DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

ONDERZOEKSRAPPORT NR 9802

A BROWSING PARADIGM AND APPLICATION MODEL FOR IMPROVED HYPERMEDIA NAVIGATION

by

W. LEMAHIEU

Katholieke Universiteit Leuven

Naamsestraat 69, B-3000 Leuven

ONDERZOEKSRAPPORT NR 9802

A BROWSING PARADIGM AND APPLICATION MODEL FOR IMPROVED HYPERMEDIA NAVIGATION

by

W. LEMAHIEU

D/1998/2376/02

A browsing paradigm and application model for improved hypermedia navigation

Wilfried Lemahieu

January 1998

Abstract

The advantages of hypermedia systems are often depicted in comparison to the rigid linear structure of a book. In this paper, both a hypermedia application model and a browsing method are presented that combine the best of both worlds; while holding on to the modelling richness and navigational freedom of hypertext, the use of a (partially) linear browsing strategy like in books greatly helps to reduce user disorientation. First we situate hypermedia within the general context of data storage and retrieval systems. We then address shortcomings of current hypermedia applications and suggest how imposing a linear path upon the data results into a new navigational paradigm that improves orientation and ease of navigation in a hypermedia environment. After that, we deploy a hyperbase model that supports this new view of browsing and describe the framework for an accessory application. As a conclusion, an overview is provided of the advantages this methodology offers, both for the application developer and for the end user.

1. Preliminary remarks

1.1. Aims of the application model

The hypermedia system and application models that will be developed throughout this paper, are primarily intended to benefit the end user, by providing a means to easily take his bearings and navigate fluently through the information space. However, we believe that our approach also facilitates application development and maintenance. These will be tackled sideways, while our main focus remains upon the end user.

The target of our immediate research is an "empty" application shell that can be filled with any multimedia data in order to return a *read-only*, *stand-alone* hypermedia application. Furthermore, we believe that future work will prove the model's potential to be expanded to a *distributed* hypermedia environment that allows extremely easy maintenance of its hypermedia link structure, with the added bonus of improved navigation. The absence of almost any session information makes a WWW server application an obvious field to explore.

1.2. Compliance with the Dexter hypertext reference model

Although it was not our specific concern, it will appear throughout the text that the hypermedia system model we developed fits rather well within the Dexter framework. This model is certainly not meant as a substitute for the Dexter model, rather as a complement; where Dexter mainly focuses on low-level hypermedia *system* modelling, our model aims at *application* development, particularly in combination with a relational database environment.

1.3. Terminology

As the same concepts more often than not are rather vaguely defined or plainly cover different charges in the database and hypermedia worlds, we will first provide a list of expressions used throughout this paper and the meaning we have attributed to them.

- *Data unit*: we will call a data unit any object that is directly addressable and referable in an information system model. This means that a data unit should always be associated with a unique identifier. Depending upon the environment, a data unit will be an entity in an E.R. model, a tuple in a relational database, a node in a hypermedia system, a page in a book, an object in an O.O. model,... A data unit will always be the representation in a data model of a 'thing' from the real world. In some environments, like O.O., data units may be composite, in which case they contain other data units, each of which is also equipped with a unique OID.

- *Node*: a node is a data unit in the specific context of a hypermedia environment. Each node has a unique ID within the hyperbase. Some hypermedia models allow nodes to be composite objects, so they may contain other nodes. For reasons that will become clear in section 6.1, the model proposed in this paper does not allow for nodes to contain other nodes. A node may very well be a complex object, in that it consists of several components, but these components cannot be data units. I.e. the components of a node cannot be individually referred to from outside the node. It may be possible that the node presents different components when accessed, depending upon *why* and *from where* it was accessed, but it is the responsibility of the node to decide what information will be shown: it is impossible to directly refer to the internal contents of a node, much like information hiding in the O.O. paradigm.

- *Current node*: this is the node most recently accessed and currently presented on screen. Although many nodes may be present in the internal memory cache of a hypermedia application, there will always be only one current node at a given time. This node determines which nodes are accessible for the next navigational step, since navigation is only possible to nodes that are linked to the current node.

- *Relationship versus relation*: we use the term *relation* in the broadest possible meaning: one data unit is related to another if both 'have something to do' with each other. If this relation can be modelled into an E.R. model, we will call it "relationship". A *relationship type* denotes the relation between two entity types in an E.R. model and the term is only used in this strict E.R. context, whereas the word relation retains its meaning from real life.

- *Link*: while the term 'relation' is used in a *semantic* context: a relation expresses the fact that one data unit 'has something to do' with another data unit, the term 'link' is used in a *navigational* context: one is able to travel from one node (the *link source*) to another (the *link destination*) along a link. A link might be uni- or bi-directional and will always be the consequence of a relation: it is only useful to link two data units if these are in some way related to one another.

- *System versus application*: We will distinguish between on the one hand a *database system* or *hypermedia system*: the software that manages the data (respectively stored in a database or hyperbase) and on the other hand a database or hypermedia *application*: a software component that manipulates the data and presents them to the end user and that receives services from the aforementioned *system*.

2. Hypermedia systems as data storage and retrieval systems

Throughout this text, a hypermedia model will often be compared to an entity-relationship or relational model. We will use similarities to explain various concepts, and we will indicate differences between them to advocate the use of a proprietary development methodology for hypermedia applications. Since hypermedia systems are essentially data storage and retrieval systems, we will put them in this perspective and compare them to the main representative of the class: database systems. It will appear that hypermedia systems have a number of particular problems and opportunities that will make traditional (database) application development methods inappropriate for hypermedia development, although we can certainly learn from the comparison between both types of systems.

Hypermedia and database systems have in common that they are both meant to store and retrieve data units in one form or another along with the *relations* (in the broadest possible meaning) between these data units. However, it is useful to mention some particular properties of hypermedia systems, that will prove to be important later on in this text:

- Storage of *navigational information*
- Storage of *presentation specifications*
- A very narrow coupling between system and application
- Storage of unstructured (in the database meaning of the word) data
- Distinction between data and access criteria
- Explicitly defined relations between data units
- Relations on *instance level*
- Tailored to end-users with little or no experience

2.1. Storage of *navigational information*

Databases essentially contain data and the relationships between these data. These relationships model the semantics of mutual dependencies between data units. A hypermedia system not only stores data and relations, but these relations also take the explicit interpretation of *navigational information*. A hyperbase stores the relations between data in the form of *links*, and these links not only have a semantic meaning, but they also model the potential of navigation between the data units.

While each data unit stored in a database system is accessible at any time (not counting locks or other DBMS controlled factors), this is not true for hypermedia systems. The set of data units that is available for access at a given time depends upon two factors. The first one is a variable whose value continually changes at runtime: it is the *current node*, the node most recently accessed. The second factor consists of *navigational information* stored within the hyperbase. Only nodes that are linked to the current node are accessible at a certain moment in time. Thus, by storing links into the hyperbase, the developer can influence the paths that are open for navigation to the end user.

2.2. Storage of *presentation specifications*

Not only navigational information is stored into a hyperbase, also *presentation specifications*, information about how the data should be displayed upon the screen might be stored along with the data. This might be information about fonts, on screen positions of objects, colours, the size of drawings,... Thus the hyperbase contents influence the on screen presentation of the data units to a certain extent.

This is not the case in a database environment, where it is solely the database *application*'s responsibility to present the data on screen. The data units in a database do not contain information about how presentation should be carried out.

2.3. A very narrow coupling between system and application

In the database world, DBMS and application are only loosely connected in most cases. The application feeds the DBMS the criteria of the data to retrieve. The data that satisfy the search criteria are passed back to the application. It's the application's responsibility to present the data to the user.

Even the most rudimentary hypermedia environments like HTML documents, contain both navigational information and presentation specifications like described in sections 2.1 and 2.2. This has as a result that hypermedia system and application need to be much more interwoven than their database counterparts. The hypermedia *system* retrieves and passes presentation specifications that have to be interpreted by the *application.* Furthermore, the application can only access data units that are approved of by the hypermedia *system*, as one of the factors that influence navigation are the links stored within the hyperbase.

