Using a Reference Model for Information System Compatibility: an Experience

Cesare Maioli[1], Wilma Penzo[1], Stefano Sola[2], Fabio Vitali[2]

1) Computer Science Dept., University of Bologna, ITALY
2) CIRFID, University of Bologna, ITALY

Abstract

Because of the rapid pace of advances in the field of hypertext, data interchange systems are becoming important for the longevity of information systems based on hypertext or including hypertext features. Reference models, identifying those traits that are common to all systems of a certain kind, allow us to create an intermediate data format useful for data interchange. The Dexter Hypertext Reference Model appears the most apt to capture all the different aspects of hypertext systems. In this paper we present some issues that are central in developing the interchange procedure between a hypertext information system and the Dexter model, examine some of the difficulties we faced when dealing with Dexter, and sketch, as an example, the formal model of our hypertext system RHYTHM along the lines of the Dexter Model. With a little elasticity and some minor modifications, we were able to generate a satisfactory model of RHYTHM, and this shows that Dexter has a very robust structure capable of dealing with an unexpectedly large class of hypertext systems.

1 Introduction

Information systems that are beginning to use hypertext systems for central or secondary tasks are bound to depend on architectural choices that may have drastic effects in the future. Hypertext technology is still in its infancy, its core and boundaries have yet to be unanimously defined: new systems are realised, new technologies come into widespread use and new user requirements are presented that make current decisions appear premature and inadequate for the future. Whether an information system uses highly promising hypertext prototypes, commercial systems, or in-house approaches, the chances are that in a few years they will all become obsolete because of the speed of progress in this specific and in related fields. The reuse of information in different applications (not necessarily hypertexts), the possibility of showing the same hypertext on different platforms and/or presenting it with different hypertext programs, is essential for information managers [Glu92]. Reference models provide a general though indirect solution to ensuring a long useful life to an information system; they allow us to map functionals and data structures between different applications, thereby smoothing the transition to newer or more powerful ones.

Since reference models identify only those traits that are common to all relevant systems, hypertext reference models will concentrate on their key features: data organisation (i.e., *nodes*) and data connection (i.e., *links*). The Dexter Reference Model, deriving from of a series of workshops attended by many important professionals in the hypertext field, is the most widely accepted hypertext reference model. The large number of contributors (most of whom are architects of important existing hypertext systems) and its generality and completeness of vision helped the model assume an important role in the hypertext community. The usefulness of the Dexter Model was stressed by a panel at Hypertext '89 [Aks89] and was later used as a paradigm for comparing systems (for instance in [Sch90], [Bie91], [Gar91] and [Fri91]).

In [Grø92] the realisation of a hypertext system following the strict guidelines of the Dexter Model is described. Even though some unforeseen pitfalls in the original Dexter Model were exposed, an interesting hypertext system was created, clearly thanks to the generality of the model itself. Interesting results were obtained regarding the Dexter Model *per se*: the emphasis on avoiding dangling links is considered excessive and the encapsulation of components within composites is said to inhibit using external objects within the hypertext environment.

In [Kil90], Killough and Leggett used Dexter as the intermediate format for a hypertext interchange procedure, at present limited to the interchange between Intermedia and KMS. Since Dexter was the interchange format, two procedures were needed to obtain the translation, *IIF2Dexter* (Intermedia Interchange Format is the output format of the Transfer facility of Intermedia) and *Dexter2KMS*. The whole interchange testbed foreseen in [Kil90] comprises systems such as Intermedia, KMS, Notecards, Hypercard and Guide. Many (if not all) of their main architects took part in the committee that originally delivered the Dexter Model. It is therefore hardly surprising that the Dexter Model is

This work has been developed within the Progetto Finalizzato CNR "Sistemi Informatici e Calcolo Parallelo", Subproject #7 "Sistemi di Supporto al Lavoro Intellettuale" and ESPRIT Project #6763 "Document Logistics"

an adequate intermediate interchange format for those systems and those that they closely resemble.

On the other hand, there are hypertext systems that present great differences from the systems described in the Dexter Model, and information systems that are not primarily hypertext-based but present some features that have a clear hypertext flavour. We believe these systems to be the real test for the generality of the Dexter Model, and an important issue to be that of verifying whether Dexter can be used for their formalisation.

In this paper we will stress the importance of reference models for the integration of hypertext functionalities within information systems, with particular regard to compatibility and data interchange issues.

Because of their generality, reference models need to be adapted and completed. Systems that differ widely from those that inspired Dexter need particular care in doing so, since finding the correct way may be somewhat tricky.

As an example, this paper would like to present our successful attempts to adapt the model to RHYTHM, a hypertext system that is currently being developed at the University of Bologna. For various reasons RHYTHM differs widely from the aforementioned hypertext systems, both in its features and basic structures. We feared that Dexter could not be used to model RHYTHM, and that interchange to and from RHYTHM should take other forms. However, this work proved that Dexter is robust and general enough for RHYTHM, although a few corrections and secondary modifications were carried out. Indeed, our modifications neither change Dexter's basic philosophy nor detract from its generality. On the contrary, we found it very easy to implement our main modifications to the model: allowing anchors to be stored externally (and therefore remotely) to the node they refer to, just like often happens in distributed or open hypertext systems.

To allow our system to be modelled along the lines of Dexter, we first corrected and generalised Dexter to include a larger class of hypertext systems (most notably, those employing external anchors), while maintaining the generality and incompleteness of a good reference model; then we specified RHYTHM in Dexter terms, despite their apparent dissimilarities.

