
│         c = realizer(inst) •
│     H' = ModifyComponent(H,uid,c)
└────────────────────────────────────────
```

To be complete we must allow a component to be deleted. Since a component is identified by its instantiation, the component to be deleted must have been instantiated. We also must remove any other instantiations for that component.

deleteComponent

$\Delta SESSION$

iid:? IID

$history' = history \mathbin{\char`\^} \langle DELETE \rangle$

$iid? \in \mathrm{dom}\ instants$

$\exists\ uid: UID \mid uid = second(instants(iid?)) \bullet$

$\quad H' = DeleteComponent(H, uid) \land$

$\quad instants' = \{iid?\} ⩤ instants$

A session finally ends when it is closed out. Notice that the default is not to save the results of any changes to instantiations.

closeSession

$\Delta SESSION$

$H' = H$

$history' = history \mathbin{\char`\^} \langle CLOSE \rangle$

$instants' = \varnothing$

We can model a read-only SESSION with the following schema:

READ_ONLY_SESSION

SESSION

$\{SAVE, CREATE, DELETE\} \cap \mathrm{ran}\ history = \varnothing$

4. Conformance with the Reference Model

One reason to have a reference model for hypertext is to try to answer the 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 Chapter 8 of Cliff Jones' book [Jone86] on software development using VDM. First, we must exhibit total functions, called *retrieve functions* which map the actual types and functions from given (actual) hypertext system to each of the following 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 multi-way 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 have not have links with both an explicit source and destination. Thus requiring discrimination amongst all the values of type *DIRECTION* is too much.

We are currently working on a "minimal" model which addresses the above items and others as may be necessary.

The following list summarizes the given sets (base types), abstract types, functions, and operations which must have actual realizations in a hypertext system conforming to the model.

1. GivenSets

UID
COMPONENT_SPEC
PRESENT_SPEC
ANCHOR_ID
ANCHOR_VALUE
ATOM
ATTRIBUTE
VALUE
IID
BASE_INSTANTIATION
LINK_MARKER

2. Abstract types.

DIRECTION
ANCHOR
SPECIFIER
LINK
COMP_INFO
BASE_COMPONENT
COMPONENT
HYPERTEXT
INSTANTIATION
OPERATION
SESSION

3. Storage layer functions.

CreateNewComponent
DeleteComponent
ModifyComponent
AttributeValue
SetAttributeValue
AllAttributes
LinksToAnchor

4. Runtime layer operations (schemas).

openSession
presentComponent
followLink
newComponent
unPresent
editInstantiation
realizeEdits
deleteComponent
closeSession

5. Concluding Remarks

Development of the Dexter model is still in its very early stages. As discussed in Section 4, the model as currently stated is far more powerful than any existing hypertext system. The provisions for n-ary links and for composite nodes, for example, are intended to accomodate the design of future hypertext systems. No existing system that we have examined includes both n-ary links and composite nodes. The result is that no existing system 'conforms to' the model in the sense that it supports all of the mechanisms that 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

conditionalized on the exact set of mechanisms supported. Systems would be compared on the basis of the set of mechanisms that they do support.

A related issue involves a number of consistency restrictions that the present model imposes. 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. The model does not, however, include any restrictions that prevent the creation of dangling links via the deletion of linked-to components. This restriction adequately represents the consistency guarantee of KMS. But its is overly restrictive for Augment, which allows creation of initially dangling links. In contrast, its is not restrictive enough for NoteCards and HAM which prevent dangling links at all times. As in the case of mechanisms, restrictions of this sort will have to be made optional in the model. Conformance to the model can then be conditionalized on appropriate choices of restrictions. As in the case for mechanisms, systems can compared on the basis of the set of restrictions that they enforce.

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, etc. This enterprise would provide valuable feedback regarding the adequacy and completeness of the model. In particular, it will help asess 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.

The model has already been useful in developing hypertext interchange standards

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 [Oren89], a number of efforts have been started to operationalize the abstractions of the Dexter model in the form of interchange formats. Figure 5 shows an example of one such format. This format was used for experimenting with the interchange of hypertexts between NoteCards and

Hypercard. As can be seen from the figure, the format is a fairly straightforward rendering of the entities found in the Dexter model into a SGMLish 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.

.

```
<hypertext>
    <component>
        <type> text </ type>
        <uid> 21 </ uid>
        <data> This is some text . . . . </ data>
        <anchor>
            <id> 1 </ id>
            <location> 13 </ 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 trivial interchange format derived from the model.

Acknowledgement

The model described in this paper grew out a series of workshops on hypertext. The following people attended these 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, Randy Trigg, Jan Walker, and Bill Wieland. The workshops were organized by Jan Walker and John Leggett.

End Notes

[1]The terms hypertext and hypermedia are often differentiated, with hypertext referring to text-only systems and hypermedia refering to systems that support multiple media. This distinction is not made in the present paper; the term hypertext is used generically to refer to both text-only and multimedia systems.