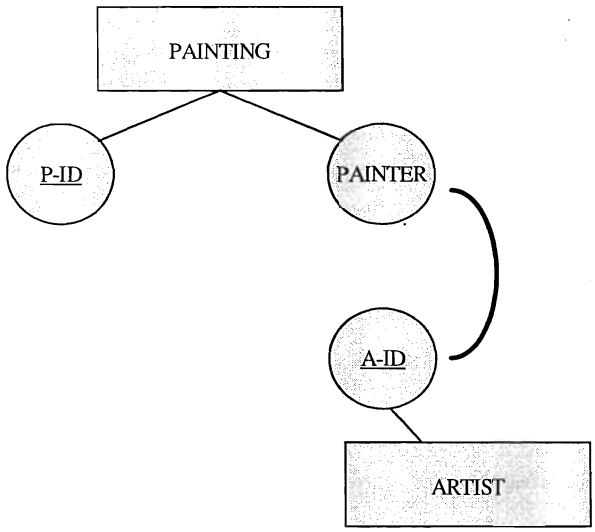
Attempts have been made to uncouple hypermedia systems and applications, but in practice most existing systems and applications are integrated into one piece of software.

2.4. Storage of unstructured data

Databases store only data 'structured' in attribute/value form. Hyperbases, like real multimedia systems, should be able to store data in different formats and belonging to different media like text, graphics, bitmaps, MIDI data, sound samples and video. These data may well have an internal structure, (e.g. a text document might be structured into chapters and paragraphs) but they don't necessarily dispose of the attribute/value structure required in a database environment.

2.5. Distinction between data and access criteria

Attributes in an E.R. (or relational) model serve three purposes: First they are used to *describe* the data unit they belong to. Second, they may be used as an *access criterion* to select the corresponding data unit. The third purpose is to *relate* data units to each other: when attributes are used as a foreign keys.



E.g.: The entity type PAINTING with P-ID as primary key and the attribute PAINTER as a foreign key referring to the entity type ARTIST. This attribute PAINTER serves three distinct purposes in a relational model:

- Describe an aspect of the painting, in other words: be part of the *information content* of a data unit of the type PAINTING
- Serve as an *access criterion* to select instances of the type PAINTING
- *Relate* a data unit of the type PAINTING to a data unit of the type PAINTER

All values in a relational database fulfil any of these three purposes, depending on the query, thus on the desired information. In a hypermedia system, these three functions are separated. Information content and access criterion are tackled in this section, the third one is tackled in section 2.6.

As a consequence of the diversity of formats and the lack of the attribute/value structure of hypermedia data, hyperbases will need to have a means of selecting data units, without using the data itself as access criteria. There will be a clean separation between the information content of a unit of data (which is an intra-node property) on the one hand and the means to access this unit of data, the *link structure* of the hyperbase, on the other hand. The latter is an inter-node property. We will return to the discrepancy between information content and access criterion in section 6.6, when we deploy a formal hyperbase model.

2.6. Explicitly defined relations between data units

The third function of an attribute in E.R. is to relate one entity type to another. Both database and hypermedia models allow data units to be *related* to each other. In a relational database, a foreign key is included within a tuple to relate this tuple to another one. However, the relation between both mainly concerns semantic constraints for update and delete actions, its influence during consultation is marginal. Indeed: an attribute does not need to be a foreign key to define a join between tables, leaving the possibility to the user to relate data units to each other that were not meant to be related by the developer, even if these relations are completely absurd. E.g. a possible query could be: 'select all employees whose age equals the shoe-size of their manager'.

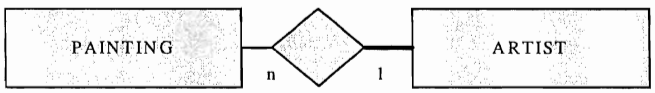
In a hypermedia environment, a *relation* between data units also causes these units to be *linked*, hence relations also have navigational consequences. Since the explicit definition of navigation paths is a key concept for hypermedia, links are only allowed to be derived from relations that are explicitly defined by the developer. The user doesn't have the freedom of relating anything to anything like in the age - shoe-size example above.

2.7. Relations on *instance level*

In the E.R. (and relational) model, relationships are always defined between entity *types*: one could define the relationship type 'is painted by' between respectively the entity

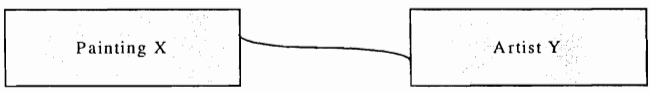
types 'PAINTING' and 'ARTIST'. It is not possible to relate two *instances* to each other, unless as an instantiation of a relationship type that has been defined between the respective entity types to which the instances belong.

In hypermedia systems, the notion of such a thing as a *type* hasn't always been present. The first systems consisted of untyped nodes and links. Nowadays, many models support the notion of *node types* and *link types* to a certain degree - the advantages of which will become apparent later on in this text. Nevertheless, the need remains for the ability to define a relation between node-types, similar to a relationship type in E.R., but also between node *instances*, where the particular meaning of the relation does not allow it to be defined on node-*type* level. For example: in a hyperbase that contains nodes of the types PAINTING and ARTIST, a relation (and consequently a link) 'is painted by' could be defined between these types, similar to a relationship type between the entity types PAINTING and ARTIST in an E.R. model.



This means that every painting is created by exactly one artist, and every artist can be the author of several paintings. We can structure this information into an E.R. or O.O. model. The relationship 'is painted by' is a relationship between the two entity *types* PAINTING and ARTIST, since every painting is painted by someone, it is a property of being a painting. We call this *structured* information, since this is the kind of information we can model into a database.

Now, suppose the textual description that is included in the data unit 'artist X' of a hyperbase mentions that seeing a picture of painting Y (by artist Z) was his immediate stimulus to take up the brushes. This also kind of *relates* artist X to painting Y, but it's not a property of the *types* ARTIST and PAINTING, it's a relation between two well-defined *instances* of these types, impossible to model on type level.



This is *unstructured* information, we cannot model it into a database model. In the hyperbase, however, if the information content of the data unit 'artist X' includes this information, it is useful to model a link between the two nodes. Such a link will only exist between these two node instances, not between other instances of the same type.

This second example shows a very different kind of link: where the first one is a consequence of the data *structure* of the underlying model, hence the term *structural link*, the second is just the expression of an *ad-hoc* relationship between two node instances, hence the term *ad-hoc link*. We will come back to this issue in section 6.4.

2.8. Tailored to end-users with little or no experience

The property of hypermedia systems that navigation can already be designed during the data modelling phase, makes them a perfect choice to 'guide' end users through a large information system. The developer has much more control over end user browsing than in the case of database applications. The target user for such applications will often be someone unfamiliar with the application, possibly with little or no computer experience, often (but not necessarily) with read-only access to the hyperbase.

While this may seem a rather futile remark, it most certainly is not. Obviously, it requires the user interface of a hypermedia application to be as intuitively clear as possible. But the impact is much larger than the user interface alone, it will also put certain demands upon the data model. *The* problem with hypermedia applications is user disorientation, and since the model also stores navigational information, the quality of the data model will have a very important influence upon how well the end user is able to attain the information he desires. Whereas the underlying model in database applications remains more or less transparent to anyone but the development team, a hypermedia model should be sufficiently comprehensible to the end user, as well as offer the (navigational) support necessary to make orientation and browsing as efficient and as satisfactory as possible.

As a conclusion, we can state that a hypermedia model has a number of particularities that call for an apt approach: in many ways it is more elaborate than a database model, in that it incorporates presentation and navigation aspects.

3. Where current hypermedia applications fall short

3.1. Hypermedia navigation compared to linear browsing

To highlight the advantages of hypermedia applications, comparisons are often made to books. Books are said to be *linear* information systems: their data units (pages) are organised in a fixed order, one after the other. Hypertext offers the possibility to break through this linear constraint and organise data in more complex structures. This allows the data to be accessed following different possible paths, depending on the user's preferences and interests. One should be able to 'freely navigate through the hyperspace'. Unfortunately, 'freely navigate' comes down to 'wander without a clue' in many a case. User disorientation is the Achilles tendon of all hypermedia applications. Two questions sum up the problems related to hypermedia navigation: "Where am I?" and "Where can I go from here?". These questions represent the difficulty to locate the current node within the whole hypertext structure and to determine the navigational options that are open from the current node. To accommodate user orientation, most hypermedia tools include maps, graphs and overviews which relax the problem to a certain extent, but we believe the main cause for disorientation is exactly this *absence of a linear structure*.

3.2. Linearity and user (dis)orientation

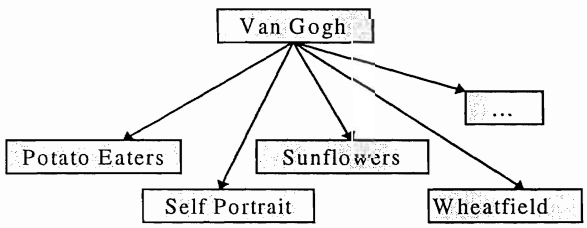
Indeed, the linearity of a book constrains navigational freedom, but also prevents the reader from 'loosing the thread'. Reading a book never causes the navigational difficulties one experiences with hypermedia applications. Linearity allows one to determine one's position within the collection of data units: the fact that a data unit (a page) only contains two links, one to the previous data unit and one to the next one,

transforms this collection into a one-dimensional space. It allows a 'linear' reader to always ascertain his position: which data units he has already visited and which ones he has not. Also the second question 'where can I go from here?' becomes trivial, since the options are restricted to only two links for each data unit: forward or backward, of course at the cost of navigational freedom.