We hope that these efforts will convince information system managers of the usefulness, soundness and robustness of the Dexter Model, and that our modified model will be the basis for an evolving and developing view of a complete reference model. Some new hypertext features still have to be integrated (including distribution support, multi-user creation and multi-authored note taking), some others will have to be made possible by appropriate spaces (we are particularly interested in versioning, inclusion and innovative access mechanism such as browsers).

This paper will summarise the efforts needed to adapt Dexter to largely different hypertext systems or information systems employing hypertext features. In the next section we will briefly introduce the basic steps that need to be taken when applying hypertext features to information systems by means of reference models. In section 3 we will briefly introduce the Dexter Model. Section 4 is devoted to the modifications to Dexter being in our opinion general enough to be included in the Model itself. Section 5 and 6 use RHYTHM as an example of this adaptation: we will first introduce RHYTHM's basic features and then show the most prominent points of our specific model.

2 Information systems and reference models

A reference model is a structured description of a domain used to compare existing implementations, design new ones and map out possible areas for standardisation [VDP90]. Reference models come in handy when defining and describing the boundaries and basic features of a field and help to provide a more formal and structured approach in designing new systems and in relating them to existing ones. Finally, they may help to define a transition from an older system to a newer one.

Reference models (RMs) are an aid towards standardisation, but are not standards in themselves. Indeed, as [Cra90] notes, long-lived standards can only emerge after we know much more about how people use hypermedia. The field of hypertext is still young and many major advances can be expected within the next few years. A premature effort to standardise may inhibit innovation ([HKo90], [Ore90]), while hypertext RMs may lead in the right direction, helping to integrate them into a general understanding of what constitutes the hypertext field. Furthermore, RMs are not full formal models: they aim to model only common features of different systems, present and future, and do not specify any of them in full. Therefore an RM is by nature incomplete and general, explicitly leaving provisions for the detailed specification of each system.

RMs are an invaluable aid in defining standards for interoperability and data interchange. They provide a clear and simplified path for transferring data between systems: they can be used either to modernise an information system by moving all the data to a new hypertext system, or for the regular exchange of data in both directions between two different systems that are used for different purposes or in different parts of the same enterprise.

In a certain sense, therefore, they provide some solutions for managing information systems: whether a specific hypertext system is to be chosen that will clearly be obsolete in a few years, or it is decided that two different hypertext systems be used in different parts of the information system, the existence of a hypertext RM may guarantee that current efforts will not be wasted in subsequent upgrades and reunifications.

The interchange between different systems can be approached as an *ad hoc* procedure which involves the case by case determination of those functions that can be maintained, those that need to be remapped to similar ones, and those that will be lost. Furthermore, this procedure needs to be repeated for all systems involved, thus making interchange a complex process. Otherwise, interchange may involve the determination of an intermediate format that provides a framework for all specific procedures. The intermediate format helps to limit the number of interchange procedures between many different systems, which would otherwise be needed from every source system to every destination system. The number of different combinations that have to be examined is therefore n instead of $n(n-1)/2$.

The intermediate format may easily result from a reference model. The Dexter Model, for instance, was explicitly devised to produce such an intermediate format for hypertext.

After finding a proper reference model, the complete model of the system can be obtained by completing and specifying the parts that were incomplete or too general in the reference model. This can be tricky if the system widely differs from those that inspired the original model, as our experience shows. In our case, this required a certain amount of elasticity and fantasy, but the results were satisfactory.

Nodes and links are the key characteristics of hypertext systems and systems with hypertext features. A complete model along the lines of Dexter will therefore try to define nodes and links in Dexter terms, by separating the parts concerned with content, storage and display of nodes and links with the aid of the three layers at the core of the Dexter Model. Modelling will continue by filling in the blanks in the model: the Dexter Model is concentrated on more general issues than data types, file systems and other specific computer- related concepts. It is therefore necessary to specify Dexter schemata down to commonly understood computer terms (for example, JPEG or RTF data types, the operating system's filename conventions, etc.).

An important step is then identifying where the system falls short of the model: for instance, while the Dexter Model allows n-ary links, many systems (including RHYTHM) only allows two end-points for each link. All restrictions with respect to the generality of the model should be carefully noted and pointed out. This is important if the system being modelled is a source of the interchange procedure, and becomes fundamental when modelling the destination system. In this case, it is also necessary to establish a policy for dealing with cases not covered by the system. For instance, since RHYTHM allows link end-points to partially or totally intersect, the policy would provide for an n-ary link to be transformed in as many coinciding binary links as necessary.

3 The Dexter Model

In this section, we briefly introduce some concepts of the Dexter Model, which we will amend and then apply in sections 4 and 6. The Dexter Hypertext Reference Model has been developed taking many different hypertext systems into account, among which Augment, Concordia, IGD, FRESS, Intermedia, Hypercard, Hyperties, KMS, HAM, Notecards, Sun Link Service and Textnet. It is written using Z, a set-based specification language [Spi92]. A small introduction to some basic concepts of Z can be found in the appendix at the end of this paper.

Hypertext systems are divided into three layers, called *within-component layer* (WCL), *storage layer* (SL) and *run-time layer* (RTL). The focus of the model is the SL, where the basic node-link structure is represented through the concept of *component*, generic containers of data, collected in a database (the *hypertext*) and connected by relational *links*. The WCL concerns the internal representation of application data within the application components, while the RTL concerns the way these components are shown on the screen and how the user interacts with them. To obtain a relevant degree of generality, the WCL and the RTL are only modelled superficially, since they are application- dependent and are different in each system.