[2]Participants in the two workshops are listed in the acknowledgements on the first page of this paper. Among the systems that were 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.

References

Akscyn, R. - Halasz, F.

- [Aksc88a] Akscyn, R. M., McCracken, D. L., Yoder, E. A.: KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. Communications of the ACM, 31:7, 820-835. 1988.
- [Camp88] Campbell, B. (et al.): HAM: A General Purpose Hypertext Abstract Machine. CACM, 856-861, 17-41. 1988.
- [Conk87a] Conklin, J.: Hypertext: An Introduction and Survey. IEEE Computer, 20, 9, 17-41. September 1987.
- [Deli86a] Delisle, N., Schwartz, M.: Neptune: A Hypertext System for CAD Applications. Proceedings of ACM SIGMOD International Conference

on the Management of Data. Washington, D.C. 132-142. January 1986.

- [Enge84b] Engelbart, D. C.: Authorship Provisions in AUGMENT: Proceedings of the 1984 COMPCON Conference, COMPCON '84 Digest, San Francisco, California. 465-472. February 1984.
- [Enge84a] Engelbart, D. C.: Collaboration Support Provisions in AUGMENT, Proceedings of the AFIPS Office Automation Conference, OAC '84 Digest, Los Angeles, California, 51-58. February 1984.
- [Fein82] Feiner, Steven, Nagy, Sandor, van Dam, Andries: An Experimental System for Creating and Presenting Interactive Graphical Documents , ACM Transactions on Graphics 1(1), 59-77. 1982.
- [Good87] Goodman, D.: The Complete HyperCard Handbook. Bantam Books, New York. New York. 1987.
- [Hala87a] Halasz, F. G., Moran, T., Trigg, R.H.: NoteCards in a Nutshell. Proc. ACM CHI+GI'87 Human Factors in Computing Systems and Graphics Interface Conf., Toronto, Canada, 45-52. April 5-9, 1987.
- [Hala88] Halasz, F.G.: Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems, Communications of the ACM, 31(7), 836-852. July 1988.

Hypertext'87 - Walker, J.

- [HT87] Proceedings of Hypertext '87, TR88-013, University of North Carolina, Department of Computer Science. ACM. March 1988.
- [HT89] Hypertext '89 Proceedings. Pittsburgh, PA USA. ACM Press (Order Number 608891). November 1989.
- [Jone86] Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Hertfordshire, England. 1986.
- [Lang89] Lange, D.B.: A Formal Approach to Hypertext Using Post-Prototype Formal Specification. Department of Computing Science, Technical University of Denmark. October 31, 1989.
- [Meyr86] Meyrowitz, N., et. al.: Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications

Framework, Proceedings of OOPSLA 86, Portland, Oregon, 186-201. September 29 - October 2, 1986.

- [Oren89] Oren, Tim-chair: Panel: Interchanging Hypertexts. In Proceedings of Hypertext'89. 379-381. 1989
- [Pear89] Pearl, A.: Sun's Link Service: A Protocol for Open Linking. Hypertext'89, ACM, 137-146. November 5-7, 1989.
- [Shne89d] Shneiderman, B.: Hypertext on Hypertext. Addison-Wesley: New York. 1989.
- [Spiv89] Spivey, J.M.: The Z Notation. Prentice-Hall International, Hertfordshire, England. 1989.
- [Trig86a] Trigg, R.H., Weiser M.: TEXTNET: A Network-Based Approach to Text Handling, ACM Transactions on Office Information Systems, 4(1). 1-23. 1986.
- [Walk87a] Walker, J.: Document Examiner: Delivery Interface for Hypertext Documents, Hypertext'87 Papers, University of North Carolina, Chapel Hill, North Carolina, 307-323. November 1987.

Walker, J. - Yankelovich, N.

- [Walk88a] Walker, J. H.: Supporting Document Development with Concordia. Computer, 21:1, 48-59. 1988.
- [Yank88a] Yankelovich, N., Haan, B., Meyrowitz, N., Drucker, S.: Intermedia: The Concept and the Construction of a Seamless Information Environment, IEEE Computer, 21, 1, 81-96. January 1988.

Table of Contents

Abstract	1
Introduction	2
1. An Overview of the Model	3
2. Simple Storage Layer Model	7
2.1 An Overview of the Storage Layer	7
2.2 Formalization of the Storage Layer	11
2.3 Adding New Components	18
2.4 Deleting A Component	21
2.5 Modifying Components	21
2.6 Retrieving A Component	22
2.7 Attributes	22
2.8 Anchors	23
3. Simple Runtime Layer Model	24
3.1 An Overview of the Runtime Model	24
3.2 Formalization of the Runtime Layer	26
4. Conformance with the Reference Model	32
5. Concluding Remarks	34
The model has already been useful in developing	35
Acknowledgement	37
End Notes	37
References	37
Akscyn, R. - Halasz, F.	37
Hypertext'87 - Walker, J.	38
Walker, J. - Yankelovich, N.	39