This is not the case in hypermedia applications. After only a few browsing steps, the user looses track of things and is condemned to wandering haphazardly. A hypermedia application seems to be fit for 'casual' browsing through nodes, following a few links and picking up a bit of information here and there. But it doesn't really allow for a thorough study of a certain topic, where it is necessary to exhaustively read everything there is to read that is related to this topic. In that case, a linear structure is by far the better. The linear structure is the leading thread that prevents the reader from getting lost. Breaking through the linear structure of a book by tearing out all the pages and allowing them to be ranked in random order will certainly not improve reading comfort.

3.3. Poor navigation and guided tours

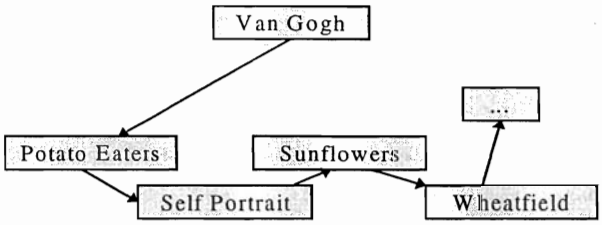
A second shortcoming of many contemporary hypermedia applications is their poor navigational structure, resulting in unnecessary browsing steps, at the risk of disorientation. Let's return to our PAINTING - ARTIST example. Suppose we would like to visit all nodes describing a painting by Van Gogh. Most applications would present the following link structure:



As a result, visiting all paintings comes down to selecting a painting, returning to the node 'Van Gogh', selecting another painting, returning to 'Van Gogh' etc. Navigation is

only possible in a tree-like fashion. If the information required is more than two levels deep, browsing becomes very tiresome and unsatisfactory.

More advanced applications add the facility of so-called 'guided tours', where all nodes pertaining to a common subject are chained together (thus in a *linear* structure!), allowing them to be browsed one after the other.



This certainly improves navigational comfort, but at the cost of a considerable overhead (links have to be added for each tour in which the node participates) and, even more important, poor maintainability. Indeed, suppose additional tours exist linking together paintings with a common theme, from the same era, belonging to the same museum,... Adding or removing one painting implies updating the 'linked list' structure of each guided tour, which becomes an impossible task for even a medium-sized hyperbase, resulting in inconsistency, dangling links, etc.

We can conclude that it is next to impossible to solve this navigation problem on hyperbase *contents* (the links stored within the hyperbase) level. It should be the hypermedia *environment* that is flexible enough to allow for the necessary navigational freedom. Besides, the structure of a guided tour introduces redundancy into the hyperbase, since linking nodes into a guided tour implies they have some property in common. However, in the example above, the common property of 'being painted by the same artist' is already established within the respective links from each PAINTING to its ARTIST. Thus, it would be possible for an intelligent hypermedia system to infer this knowledge and generate guided tours *at runtime*, without burdening hyperbase maintainability.

In our opinion, the key to more user friendly hypermedia applications consists of a combination of both navigational freedom and the ease of linear navigation. Hereby, the

concept of *at runtime* generated guided tours both improves ease of navigation and offers a linear path throughout (part of) the hyperbase to reduce user disorientation. One could compare this approach to an "intelligent book", that always maintains its linear structure, but constantly rearranges its pages according to the user's interests. To generate these guided tours, the relations between the nodes stored within the hyperbase are of utter importance. We will elaborate upon these interrelations in the next section.

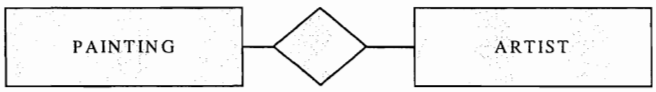
4. Relations and links

4.1. Direct and indirect relations

One could look at a hyperbase as a collection of data units that are interrelated. The relations in a hypermedia environment not only carry a *semantic* meaning like in the E.R. model, but also a *navigational* one: they are represented as *links* within the hyperbase. So a link provides a path between two nodes that are, in some way, related to one another. As we see it, these relations/links between nodes always fall into one of two categories, each with its own specific properties. Acknowledging the *semantic* distinction between *direct relations* and *indirect relations*, as we will call them, entails a new look upon their *navigational* interpretation, which results in easier orientation and improved navigation.

4.1.1. Direct relations

If there exists a *direct* relation between two data units, the two units contain additional information about one another. We can compare this to an instance of a relationship type in an E.R. model. E.g. the relationship type 'is-painted-by' between entity types PAINTING and ARTIST:

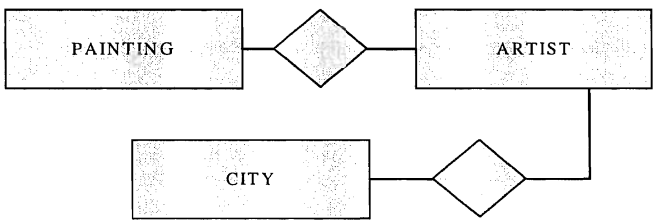


One of the instances of this relationship type relates the data unit 'Sunflowers' to the data unit 'Van Gogh':

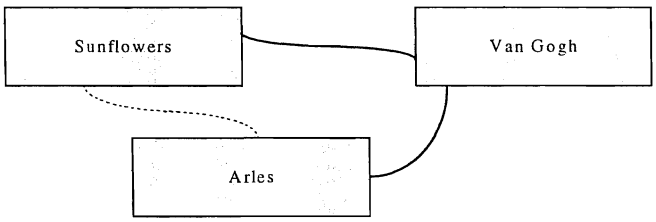
Both of these data units provide additional information about one another: there exists a *direct* relation between them. In a hypermedia environment, the *semantic* aspect of this relation will also have a *navigational* counterpart: there will be a *link* between the nodes 'Sunflowers' and 'Van Gogh'.

4.1.2. Indirect relations

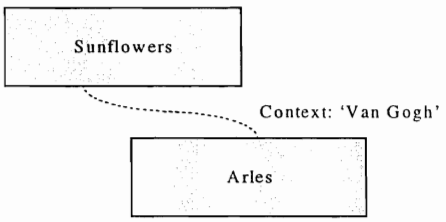
An *indirect* relation between two data units indicates that they both have a direct relation with a third unit in common. They both contain additional information about this third data unit. We call this last unit the *context* of the indirect relation. We will illustrate this again starting from the E.R. model consisting of the relationship types 'is-painted-by' between PAINTING and ARTIST and 'lives-in' between ARTIST and CITY.



Instantiation could deliver the following data units and relationship instances:



'Sunflowers' and 'Van Gogh' respectively 'Van Gogh' and 'Arles' are directly related. This results into an indirect relation between 'Sunflowers' and 'Arles' with 'Van Gogh' as the context.



It would be useful for our hypermedia application to provide a link between 'Sunflowers' and 'Arles', but only in the case where we are exploring information about 'Van Gogh': such a link is only required in this particular *context.*

4.2. Link properties

Since a link is the navigational reflection of a relation between two data units, the distinction between direct and indirect relations is carried over to the links derived: we discriminate direct links from indirect links and demonstrate how both link types have different properties satisfying different navigational requirements.

Direct links:

- Result from a direct relation between two nodes
- The two linked nodes describe each other
- This kind of link always exists and is independent of the context
- Direct links are static and are stored explicitly into the hyperbase

Indirect links:

- Result from an indirect relation between two nodes
- The two linked nodes each describe a third node (the context)
- This kind of link only exists within and depends upon a certain context
- Direct links are dynamic, they are generated at runtime according to the current context

Note that, although many data modelling techniques allow for ternary and higher-order relations, we will restrict ourselves to binary relations and links. The *semantics* of higher-order relations might be easily comprehensible in a data model, but the implications of ternary links become blurred in terms of *navigation*, making them unsuited for hypermedia modelling. We will now elaborate upon the properties of both link types.

4.2.1. Direct links

A direct link results from a direct relation between two nodes. Providing a means of navigation between these nodes is useful, since (part of) the content of the one node is also relevant to the other. Direct links have a permanent character as they are a consequence of the conceptual data model behind the hyperbase. Whenever any one of two directly linked nodes is the current node, the other one is accessible regardless of the context at that time (we will provide a formal description of the notion 'context' in section 7). Direct links are present in any hypermedia application. They are stored explicitly into the hyperbase, as they represent lasting relations between data units.