The SL is based on the concept of components, which can either be atoms, links or arbitrary sequences of other components, called *composites*. Thanks to composites, a certain freedom in composition is obtained and it is possible to create new first class structures in the hypertext database as collections of simpler ones. While the content is contained in the schema BASE_COMPONENT, a schema is devoted to handling properties of the components, COMP_INFO, and an identifier, UID, is uniquely associated with the components via the accessor function:

$$
\begin{aligned}
BASE\_COMPONENT ::= \; & atom\langle\!\langle ATOM \rangle\!\rangle \\
& \mid link\langle\!\langle LINK \rangle\!\rangle \\
& \mid composite\langle\!\langle \mathrm{seq}\ BASE\_COMPONENT \rangle\!\rangle
\end{aligned}
$$

COMPONENT

$$
\begin{aligned}
& compBase : BASE\_COMPONENT \\
& compInfo : COMP\_INFO
\end{aligned}
$$

$$accessor : UID \rightarrowtail\!\!\!\!\!\rightarrow COMPONENT$$

A link is a sequence of specifiers, which are structures used to refer to the whole or some specific parts of an existing component. For our purposes, the Dexter specifier is a key concept: it captures the essence of link end-points. Each specifier refers to a component, and within it to an anchor, i.e., a specific point (some words, a place, some objects) within the component to which the link is attached. The direction field is used to model the direction of links, which are directed arcs in some systems, two-way connections in others and can be simple bookmarks in others.

Presentation specifications provide a generic way of handling different policies in showing link markers (signs for the user of the existence of a link).

$$
\begin{aligned}
DIRECTION ::= \; & FROM \mid TO \\
& \mid BIDIRECT \mid NONE
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
\textit{SPECIFIER} \\
\textit{componentSpec} : \textit{COMPONENT\_SPEC} \\
\textit{anchorSpec} : \textit{ANCHOR\_ID} \\
\textit{presentSpec} : \textit{PRESENT\_SPEC} \\
\textit{direction} : \textit{DIRECTION} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{LINK} \\
\textit{specifiers} : \mathrm{seq}\ \textit{SPECIFIER} \\
\hline
\#\textit{specifiers} \geq 2 \\
\exists\, s : \mathrm{ran}\ \textit{specifiers} \bullet s.\textit{direction} = \textit{TO} \\
\hline
\end{array}
$$

$$\textit{resolver} : \textit{COMPONENT\_SPEC} \nrightarrow \textit{UID}$$

A component is usually referred to by the UID, an unique identifier that allows direct access to it. Specifiers store a component specifier, which is mapped onto the relevant UID through the resolver function.

This is very important because it allows links to store virtual references, which can change over time and act as dynamic queries. In many systems (Hypercard is the clearest example) a link is not stored as a connection between two fixed nodes, but is computed at run-time executing a script (e.g. in Hypercard: "Goto card number a+b+c", where values a, b, and c may change over time).

Anchors represent a way of modelling span-to-span links, i.e., those that connect a specific part of a node (a word, an object, a place) to a different specific part of the destination node. The anchors are used to represent the said part of the nodes. Specifiers only store the identifier, while in the component a table of anchors helps to map the identifier to the corresponding value.

$$\textit{ANCHOR} == \textit{ANCHOR\_ID} \times \textit{ANCHOR\_VALUE}$$

$$
\begin{array}{|l}
\hline
\textit{COMP\_INFO} \\
\textit{attributes} : \textit{ATTRIBUTE} \nrightarrow \textit{VALUE} \\
\textit{anchors} : \mathrm{seq}\ \textit{ANCHOR} \\
\textit{presentSpec} : \textit{PRESENT\_SPEC} \\
\hline
\#\textit{anchors} = \#(\textit{first}(\!|\, \textit{anchors} \,|\!)) \\
\hline
\end{array}
$$

It is to be noted that specifiers use a very general mechanism to refer to components, which can change dynamically over time, but permanently store the identifier of the specific anchor: different components that at different times become end-points of the same link must share the same anchor identifier. A critical piece of the Model is the interface between SL and WCL: ATOMs and ANCHOR_VALUEs are important in the Storage Layer but are not modelled because they are specific to each hypertext system, since they depend on the internal characteristics of the components.

The main Run-Time Layer concept is that of the instantiation of a component, which is shown on the screen and interacts directly with the user. Anchors are instantiated in link markers, but there is no strict relation between the anchor defined in a component and the marker shown in the instantiation: this means that we are free to use the concept of anchor for more things than just navigational links: for instance, to define bookmarks, notes, script activators, etc.

$$
\begin{array}{|l}
\hline
\textit{INSTANTIATION} \\
\textit{base} : \textit{BASE\_INSTANTIATION} \\
\textit{links} : \mathrm{seq}\ \textit{LINK\_MARKER} \\
\textit{linkAnchor} : \textit{LINK\_MARKER} \rightarrow \textit{ANCHOR\_ID} \\
\hline
\mathrm{dom}\ \textit{linkAnchor} = \mathrm{ran}\ \textit{links} \\
\hline
\end{array}
$$

4 Modifications, adaptations and additions to Dexter concepts

Many obstacles can be found in using Dexter to create the complete model of a hypertext system. Some of these problems are apparent, and can be resolved by the careful use of existing Dexter structures (e.g. in our case, Dexter's node-link structure is very different from RHYTHM's virtual reference list); others are due to specific features of the system not considered by Dexter (e.g. users and group handling) because of generality. Others involve further specifications of existing structures (e.g. inclusion links or distribution support). In the next section we will describe our own experience in facing these kinds of problems for our system, RHYTHM. Whereas, in this section we would like to describe some problems that in our opinion have some kind of generality, so that our solutions should be considered modifications to the Dexter Model *per se*, rather than adaptations of it to a specific purpose. However, this is not the place for a lengthy discussion, so we will only hint at some problems and our proposed solutions.