4.2.2. Indirect links

This type of link results from an indirect relation between two nodes, which also implies the presence of two direct relations/links to a common third node. Navigation between the two indirectly linked nodes is only useful within the context of this third node. If this

third node is the focus of attention, the two other nodes both supply an additional portion of information about it and can be browsed sequentially.

Indirect links not only depend upon the data model behind the hyperbase but also upon a run-time variable: the current context. Thus, indirect links cannot be stored within the hyperbase, they are to be created *dynamically* upon change of the current context. When this context changes, indirect links are destroyed and new ones are created according to the new context.

5. Towards a navigational paradigm

5.1. An improved browsing strategy

As previously stated, we believe that the ideal browsing environment would be a combination of the best of both worlds: it should allow the user the navigational freedom he experiences with conventional hypermedia applications, but also offer a linear path throughout (part of) the information space to fall back to. This path should depend upon the user's interest and change dynamically with his focus of attention. Thus, the links that are open for navigation at a certain time should not only be influenced by the user's current position within the hyperbase (the *current node*), but also by the broader backdrop (the *current context*) against which he is browsing for information. The current context is the variable that should allow the system to suggest such a guided tour that takes the user's focus into account.

This brings us to the idea of a set of nodes, that all have a certain topic in common and that can be browsed sequentially, always clicking a "next" button to select the next node in line. Apart from that, one should be able to explore each node visited, randomly follow links to other nodes that are connected to the current node without loosing one's position within the tour, and always with the possibility to resume the linear path one was following in the first place. If browsing the hyperbase results in a new topic of interest, the application should provide a new tour, with all nodes related to this new topic included. This allows the user to follow a dedicated path with the possibility of

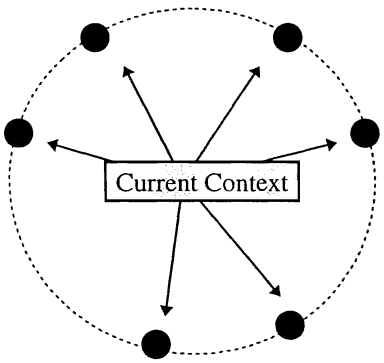
exploring extra information around a certain topic, without loosing a sense of orientation: one is able to digress without loosing the thread of the discussion.

5.2. A guided tour depending upon the current context

In conventional hypermedia applications, the *current node* is the only variable that determines which nodes are accessible at a given time. One can only navigate to nodes that are linked to this current node. These (direct) links are all static and are stored within the hyperbase. Introduction of a second variable, the *current context* allows for dynamic link creation: a *guided tour* is defined by generating indirect links between all nodes related to the current context.

We will define this current context as one single node that is selected by the user as his current focus of attention. As a data unit, this node represents an object from real life. Hyperbase navigation concentrates upon searching all information related to this object. This definition will be refined in section 7, after we examined node and link typing.

A guided tour results from the current context as follows: when a node is selected as the new context, all nodes directly related to it are collected and ordered alphabetically according to a 'node descriptor' field (see section 8.2) or some other criterion. Indirect links are run-time generated between all successive nodes, defining a chain of nodes directly linked to the current context and indirectly linked to each other. Throughout the rest of this paper, we will represent a direct link/relation by an arrow and an indirect link/relation by a dotted line.



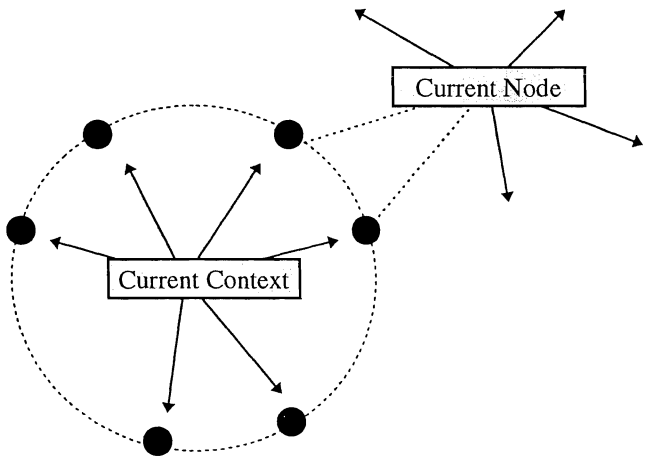
5.3. Navigation

Our navigational paradigm combines linear browsing along a guided tour with completely free navigation along direct links. The user determines the context of the guided tour: whenever he decides to focus his attention on the current node and to explore all things related to it, he can select the current node to become the current context. The previous context is deleted and navigation is centred around the new context. Although each subsequent navigational step causes another node to become the current node, the current context is preserved until it is explicitly changed by the user. Where the current node is a short-term factor that changes with each step, the current context can be seen as a long-term factor that 'glues' the various visited nodes together and provides a background about which common topic of these nodes is being explored.

At any time during a session, the hyperbase consists of both the direct links that are always present and the indirect links between subsequent nodes under the current context. The user has the choice between three navigational options:

- Follow a *direct link* to a node directly related to the current node
- Follow an *indirect link* that leads to the next node related to the current context
- Select the current node to become the new context

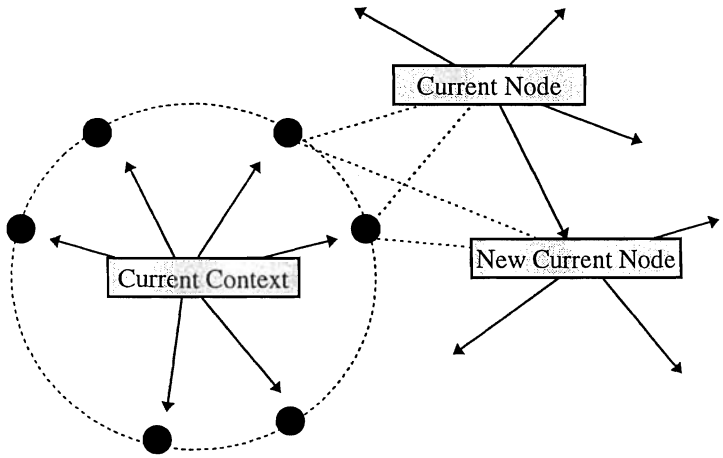
Schematically, we can depict a navigation situation as follows:



The current tour is represented by the circle. All nodes upon the circle are directly linked to the current context and indirectly to their predecessor and successor. The current node may or may not be part of the current tour. The current node has its own direct links, as well as indirect links leading back to the current tour.

5.3.1. Following a direct link

This comes down to exploring information that is directly related to the current node. It is similar to conventional hypermedia navigation. Here, the current context is of no importance, since direct links are context-independent. When a direct link is followed, the newly accessed node becomes the current node. The current context is not affected, nor is the current position within the guided tour.

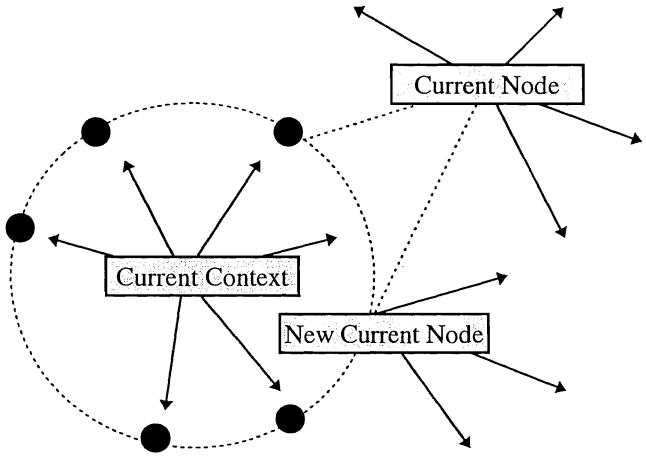


Following a direct link is represented as a movement independent of the circle. Note that the indirect links remain unchanged.