For instance, Dexter allows links to be part of composites, but then avoids checking link properties on these: only LinkComps (i.e., components, the BASE_COMPONENT of which is only one LINK) are checked in various functions like linksTo, linkConsistent, etc. Indeed, LinksToAnchor returns the UID of the links associated with a given anchor, but links belonging to a composite may not have a UID of their own, since they have been used as an element within its BASE_COMPONENTs, and a UID is only associated with the whole composite. We have reformulated all these functions to take into account links belonging to composites. The full schemata are somewhat longer than the originals and cannot be presented in this paper[1]. Another example regards the definition of specifiers. By rigidly storing the ANCHOR_ID of the end-point component, yet still allowing components to change over time thanks

[1] A couple more oversights regard the definition of HYPERTEXT and the definition of ModifyComponent, where the COMP_INFO part of the new component is exactly the old one, so that it is impossible to modify (attribute,value) pairs or update local anchors when the underlying content has been modified. For a complete exposition of these points, refer to [Pen93].

to a query mechanism based on COMP_SPEC, the system is forced to use a consistent numbering scheme of anchors within components: if a link points to different components at different times, the list of anchors of each must use the same identifier for the corresponding anchor, which in our opinion entails a heavy consistency burden. Therefore we suggest storing a specifier for anchors as well, so that the end-point anchor identifier is computed at run-time along with the component identifier:

SPECIFIER
$componentSpec : COMPONENT\_SPEC$
$anchorSpec : ANCHOR\_SPEC$
$presentSpec : PRESENT\_SPEC$
$direction : DIRECTION$

$$resolver : COMPONENT\_SPEC \times ANCHOR\_SPEC \rightarrowtail UID \times ANCHOR\_ID$$

or, equivalently, if we do not like to modify the existing resolver function,

$$resolver : COMPONENT\_SPEC \rightarrowtail UID$$

$$AIresolver : UID \times ANCHOR\_SPEC \rightarrowtail ANCHOR\_ID$$

Computing the end-point of a link therefore becomes a run-time query both for identifying the component and for identifying the exact anchor.

A final remark must be made regarding *external anchoring* and the associated problem of *anchor updating*. Some systems do not store the reference to the end-point of the anchor within the data container, but in an external repository so that creating a new anchor requires no modification to the data or the data container. Such systems require an explicit update mechanism for anchors since the data editor does not know of the anchor, which is useful in open hypertext systems [Dav92], distributed systems [Mai93] or when dealing with ROM and WORM devices [Ore87] and [Ore90]. In Dexter terms, an external anchoring mechanism is one that allows Anchor Values to be stored externally to the Component they refer to.

The Dexter Model clearly supports only internal anchors: specifiers store the Anchor ID, which is mapped to the appropriate Anchor Value thanks to the anchor table that is stored in the CompInfo of the component.

In internal anchoring *á la* Dexter the update of Anchor Values is the duty of the same entity entrusted with modifying the component itself (in practical terms, if the anchor is stored as a tag within the data, the anchor moves automatically when the data is modified). This is not always possible for all systems: for instance open hypertext systems leave the editing to the original application, which is unaware of links and anchors, and updates all anchors when saving the document. Others (particularly Xanadu or RHYTHM) refer to versioning to obtain the current value of the anchor.

Internal or external anchoring is a basic decision that different hypertext systems take in different ways. We did not find a common solution that models both ways and leaves the choice of specifying one to each system model. On the other hand, we have previously proposed a change in the way anchors were handled by the specifier, so that their identifier was the result of a run-time query, by having the specifier schema store an Anchor Spec in place of the Anchor ID, and by adding a function that maps Anchor Specs to Anchor IDs when traversing links. This change is sufficient for our proposed solution: the basic difference between internal and external anchoring is that with internal anchoring the identifier is translated into the appropriate value by means of a table in the component, while with external anchoring the specifier contains or maps directly to the anchor value without querying the component.

Therefore an internal anchoring mechanism will depend on the AIresolver function previously shown:

$$AIresolver : UID \times ANCHOR\_SPEC \rightarrowtail ANCHOR\_ID$$

while an external anchor mechanism is simply characterised by a different AVresolver function, which directly returns the Anchor Value:

$$AVresolver : UID \times ANCHOR\_SPEC \rightarrowtail ANCHOR\_VALUE$$

Different hypertext systems will use one of the two functions in their HYPERTEXT definition.

Anchor updating then becomes an autonomous choice for system implementers: they either have it automatically when editing the component, or deduce it when saving (e.g. during the RTL function *realizeEdits*) or else, thanks to versioning, deduce it when traversing the link (i.e., in the AVresolver function itself).

In this section we have briefly mentioned some modifications that we would make to the Dexter model *per se* in order to make it more general and adaptable to a larger number of hypertext systems. We tried to maintain the original philosophy and found that we had to modify only marginal aspects of the model. Better and more elegant solutions to the problems stated are surely possible. In particular, since Dexter specifies neither Anchor IDs nor Anchor Values, one could argue that the external anchoring problem could be solved by using Anchor Values as Anchor IDs, so that there would be no need to add the AVresolver function. However, we believe that this answer would have meant merely playing with words, and that it was more important to find a general and open solution.