5.3.2. Following an indirect link

This means moving forward or backward within the guided tour that is generated by the current context. Following an indirect link implies accessing another node in a string of nodes directly related to the current *context*. This node may or may not be directly

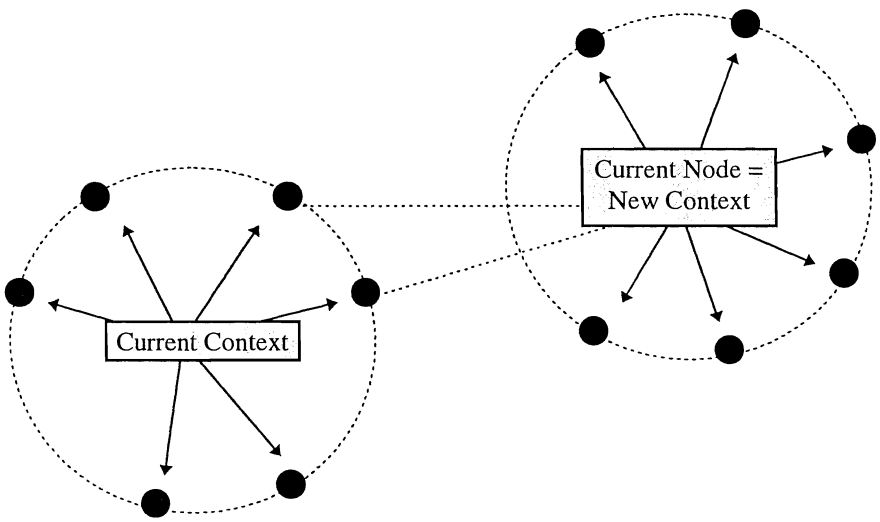
related to the current *node*. When accessed, this node becomes the new current node. In practice, an indirect link will be selected by pressing a "next" or "previous" button, where the system calculates the correct destination at runtime.



Following an indirect link causes a movement along the circle that represents the guided tour.

5.3.3. Selecting a new context

A context change causes the current node to become the current context. This reflects the user's decision to concentrate upon the current node as a new topic of interest. All indirect links are destroyed and redefined around this new context.



Changing the current context is represented by moving the circle so that the current node, which is also the new context, becomes the centre.

5.4. Conclusion

The at runtime generation of indirect links resulting into guided tours sensitive to the current context, allows for different possible linear paths throughout the same information space. Along with this additional linear aspect, the user retains all navigational freedom from conventional hypermedia systems. Obviously, the methodology just described only offers a very basic means of defining guided tours. Only constructions along the lines of 'all nodes that have something to do with Van Gogh' are possible. More complex expressions like 'All museums that contain paintings by Van Gogh' require additional complexity within the data model. We will first address some data modelling issues including node and link typing, after which we extend the strength of our navigational model.

6. The data model

Until now, the only information mentioned to be stored into the hyperbase were the nodes and (direct) links. In this section, we elaborate upon their definition and develop a formal data model that should allow for the design of hypermedia systems that support the navigational paradigm from section 5.3. In the first place, the model is aimed at ease of navigation and intuitive clarity for end users, but also design and maintenance should benefit. The following aspects will be treated:

- Nodes
- Links
- Node typing
- Links on node type level
- Link typing

- Incorporation of attributes
- Link directionality
- Minimum and maximum cardinality
- Inheritance
- A schematic representation

6.1. Nodes

The nodes are the data containers of the hypermedia model. Each node is a data unit that is treated as an atomic entity and represents a real world object. Although a node may (and probably will) have a complex internal structure, this structure is beyond the scope of our model. A node may contain heterogeneous components, connected by internal links. However, external references to the node are always made to the node *as a whole*. Internally, a node may contain the intelligence to react differently to different types of links (see section 8.1), but from the outside, a node is seen as an indivisible data unit to which links can be attached.

A consequence is that nodes are not allowed to be composite objects in the O.O. sense of the word, in that they are not allowed to contain other nodes. Relations of the type 'is part of' have to be defined as links just like any other relation. We won't argue that such aggregations may be very useful for conceptual data modelling. However, the data model visible to the end user becomes less comprehensible. Indeed, the use of composite objects defines an additional structure of relations between the data units, apart from the ordinary link structure. Since hypermedia systems have the particularity that navigation depends upon the interrelations of their data units, it is important to keep these interrelations as uniform as possible.

6.2. Links

A link is a one to one association between two nodes. As already explained in section 4.2, only binary links are allowed. Upon definition of a link, an inverse link is

automatically defined. A link and its inverse consist an indissoluble pair. Each link has a direction and offers an access path from its *source node* to its *destination node*. Source and destination are reversed for the inverse link. Link directionality will be treated in section 6.7. First, we will introduce the concepts of node and link typing.

6.3. Node typing

The definition of classes of nodes can be very useful, as well for development purposes as to the end user. The general procedure to define this kind of abstraction between data units is to look for one or more common properties and to define a generic type that explicitly incorporates these common properties. In the case of hypermedia data units, the only possible properties of nodes that allow them to be classified into respective types, are the links to other nodes. Thus, definition of a node type comes down to specifying what links are to be associated with its instances. Therefore, we will also have to define *link types*, which we will tackle in the next two sections. For now, we will suffice with defining a class of nodes as a collection of nodes pertaining to the 'same kind of real world objects', e.g. ARTIST, MUSEUM, PAINTING, ...

Node typing combined with link typing is advantageous to the hyperbase developer, since it allows for system-based *referential integrity and completeness checking* in the same way as in a database environment (e.g. 'is every PAINTING linked to an ARTIST?'). Also hyperbase *maintainability* improves, as the system is able to suggest the appropriate link types upon definition of a node instance. Also the *design* of a node is facilitated, since the *use of a template* for similar nodes speeds up node design and link definition. Moreover, the resulting *uniform layout* for similar nodes enhances user comprehension of the underlying model, which in turn greatly improves ease of orientation.

6.4. Links on node type level

Thus far, links have always been defined on node instance level, this was said to be one of the properties that distinguish hypermedia models from database models (see section 2.7). However, the definition of node types allows for the definition of links on *node type level*, next to links between *node instances* whose definition is not valid on a higher level. Therefore, we will distinguish between *structural links* and *ad-hoc links*, which are defined respectively among node types and among single node instances.

6.4.1. Structural links

These are links that represent a relation between two well-defined *node types*, e.g.: between PAINTING and ARTIST. A structural link is always an instance of a *link type*, which we will define in section 6.5. All nodes of the same node type share the same set of structural link types. Structural links match relationship instances in an E.R. model, thus are the result of *structured information* within the hyperbase. They make out the backbone of the hyperbase structure, hence the name structural link.

6.4.2. Ad-hoc links

Ad-hoc links are defined between two node *instances*, while their respective node types are irrelevant. If a link is defined that is only meaningful to two specific nodes and not to other nodes of the same type, an *ad-hoc* link is defined on node instance level. Links like these will often be anchored within text fragments embedded within one of the linked nodes. This information is not structured in the database sense of the word, like it is the case with structural links.

6.5. Link typing

Where structural links are the equivalent of relationship instances in an E.R. model, we will define a (structural) link type as the equivalent of a relationship type. However, the equivalence is not complete, since a link type is not defined *between* node types (like a relationship type is defined between entity types). Contrary to an E.R. relationship type, the same link type can exist between different pairs of node types.

Rather, a link type can be seen as a label that is attached to all similar structural links (not necessarily between nodes of the same types). A link type can be *attributed* to one or more node types, which means that the source node of each link instance should belong to any of these node types. The destination node types are defined by attributing the inverse link type.

An example: the link type "property of" can be attributed to the node type PAINTING as well as to the node type MUSEUM. Its inverse, "is owner of" can be attributed to the node types PERSON, CITY and MUSEUM. The reason for this approach is that it allows node types to be defined and link types to be attributed to them, without knowledge of other node types. The link type becomes the interface between node types, where these can be modelled separately without knowledge of each other. When new node types are defined, link types can be attributed to them without redefining the rest of the hyperbase. An additional advantage is that nodes of different node types can be linked to a node using the same link type, the relevance of which will be explained in section 7.

Thus, an instance of a link type links two nodes of types to which respectively the link type and the inverse link type are attributed. The equivalent of a binary E.R. relationship type would constitute the triplet (node type A, link type, node type B). If a link type is attributed to a node type, all nodes of this type can or must (depending upon cardinality, which will be tackled in section 6.8) participate in an instance of such a link. Link types are attributed to node *types*, so only structural links can be typed. We can illustrate the discrepancy between structural and ad-hoc links respectively direct and indirect links in the following table:

	Direct relation	Indirect relation
Between node types	(Structural) link type	Link type = 'Indirect'
Between node instances	Structural link (instance) Ad-hoc link	Indirect link instance

A link type is attributed to node types. The link type 'indirect' is attributed to all existing node types. Links between nodes are either structural links, in which case they are instances of link types, or ad-hoc links, in which case they are typeless. An indirect link has 'indirect' as its link type. Links of the type 'indirect' are generated at runtime.

Note that link typing and node typing are completely independent of each other; there may exist different link types between two node types and the same link type may exist between different pairs of node types. Section 7 shows how link types play a key role in the definition of more complex guided tours.

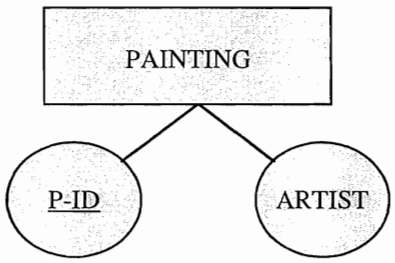
6.6. Attribute versus entity

As already stated in section 2.5, in the database world, attributes are information containers as well as access criteria for data units. The access criteria and the data themselves are one and the same. Since the data in hypermedia systems cannot be captured in such an attribute/value framework, information content of a data unit and access criteria are two different matters.

What happens upon translation of an E.R. model into a hypermedia model? An attribute/value pair can be treated in two possible ways, depending upon its use as either information content or access criterion. If the attribute/value pair is purely descriptive, i.e. it contains information about a certain data unit like a video clip or a text fragment does, it is incorporated into the node like any other component of the node. If, on the other hand, it is useful to treat the attribute as a selection criterion for the data unit it describes, an access structure in the proper hypermedia fashion has to be constructed: as

a *link* between nodes. Therefore, the attribute has to be transformed into a *node type*, with as many node instances as the attribute had different possible values. Each of these instances is to be linked to all nodes that represent E.R. entities with the corresponding attribute value.

As an illustration, let's return to the PAINTING - ARTIST example. Suppose the E.R. model contains the entity type PAINTING, with "artist" as one of its attributes.



If the name of the artist will only be presented in a field along with other information about a certain painting, it can be incorporated into the internal contents of the node. If, on the other hand, "artist" is to be used as an access criterion to select nodes of the type PAINTING, the attribute type has to turn into a node type. As many instances of the type ARTIST have to be defined, as there are different attribute values in the E.R. model. Nodes of the type PAINTING are then to be linked to a proper instance of the type ARTIST. This way of thinking clearly adheres to the O.O. 'identity based' concept, rather than to the E.R. 'value based' concept.

The hypermedia model doesn't distinguish between E.R. attributes-turned-into-nodes and E.R. entities-turned-into-nodes. Note that, in the E.R. model itself, the difference between attribute and entity mostly depends upon the scope of the model: if extra information pertaining to an artist is added to the model, the attribute type "artist" becomes an entity type itself.

6.7. Definition of link direction

In most hypermedia models, a link has a *source* and a *destination*. Directionality is useful for two reasons: first there is a *semantic* aspect, the same reason why association types are defined within an E.R. model: because the exact meaning of a relation might otherwise be confusing, e.g. for the relation 'is a parent of'. Second, because of the *navigational* aspect, where a source and a destination are inherent to each navigational step.

In our model, definition of a link (type) automatically effects into the definition of an inverse link (type). Only the source node of a link is defined, the destination is defined as the source of its inverse. So if a link is added to a node, the destination node type can be any node type to which the inverse link type is attributed.

6.8. Attributing minimum and maximum cardinality

Link cardinalities are attributed to the combination (source node type, link type). Cardinalities can vary for the same link type, depending upon node type. So the same link type can be optional for one node type and mandatory for another. Minimum cardinalities can be either 0 or 1, maximum cardinalities either 1 or n (note that a link *instance* is always one to one). Cardinality is only attributed at the source node. Instead of defining a "destination cardinality", cardinality of the inverse link type is used. Ad-hoc links are untyped and therefore don't have cardinalities defined. They are always one to one. A table with examples of each possible combination looks as follows:

Source node type	*link type*	*min. cardinality*	*max. cardinality*
Painting	is-painted-by	1	1
Painting	is-exhibited-in	0	1
All-round artist	has-painted	0	n
Museum	exhibits	1	n
Painter	has-painted	1	n

The link types 'is-painted-by' and 'is-exhibited-in' are the inverses of 'has-painted' and 'exhibits' respectively.

6.9. Inheritance

In our model, node and link types have been defined, without the possibility of defining subtypes/supertypes with potentially (multiple) inheritance of both link types and presentation properties. We think that this option is very valuable in the hypermedia modelling stage and for application development, but doesn't contribute much to the user's perception of the model. We preserve this topic for future work, when application development will be tackled.

6.10. Schematic representation of the model

It is impossible to capture this model within a graphic representation like the E.R. model, due to the fact that link types can be attributed to various node types, and are not defined between a pair of entity types like in E.R. Schematically, we can represent the type-level aspects of our model as follows, where a table with attributed link types is created for each node type:

Node Type X

Attributed Link Types	Inverse Link Types	Minimum Cardinality	Maximum cardinality
Link type A	A^{-1}	(0 or 1)	(1 or n)
Link type B	B^{-1}	(0 or 1)	(1 or n)
Link type C	C^{-1}	(0 or 1)	(1 or n)
...	...	...	...

Note that such a scheme can be defined for each node type independently. Only knowledge of the available link types is needed, not of other node types. So internal node design can be carried out for each node separately, without knowledge of other

node types. We will see in section 8.1 that node instances can be created using different media and formats, where the link database interfaces between these different node types.

7. Navigation revisited

So far, the context has been defined as one single node around which guided tours evolved. This only allowed for tours to be defined like: "all nodes related to Van Gogh". We will now expand the definition of context to allow for tours to be fine-tuned around certain *types* of links, and tours that represent the *composition* of multiple links, allowing for *transitive* relations between nodes. We will be able to define tours like 'all paintings by Van Gogh', excluding nodes that are in any other way related to Van Gogh than by a 'painted by' link type. Also possible will be the following kind of tour definition: 'all museums that exhibit at least one painting by Van Gogh', without a direct link between the node types MUSEUM and ARTIST, but making use of the composition of link types between MUSEUM and PAINTING respectively PAINTING and ARTIST.

Link typing will play a crucial role in the definition of guided tours. Therefore, we associate each link type with a unique identifier, a *link label* that is shared by all links of the same type. This link label will be used to refer to a link *type*, like a link ID is used to refer to a link *instance*. With the knowledge of link types and link labels, we extend the definition of 'current context' as follows:

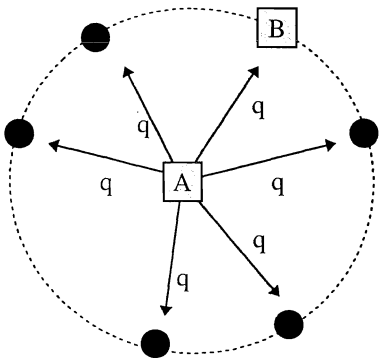
The current context consists of a single node ID, followed by an ordered list of one or more link labels. We call the single node the context node. The link type represented by the first link label should include the type of the context node as one of the node types it is attributed to. Each consecutive link type should have at least one source node type that is a destination node type of the preceding link type.

A tour is generated as follows during a browsing session: first (like described in section 5.3.3) the user selects the current node, say node A, as the new current context.

Current node: **A**

Current context: **A**

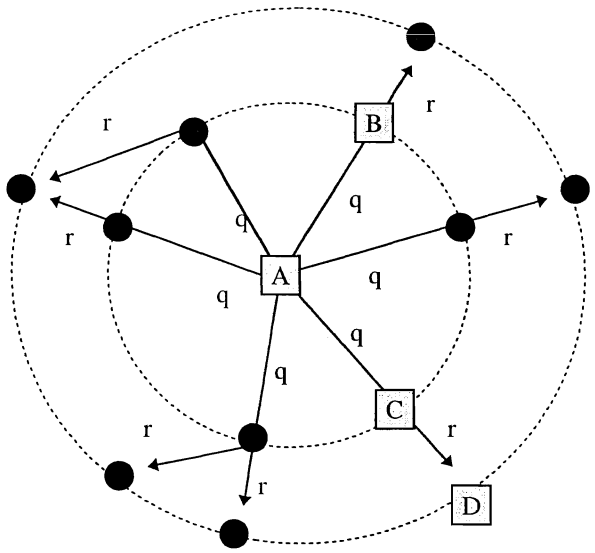
The context only consists of the context node. The user now has the choice between all labels corresponding to link types that are attributed to node A. Selection of label q, generates a tour of all nodes that are linked to node A by links labelled q. So the tour may consist of differently typed *nodes*, but they are all connected to the current node by links of the same *link* type. The first of the resulting set of nodes becomes the current node.