In our opinion, the most important function that a hypertext RM should perform is to give a clear and unambiguous aid to understanding similarities and differences between different systems, so that data interchange and interoperability are obtained with limited effort. The force of Dexter lies in the fact that it is largely general and unspecified: it is then possible, by twisting the underlying semantics, that all its schemata mean completely different things, but that, at the same time, none of its constraints is violated. However, doing so would eliminate any hope for Dexter to be used as a satisfactory means of facilitating hypertext interchange.

5 An example: The RHYTHM hypertext system

We now introduce our own experience with checking the generality of the Dexter Reference Model. We used as an example the RHYTHM (Research on HYpertext THeory and Management) Hypertext System, which is being developed at the University of Bologna (for a complete description of RHYTHM, see [Lam92], [Mai91], [Mai92] and [Mai93]). In this section we emphasize the most prominent characteristics of RHYTHM, emphasizing where it mostly differs from those systems that inspired Dexter. In the next section, the main schemata of our model are presented.

RHYTHM is a distributed hypertext system that, besides free linking capabilities among distributed servers, offers inclusion, versioning and authoring control. RHYTHM features a document system to store, retrieve and show documents in a distributed manner. Documents are versioned, i.e., for each document the system can retrieve and show any intermediate state of its development. In the same way, they are authored: the system can retrieve and show the contributions of each author of a collaborative document. A document is therefore a complex entity developing asynchronously over time many different users.

Each version of a document is a virtual entity, i.e., a list of references (called anchors) either to other documents or to hidden data containers; all anchors to documents are recursively resolved in other references until an anchor to the actual data containers is obtained. Distribution is due to the fact that each anchor may refer to an arbitrary server on the network, and inclusion is obtained by adding a reference pointing to a specific part of another document.

A new version of a document is automatically activated when it is shown to the user: at the start it will contain just a single reference to the previous version of the document, but during the editing many new anchors are added to account for the insertion of new data, the deletion of old data and the inclusion of data from other documents. Editing is completed on saving the new version of the document: only the new data and the new list of anchors need to be stored, since the old data already exists and can be obtained by querying the previous version of the document.

Each document is associated with a group of authors, who are the only ones allowed to create a new "official" version of the document. But a new anchor list is created whenever anybody opens a document: if the user belongs to the group, the system creates a new version (we call this *versioning*), otherwise it creates a new document containing all or part of the said document (we call this *authoring*). In this way everybody is allowed to access and modify each readable document as if owning it and can freely add or delete, because all modifications are automatically stored in a personal copy of the document (without duplication of the common data). Furthermore, every version of a document is still susceptible to being modified, so that independent branches of subsequent versions can be created and parallel versions can be maintained.

As documents are divided into versions, links and inclusion end-points can either refer to a specific version or be requested to transform themselves into the corresponding end-points of the current version of the receiving document. They can therefore contain either *static anchors* or *dynamic anchors*. The system automatically checks consistency and existence and avoids circularity.

The server, which is responsible for storing data, is independent from the data format: it just handles blocks of bytes. The server builds the document from the list of anchors, creating a list of blocks, called *handlers*, which store different kinds of data. The client receives these blocks and connects an appropriate data manager and a part of the screen to each handler. Only the data managers understand the data format. The client thus appears to the user as a generic multimedia editor with which it is possible to create zones containing different types of data.

6 Basic concepts of the RHYTHM-Dexter Model

As we have said in section 2, creating a model of a system along the lines of a reference model requires the mapping of the system's capabilities in terms of the model, and the detailed specification of all features that fall in the model's spaces, to obtain a complete model.

There are three different tasks that have to be undertaken to obtain the complete specification of a real hypertext system: the given sets (i.e., unspecified entities) of the model must be specified according to the real system (e.g. ATOMs and ANCHOR VALUEs); the schemata that are too general for the real system have to be restricted (e.g. Dexter allows n-ary links, while RHYTHM allows only binary links), and specific features must be added by modelling and appropriately inserting them into the existing model. The last task has two aspects: on the one hand, completely new concepts have their own new schemata (e.g. users and group handling), on the other, limited schemata are to be expanded (e.g. specifiers' DIRECTION only refers to navigation links, while we need to use links for inclusion as well).

What follows is therefore a brief account of our experience in building the model of a hypertext information system with the aid of the Dexter Reference Model. The entire Dexter Model revolves around the definition of Base Components. RHYTHM components are *files*, *documents* and *versions*. None of them were easily mapped to Dexter components, unless a sort of creative approach was taken. In RHYTHM there is a complete distinction between the entities storing the actual data (files) and entities shown to the user (versions of documents). Since a version is a list of references, we decided to exploit links by introducing the concept of inclusion links: versions are composites made exclusively of links: some of them are navigational links, and the others, i.e., inclusion links, represent the actual data. Data is actually stored in other components, called *docuverses*, composed entirely of data containers, ATOMS, which are then directly mapped to files. Different docuverses correspond to different RHYTHM servers, thus obtaining distribution. While RHYTHM WCL and SL only deal with data in their simplest form, *bytes*, the system then transforms them into meaningful structures, *handlers*, when transferring them to the RTL, corresponding to RHYTHM clients. The Presentation Specifier thus helps to determine the handler structure that maps the blocks of data in meaningful multimedia blocks. The following briefly explains these points in greater detail.