Current node: **B**

Current context: **A ^ q**

After one or more browsing steps (possibly following both direct and indirect links), a node C belonging to the current tour might be reached. From such a node (being part of the current tour), the context can be changed by selecting a link label, say label r, associated with a link type that has been attributed to the current node. The context node remains the same, but the tour now consists of all nodes that were connected by a link with label r to one of the nodes belonging to the previous tour. The resulting tour is a tour of all nodes that are related to the context node by links whose type is the *composition* of all link types involved in the current context, in this case q and r. The graphical result is a circle that is concentric to the previous one.



Current node: **D**

Current context: **A ^ q ^ r**

So each tour is generated by the context and consists of a set of nodes, that is the result of the following operations: first, a list is created of all nodes that are connected by a direct link of the first link type to the context node. Then, a second list is created of all nodes that are linked by a link of the second type to one or more nodes from the first list. This action is continued until the last link type has been processed. The last resulting list contains the nodes that participate in the guided tour. A few examples will clarify this:

'Flowers' ^ *has-as-theme*

The result is a tour of all paintings with 'flowers' as the main theme.

'Louvre' ^ *exhibits* ^ *is-painted-by*

This tour shows all artists that have one or more paintings exhibited in the Louvre museum.

'Paris' ^ *museums* ^ *exhibits*

This is a tour of all paintings that are exhibited in Paris.

'World War II' ^ *

This tour shows everything related to World War II.

Note that node types are not incorporated within context definitions. A user always selects *link types*, unaware of the types of the resulting nodes. This allows for more flexibility during data modelling: when the link type 'owner' is attributed to the node type PAINTING, the destination nodes might be of different types: MUSEUM, PERSON, CITY,... A node is not selected because of its *data type*, but because of the *type of its relation* to the current node. It is the hyperbase developer's duty to construct a hyperbase where the link type definitions are adequate for efficient navigation.

We can now refine the classification of navigational options that was carried out in section 5.3 and where three categories were singled out. *Following a direct link* means either selecting a structural link (instance) or an ad-hoc link. *Following an indirect link* was already aptly described in section 5.3.2. *A context change* can now be initiated either by selecting the current node as the new context node, in which case the whole context changes. Or, a link label can be selected from a node belonging to the current tour, in which case the context changes, but the context node remains the same. The next section describes a rough framework for an application that implements this navigational paradigm and exploits a relational database to store the hyperbase model described in section 6.

8. An application model

The goal of this application model is twofold: first, it should incorporate all features necessary to support the navigational paradigm explained in sections 5 and 7, second, it should allow for easy hyperbase development and maintainability.

The navigational paradigm calls for a hyperbase that is *searchable* for its link structure: to generate the necessary indirect links at runtime, the application needs to be able to query the hyperbase for all nodes directly related to the current context. Thus, all direct links have to be stored within a searchable database. We discern two possible alternatives to accomplish this: the first one is to encapsulate all links within the body of the nodes (like it is the case in most hypermedia environments, like the WWW). However, unlike these other environments, ours should allow all nodes to be searched

for their link information. This calls for an O.O. database where each node is an object and where all links are represented as symbolic pointers to other objects. An O.O. query language allows the nodes to be searched for their direct links, such that indirect links can be generated. Where this option might be very valuable in the future, at present it shows two major drawbacks: the instability of current O.O. database technology and the lack of openness that results from forcing all nodes with their (possibly very distinct data formats) into one proprietary O.O. database model. Such an approach might still fit our present stand-alone application model, but it would be prohibitive towards a future implementation into a distributed environment.

We opted for a second alternative, where the *information content* and *navigation structure* of the nodes are separated and stored distinctly. A (relational) database is used to capture the link structure of the hypermedia system, along with references to the physical addresses of the corresponding nodes. This option leaves much more freedom to implement the *contents* of a node. The only requirement imposed upon a node is that it can be *referred to*. Thus the nodes aspect of the hyperbase can be a very heterogeneous collection, ranging from flat files to objects in an O.O. database, as long as each node is associated with a filename or any other unique ID. Since a node is not specified as a necessarily searchable object, linkage information cannot be embedded within each node. A *linkbase* is used to link all these nodes of different types together and to manage their interrelations. This whole link structure is captured within the semantics of a relational model. Only the tables of the linkbase are to be searched for any link information, not the nodes themselves. The resulting system consists of three aspects: the nodes, a linkbase and a *hyperbase engine*.

8.1. The nodes

We define a node as a collection of data, possibly along with procedural code, that share one common physical and/or symbolic address for references from outside the node. An external object is not allowed to refer to an internal component of a node even though the node itself may take the initiative to present different aspects of its contents, depending upon the link type by which it is accessed. How this is accomplished is left to

the internal design of the node. As already said, the application treats each node as an atomic entity. Nodes are very loosely specified, so that the hyperbase may contain an amalgam of objects of varying complexity. The basic property of a node is that it has to be referable. Nodes can be simple documents like MS Word, HTML or PowerPoint files, as long as the necessary code is provided for on screen presentation. They can be any OLE object, as long as an appropriate viewer is configured. More complex nodes can be real programs to be executed, or objects in an O.O. database.

As already stated, a node may or may not be equipped with intelligence to react differently to different link types. Upon activation, a node is provided with the label of the link by which it was accessed. The procedural code associated with a node may be designed to respond to this label. This approach replaces the anchor concept in the Dexter model. The node does not react to *by whom* it is accessed, but to *the reason why* it is activated.

Where the above dealt with *incoming links*, we now address the possibility to embed references to *outgoing links* within the body of a node. Both concepts are optional, since all necessary facilities for navigation are offered by the hyperbase engine (see section 8.3). However it might be useful to allow for a link or a link label to be selected by clicking a hot spot *within* the visible part of a node and not on a separate panel. It is, again, left to the internal design of a node to provide the application logic to map a keyword, a hot spot, part of a clickable map, a button,... to a link ID or a link label. The hyperbase engine accepts the link ID or label from the node and queries the linkbase for the appropriate link(s). It is a trade-off between maintainability and user friendliness to decide how many references to link ID's and labels will be included within the node itself. A possible strategy could be to include all *link labels*, which are associated with a link type and can be designed on node type level, and to include references to only a few important *link ID's* within each node instance. Note that the destination of each link is unknown to the node, it has only knowledge of its embedded link ID's and link labels. An embedded link ID is only to be adapted when the link itself is destroyed, not when its destination is altered. An embedded link label only needs adjustment when the accompanying link type ceases to exist, not when a link is added or deleted.

8.2. The linkbase

Although additional tables may be useful, the linkbase consists mainly of two important tables: one where each tuple represents a single node and one where each tuple represents a single link. The node table carries a unique *node ID* as a primary key, along with a *node type* attribute, a *description* of the node and a *pointer* to the physical node address. The description is the identifier the user will be confronted with for node listing and selection purposes, in addition it might be used as a criterion to order the nodes taking part in a guided tour alphabetically (other criteria could be used as well). The node ID is unique and location independent: it remains the same during the whole life cycle of the node. The pointer to the physical address is unique also, but it is location dependent and changes when the node is moved to another location.

Note that all nodes, regardless of their node type, are stored within only one table, contrary to conventional relational database models, where there is a table for each entity type. Since strict object typing is of less importance in a hypermedia environment, the loss of modelling richness is more than compensated by the querying advantages: it allows selection of all nodes of any type related to a certain node, something that is impossible in a database with multiple tables. Semantic constraints are not enforced by node typing, but by link typing and by attributing links to node types. Another advantage of this single node table approach is maintenance: if the physical location of a node is altered, only one entry in the database has to be updated. No nodes have to be searched for references to the node that has been moved. No links have to be adjusted.