6.1 The Within Component Layer

The Within-Component Layer refers to the parts of the model which are internal to the component. There are two important contact points between the SL and the WCL: the atoms, which model the actual containers of data, and the anchor values, which model a referencing schema to internal parts of the component, independently of data type, data format or modifications over time.

We thought that our main task was to specify the model down to very simple computer concepts: file and bytes. Therefore an ATOM was thought to be a sequence of data elements made of bytes which could easily be mapped to files. The choice of a very general data type for atoms is due to the fact that in practice the RHYTHM system does not resolve blocks of data to their specific data type until the block is delivered to the client: the server only knows one data type, which is the array of bytes. This was easily represented in the Dexter Model by considering the Presentation Specifiers as the descriptions of data types that allow the client to consider a generic block of data as an array of a specific data type.

$$ATOM == \mathrm{seq}\, BYTE$$

It is not clear whether the FILE schema should be put in the Storage Layer, where it belongs by name, or in the WCL, where all low level concepts should probably be placed. Files are nonetheless the places where data collected in atoms is permanently stored. Files can be accessed through filenames.

```
┌─ FILE ─────────────────────────
│ fileName : FILENAME
│ data : ATOM
└────────────────────────────────
```

Anchor values are a means of referencing internal subparts of components. In RHYTHM an anchor value refers to an interval within the sequence of data-elements of the component. This interval is specified by its starting point and its length:

$$ANCHOR\_VALUE == POSITION \times LENGTH$$

6.2 The Storage Layer

The Storage Layer of the RHYTHM model is inevitably more complex than Dexter's. However it is to be noted that, with the exception of the modifications expounded in section 4, all further modifications to the Storage Layer schemata are just additions to handle specific RHYTHM features: whenever the RHYTHM Model derives directly from Dexter, no further modification is necessary. In the next subsection we will briefly account for the most prominent features of the RHYTHM Hypertext System: we will first introduce the two main kinds of link: navigation and inclusion. In the following subsection Dexter components as handled in the RHYTHM schemata are shown: docuverses and versions. Next, documents are introduced as a superstructure that connects versions in useful aggregates. Finally, the basic concepts of authorship and ownership are introduced, and the difference between versioning and authoring is explained.

6.2.1 Navigation links and inclusion links

RHYTHM heavily depends on inclusion links: all new versions are created including previous ones, and it is always possible to include an arbitrary chunk of data from a version of another document. Therefore we add a new specifier type, INCLUDE, and divide links into two disjointed classes: navigation links and inclusion links. In RHYTHM links have only two specifiers and it not possible to have mixed kinds of link: they are either FROM-TO links (navigation links) or INCLUDE-TO links (inclusion links).

$$LINK\_TYPE ::= FROM \mid TO \mid BIDIRECT \mid NONE \mid INCLUDE$$

```
┌─ INCLUSION_LINK ───────────────
│ LINK
├────────────
│ ∃ s : ran specifiers •
│     s.direction = INCLUDE
└────────────────────────────────
```

```
┌─ NAVIGATIONAL_LINK ────────────
│ LINK
├────────────
│ ∃ s : ran specifiers •
│     s.direction = FROM
└────────────────────────────────
```

Though the difference between navigation and inclusion links is shown as part of a specific model, we believe that it can be thought to contain some generality of its own. Indeed, our modifications can be applied also to other information system using similar kinds of references: hot and warm links, the "publish and subscribe" mechanism in Macintosh System 7, etc.

6.2.2 Docuverse and versions

We needed to differentiate between components as well: in RHYTHM, data is never stored in components that can be accessed by the user, and components accessible by users do not contain any kind of data, only references. Through the use of inclusion links, on the other hand, it is possible to regain all the advantages obtained by the complete definition of composites. Therefore there are two kinds of components, ATOM_SEQ_COMPONENT and LINK_SEQ_COMPONENT, which are then aliased to the corresponding RHYTHM terms, *DOCUMENTs* and *VERSIONs*:

$$
\begin{array}{|l}
\textit{ATOM\_SEQ\_COMPONENT} \\
\hline
\textit{COMPONENT} \\
\hline
compBase \in \operatorname{ran} composite \land \\
\quad (\forall\, bc : BASE\_COMPONENT \mid \\
\quad bc \in \operatorname{ran}(composite^{\sim}(compBase)) \bullet \\
\qquad bc \in \operatorname{ran} atom) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{LINK\_SEQ\_COMPONENT} \\
\hline
\textit{COMPONENT} \\
\hline
compBase \in \operatorname{ran} composite \land \\
\quad (\forall\, bc : BASE\_COMPONENT \mid \\
\quad bc \in \operatorname{ran}(composite^{\sim}(compBase)) \bullet \\
\qquad bc \in \operatorname{ran} link) \\
\hline
\end{array}
$$

The docuverse (the name obviously comes from Nelson's Xanadu [Nel90]) is therefore the container of all the actual data (which will never be shown to the user) and is used by the system to create the actual content of versions. The docuverse does not contain links and cannot be referenced by a navigation link. Versions, on the other hand, are made exclusively of links, which are either inclusion links (the actual data content of the component) or navigation links. All navigation links necessarily point to another version. To show a version, it is necessary to recursively explode all inclusion links to other versions until they all become inclusions of atoms of the docuverse, which contain the data to be shown.

6.2.3 Documents

Versions are collected in larger entities, documents, that establish relations between them. From the user's point of view, the document is the foremost entity of the system. A document is usually accessed in its most recent form, the *latest version*, but it is possible to show or refer to any previously saved state, which is stored in an autonomous version. In its more general form, a document is a tree of versions, as its development may diverge into two or more different current versions. All documents therefore share a common root, and for each version there exists one, and only one, sequence of versions starting from the root and leading to it. A numbering schema has been devised to uniquely identify the position of a version within the version tree.

$$
\begin{array}{|l}
\textit{DOCUMENT} \\
\hline
name : NAME \\
group : GROUP\_ID \\
root : VERSION\_UID \\
latestVersion : VERSION\_UID \\
\hline
\end{array}
$$

6.2.4 Users and groups

Ownership relates to documents, and is the root of the distinction between authoring and versioning. As in many multi-user systems, users can be collected in GROUPs, which can freely be created, mixed and modified.

Documents are owned by GROUPs, so that it is possible to have multi-authoring of the same document. Each version is due to a single user, so that it is necessary that every user who creates a version belongs to the group owning the document. If he does not, then a new document has to be created. Information regarding the author and the position of a component within a version tree is stored in the COMP_INFO, as pairs (attribute, value). It has therefore been necessary to make the given sets ATTRIBUTE and VALUE partially explicit. For example,

$$
\begin{aligned}
ATTRIBUTE ::= {} & root \mid author \mid num\_version \\
& \mid sys\_admin \mid OTHER\_ATTRBS
\end{aligned}
$$

A docuverse component only needs the system administrator value, while a version component stores its author, its position number within the version tree of the document it belongs to and the root version of the document it belongs to.

6.3 The Run-Time Layer

The Run-Time Layer is responsible for presenting the component to the user as an instantiation. In RHYTHM the version is built by the server and sent over the network to the client, which is responsible for showing it. While for the server the version is instantiated as a block of bytes, the client is able to apply a meaningful structure to it by means of the Presentation Specifiers and divide it into many smaller blocks of known structure, called *handlers*.

$$
\begin{array}{|l}
\textit{INSTANTIATION} \\
\hline
handlers : \operatorname{seq} HANDLER \\
links : \operatorname{seq} LINK\_MARKER \\
linkAnchor : LINK\_MARKER \\
\qquad \rightarrow ANCHOR\_SPEC \\
\hline
\operatorname{dom} linkAnchor = \operatorname{ran} links \\
\hline
\end{array}
$$

A handler is a complex structure. As mentioned above, a new list of inclusions (called "anchor list" even if they are actually inclusion links) is built every time a document is opened, and the system only needs to save the new anchor list and the new data added in the current session. The mechanism is too complex to be shown here, but it is reflected in the inner structure of the handler: The data_block sequence contains the current state of the handler to be shown to the user. The new_data sequence contains just the new data added during the current session and the anchor_list contains all inclusion links making up the new version of the document. The data_block sequence will be discarded when the version is saved.

HANDLER
$anchor\_list : \mathrm{seq}\ BIN\_INCL\_LINK$
$data\_block : \mathrm{seq}\ DATA\_ELEMENT$
$new\_data : \mathrm{seq}\ DATA\_ELEMENT$

For a complete explanation of the inner mechanisms of handlers, we refer to [Pen93].

7 Conclusions and future work

In this paper we have shown some issues in developing the data interchange procedures between different hypertext information systems with the aid of a reference model providing an intermediate data format. This procedure focuses on completing the Dexter Hypertext Reference Model to create the specification of a real hypertext system. This specification can then be used to implement an interchange mechanism for hypertext data between the system and any other hypertext systems for which an analogous model is created.

Our curiosity was aroused by the idea of using Dexter to model a system, RHYTHM, which is very different from those Dexter was originally designed from. This demonstrates that, with an elastic approach, Dexter can model a very large set of different hypertext systems and can be a valid support establishing *una tantum* or regular interchange procedures even between widely different systems. We therefore found Dexter a robust and trustworthy model for the specification of hypertext systems. Due to the rapid advances and changes in the hypertext field, these procedures will result indispensable in guaranteeing a lasting and useful life to information systems based on hypertext.

Furthermore, this paper has presented two principal results: in the first place, that few modifications are needed to enable Dexter to be a model for a larger class of hypertext systems. In the second place, that modelling a real system with Dexter does not require any further key modification, even if the real system is apparently quite different from the system implied by Dexter. All further modifications are mere additions regarding specific features not covered by the original model.

A lot of work still remains to be done in this field. We would like to create *RHYTHM2Dexter*, an interchange procedure between RHYTHM and Dexter, and use it with *IIF2Dexter* and *KMS2Dexter* created by Killough and Leggett [Kil90], so that our efforts are checked against some existing results. Using this procedure, we wish to prove that the interchange between RHYTHM and KMS or Intermedia is easy and possibly automatic. Any system that, designed to import data coming from already existing hypertext systems, establishes the interchange format with Dexter, will certainly simplify any transition.

APPENDIX: The Z Language

Z is a formal specification language whose notation has been widely accepted for the specification of several software systems in the industrial environment. The Z language is based on elements of set theory and first order logic.

Here we give an essential view on the notational elements used in this paper. The description of the evolution of a system state is specified through a box called *schema*. A schema is formed by a *declaration* part, in which elements of the state and relationships among them are included, and a *description* part in which state invariants, preconditions and sequences of actions are specified. These actions lead to a modification of the state of the system. Briefly:

SchemaName
declarations...
descriptions...