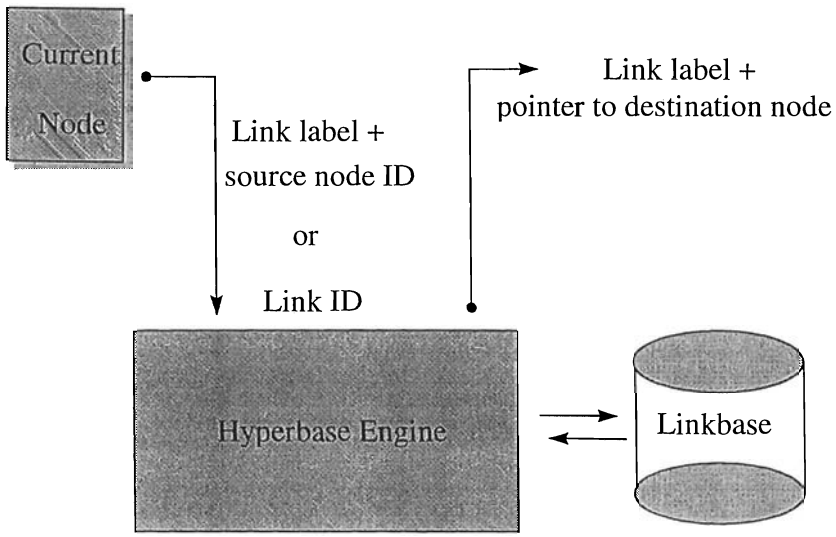
The link table consists of all direct (structural and ad-hoc) links. Information about a link and its inverse are represented in a single tuple. Primary key is the *link ID*, other attributes include the *source* and *destination* node, which both refer to the node ID in the node table and the *link label* (which has a null value for ad-hoc links). It now becomes possible to edit the link structure of the hyperbase without having to access each node involved: only the link table is to be edited. Other advantages include the practice of lazy updating: dangling links can be created with a null value for the reference to their source or destination node. This leads to disaster in systems where links are embedded within the node body, since each node has to be accessed and searched for dangling links. In our

approach, only the link table has to be queried for null values. All kinds of updates can be carried out easily, which is particularly useful in an environment where link destinations are highly volatile.

8.3. The hyperbase engine

This is the software that negotiates between the linkbase and the heterogeneously typed nodes. Its tasks include the following:

- Accept a selected link ID or link label from the current node
- Query the linkbase for the correct link destination
- Map a node ID to its physical location
- Make a call to the selected node
- Keep track of the current context
- Generate the correct indirect links
- Possibly generate a panel for user interaction, as much as the necessary controls are not embedded within the nodes themselves



User input is received by either clicking a control embedded within the nodes or upon a separate panel, generated by the hyperbase engine. There are three possible types of input: selection of a *link ID*, selection of a *link label*, or selection of a *context change*.

8.3.1. Selection of a link ID

Link ID → Pointer to destination node + Link label

One particular link to one specific node is selected. The ID of the selected link is served to the engine. The link table is queried for the node ID of the link destination, using the link ID as search key. Exactly one result is returned. The corresponding physical address is retrieved from the node table and this node becomes the new current node. Also, the link label of the selected link is retrieved. This label can be used by the destination node to adapt its reaction to the type of link by which it is called.

If the selected link pertains to an indirect link, the destination node ID is generated by selecting the correct adjacent node within the current tour. The pointer to the corresponding physical address is looked up in the node table. As link label, the label last added to the current context is used. A direct link of this type will always exist for the new current node, since it was used to include the node within the tour in the first place.

8.3.2. Selection of a link label

Link label + Current node ID → Selected node ID's → Pointer to one node or context change

The link label is passed to the hyperbase engine. The link table is searched for all links that match the combination (Current node ID, Link label). If this combination is unique, the same events as with the selection of a single link instance follow: the node table is used to map the destination node ID to a physical address and the node is accessed.

If multiple links satisfy the query, the user can choose a single node from an index of all destination nodes involved or he can start a new tour by changing the context. If a node from the index is chosen, the system proceeds as above with a pointer to a single node as result. A context change generates a new tour that includes all destinations of links that satisfied the query. This case is described in the next section.

8.3.3. A context change

If the user opts for a context change, there are two possibilities. Either, the current node is selected as the new context node, in which case the whole focus of the guided tour changes. Selection of a link label generates a new guided tour, consisting of all nodes connected to the current node by a link with the correct label. The link table is queried for the combination (Current node, Selected link label).

If a link label is selected to be added to the current context, the focus of the tour remains the same, but the direct relation between the nodes included within the tour and the context node now consists of the composition of the previous relation and the relation associated with the link label selected. The resulting tour consists of link destinations matching the combination (Node belonging to previous tour, Selected link label).

As a conclusion, we suggest that not only a history of 'visited nodes' is kept, like it is the case with conventional hypermedia applications, but also a *context history*. This much shorter list allows the reconstruction of all tours followed during a session, and offers the possibility of returning to a previous focus of attention.

9. Evaluation

9.1. Comparison to CGI-like systems

Our application model is not to be confused with hypermedia applications where one or more nodes retrieve their data from a database (e.g. HTML pages with CGI-access to a database server). In such a case, each node separately retrieves its *information content* from a database, not the *navigational structure*: the links between these nodes are still embedded within the body of the nodes. In our system, it's exactly the *navigational structure* that is stored within a database, which is used to manage the interrelations of all nodes. The information content of a node is considered internal to the node, it may or may not be the result of a database query.

Besides, like real database applications, such systems are bound by the relational database model, where each data type is stored within a different table. So their properties are different, as explained in section 2.1 through 2.8. The data in our system are not bound by the relational model, information about all nodes is stored within one and the same table. The relational model is only used to store *links*, not the data.

9.2. Advantages of the proposed model

9.2.1. Advantages to the end user

The primary goal of the model was to improve the navigation facilities towards the end user. By offering a dynamic linear path throughout the information space, the risk of disorientation is diminished, whereas the task of exhaustively exploring a certain topic becomes much easier.

A second advantage is that the definition of abstractions like node and link types helps the user to grasp the underlying data model, which is described by many as a key condition for easy orientation. The use of templates for nodes of the same type will support this idea even further, by providing a similar layout for similar nodes. Along

with node typing, link typing may also assist the user in the orientation process: not only the *destination* of a link is indicated, but also *why* the source node is related to the destination node, by what kind of relation.

Storing link information within a database offers the additional advantage that a simple query reveals all nodes that are accessible from the current node, which supplies an answer to the question "where can I go from here?".

9.2.2. Advantages for data modelling and application development

The very loose definition of the node concept allows for an open system where documents of almost any type can be used as nodes and be seamlessly integrated into the system, while retaining full navigational flexibility: the hypermedia engine generates a palette containing all necessary controls. The way link types are defined as an interface between node types, allows for different classes of nodes to be developed separately from each other. Furthermore, nodes can be designed without having to worry about destinations of links (lazy updating), dangling links can be completed within the linkbase, without even having to revisit the node implementation. Besides, incomplete links can be very easily detected by a simple query, since they are stored in a database.

Nodes can be designed to react to different types of links, without knowledge of all nodes they are linked to. Only the various link *types* have to be taken into consideration, not every separate link. This replaces the concept of an *anchor* in Dexter, which had to be defined for each individual link. Our approach seems to be more natural: a node does not react to *from which node* it is selected, but to the *reason why* it is selected.

9.2.3. Advantages for application maintenance

Link maintenance can be carried out almost entirely upon the linkbase, without having to alter the internals of the nodes involved. Links can be created or adjusted without accessing the data units, just using the linkbase. Addition of a link of an existing type

doesn't affect other nodes or links. Creation of a new node only affects the node and link tables in the hyperbase. Nodes can be linked to it (with existing link types) without having to be edited.

The practice of attributing link types to node types, rather than just attributing links to individual nodes, allows for checking on consistency. Upon creation of a node instance, the system is able to ask for obligatory links, as it is able to check referential integrity. Deletion of a node that is the destination of an obligatory link, forces the developer to either delete the source node or to reconnect it to another destination node. In order to move a node to another location, the node ID in the node table is selected and the corresponding pointer is adjusted. No links have to be altered at all.

9.3. Future research

Refinement of the data model with the emphasis upon the application developer, rather than the end user will unquestionably be a primary target of future effort. The inclusion of such concepts as super/subtyping for both node and link types and the inheritance of attributed link types is certainly worth considering. Furthermore, to take full advantage of the data model, the specification of a proper development methodology is in order, preferably supported by an accessory development environment.

Although the model described was initially designed for a stand-alone system, future research topics are likely to include the implementation of the concept into a WWW environment. An additional difficulty will be that in such a case, no session information is allowed to be stored at the server side.

References

[1] F. Halasz and M. Schwartz, The Dexter hypertext reference model, *Commun. ACM 37*, 2 (Feb. 1994)

[2] M. Hammer and D. McLeod, Database description with SDM: A Semantic Database Model, *ACM Trans. Database Systems 6*, 3 (Sept. 1981)

[3] H. Maurer and N. Scherbakov, The HM Data Model, *IIG Report, Graz* (1992)

[4] H. Maurer, N. Scherbakov and A. Nedoumov, HM-CARD: A Second Generation Hypermedia Authoring Tool, *LAIR-MIPRO '95 Proceedings*

[5] J. Nanard and M. Nanard, Hypertext Design Environments and the Hypertext Design Process, *Commun. ACM 38*, 8 (Aug. 1995)

[6] P. Srinivasan, Incorporating Intelligent Navigational Techniques to Hypermedia, *LAIR-MIPRO '95 Proceedings*