You can define operations or functions using the so-called *axiomatic definition* whose form is the following:

Declaration
Predicate

where the Predicate represents the body of the function. Moreover, the Z user is allowed to define recursive structures. An example used in this paper is the following:

$$\begin{aligned} BASE\_COMPONENT ::= {} & atom\langle\langle ATOM\rangle\rangle \\ | {} & link\langle\langle LINK\rangle\rangle \\ | {} & composite\langle\langle \mathrm{seq}\ BASE\_COMPONENT\rangle\rangle \end{aligned}$$

which can be read as follows: a BASE_COMPONENT is either an ATOM, or a LINK, or a composite, i.e., a sequence of other BASE_COMPONENTs. More detailed references about specific notational aspects may be found in [Spi92].

In order to facilitate the reading comprehension of this paper we introduce a brief list of the Z symbols used most.

$\mathbb{P}$	as power set	$\rightarrow$	as function
$\mathbb{F}$	as finite set	$\nrightarrow$	as partial function
seq	as sequence	$\rightarrowtail\!\!\!\!\!\!\nrightarrow$	as partial injection

Some of the advantages due to the use of the Z language are the production of precise, unambiguous specifications and the easy understanding of its notation.

Furthermore, Z is noteworthy also for capturing the dynamical aspects of the evolution of the system thanks to the use of schemata where the transformations of the system are specified.

References

[Aks89] R. Akscyn, F. Halasz, T. Oren, V. Riley, L. Welch: *Interchanging Hypertexts*. Panel of the Hypertext '89 Conference, ACM, Pittsburgh (PE), 1989

[Bie91] M. Bieber: *Issues in Modeling a "Dynamic" Hypertext Interface*. Proc. of the Hypertext '91 Conference, ACM, San Antonio (TX), 1991

[Cra90] G. Crane: *Standards for a Hypermedia Database: Diachronic vs. Synchronic Concerns*. Proc. of the NIST Hypertext Standardization Workshop, Gaithersburg (Md), 1990

[Dav92] H. Davis, W. Hall, I. Heath, G. Hill, R. Wilkins: *Towards an Integrated Information Environment with Open Hypermedia Systems*. Proc. of the ACM Conference on Hypertext ECHT '92, Milano (I), 1992

[Fri91] M.E. Frisse, S.B. Cousins, S. Hassan, WALT: *A Research Environment for Medical Hypertext*. Technical Briefing of the Hypertext '91 Conference, ACM, San Antonio (TX), 1991

[Gar91] F. Garzotto, P. Paolini, D. Schwabe: *HDM: A Model for the Design of Hypertext Applications*. Proc. of the Hypertext '91 Conference, ACM, San Antonio (TX), 1991

[Glu92] R.J. Glushko: *Hypertext and Multimedia for Managers*. ECHT '92 Tutorial, Milano (I), 1992

[Grø92] K. Grønbæk, R.H. Trigg: *Design Issues for a Dexter-based Hypermedia System*. Proc. of the ACM Conference on Hypertext ECHT '92, Milano (I), 1992

[Hal90] F. Halasz, M. Schwartz: *The Dexter Hypertext Reference Model*. Proc. of the NIST Hypertext Standardization Workshop, Gaithersburg (Md), 1990

[HKo90] S.L. Hardt-Kornacki, L.M Gomez, J.F Patterson: *Standardization of Hypermedia: What's the Point?* Proc. of the NIST Hypertext Standardization Workshop, Gaithersburg (Md), 1990

[Kil90] R. L. Killough, J. J. Leggett: *Hypertext* i*Interchange with the Dexter Model: Intermedia to KMS*. Technical Report TAMU-HRL 90-002, Texas A&M University, 1990

[Lam92] S. Lamberti, C. Maioli, F. Vitali: *Some Modifications to the Dexter Hypertext Reference Model*. T.R. n. 7/115, P.F. Sistemi Informatici, CNR (Italy), 11/92

[Mai91] C. Maioli, F. Vitali: *Authoring tool development with the RHYTHM hypertext system*. AICA 91, Siena, 9/91

[Mai92] C. Maioli, F. Vitali: *Anchors and Paths in a Hypertext Publishing System*. AICAText'92 Proceedings, AICA, Milano, 3/92

[Mai93] C. Maioli, S. Sola, F. Vitali: *Wide-Area Distribution Issues in Hypertext Systems*. ACM SigDoc '93 Conference Proceedings, ACM, Kitchener (Canada), 1993

[Nel90] T.H. Nelson: *Literary machines*. 90.1, Mindful Press, Sausalito (CA) 1990

[Ore87] T. Oren: *The Architecture of Static Hypertexts*. Proc. of the Hypertext'87 Conference, ACM, Chapel Hill (NC, USA)

[Ore90] T. Oren: *Towards Open Hypertext: Requirements for Distributed Hypermedia Standards*. Proc. of the NIST Hypertext Standardization Workshop, Gaithersburg (Md), 1990

[Pen93] W. Penzo, S. Sola, F. Vitali: *Further Modifications to the Dexter Hypertext Reference Model: a Proposal*. Technical Report, Laboratory of Computer Science, University of Bologna, Italy, 1993

[Sch90] H.A. Schutt, N.A. Streitz: *HyperBase: A Hypermedia Engine Based on a Relational Database Management System*. ECHT'90 Conference Proceedings, Cambridge Press, Versailles (F), 1990

[Spi92] M. Spivey: *The Z Notation. A Reference Manual. Second Edition*. Prentice-Hall, 1992

[VDP90] H. Van Dyke Parunak: *Reference and data Model group (RDMG): Work Plan Status*. Proc. of the NIST Hypertext Standardization Workshop, Gaithersburg (Md), 1990