PUC

Series: Monografias em Ciência da Computação,
No. 21/90

HYPERTEXT DEVELOPMENT USING A MODEL-BASED APPROACH

Daniel Schwabe
Andrea Caloini
Franca Garzotto
Paolo Paolini

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, No. 21/90
Editor: Paulo A. S. Veloso December, 1991

HYPERTEXT DEVELOPMENT USING A MODEL-BASED APPROACH *

Daniel Schwabe
Andrea Caloini **
Franca Garzotto **
Paolo Paolini **

* This work has been partially sponsored by the Brazilian Government Office of Science and Technology.

** Politecnico di Milano, Milano, Italy, where this work has also been published, under series no. TR-90-74.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386   Telex:31078   Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

Abstract:

Hypertext development is still, for the most part, at the "handcrafting" level, where each hypertext document must be hand-designed. We present a compiler which takes hyperdocuments desined using a model-based approach and generates stacks executable in HyperCard. This compiler is implemented in standard SQL over a relational database representation of a hyperdocument designed using the Hypermedia Design Model (HDM). The compiling approach, even though illustrated with HDM, can be used with many "structured" design methodology.

Keywords:

Hypertexts, multimedia systems, authority models for hypertext, compiling hypertext spcifications.

Resumo:

O desenvolvimento de hipertextos está, ainda, num estágio essencialmente "artesanal", no qual cada hiperdocumento deve ser projetado "a mão". Apresentamos um compilador que processa hiperdocumentos projetados utilizando-se de uma abordagem baseada em modelos, gerando "stacks" executáveis em Hypercard. Este compilador é implementado utilizando-se SQL padrão numa representação do hiperdocumento na forma de um banco de dados relacional; este hiperdocumento é projetado utilizando-se do "Hypermedia Design Model" (HDM). Esta abordagem de compilação, embora ilustrada aqui com HDM, pode ser utilizada com qualquer metodologia "estruturada" para projeto de hipertextos.

Keywords:

Hipertextos, sistemas multimídia, modelos de autoria para hipertexto, compilação de especificações de hipertextos.

HYPERTEXT DEVELOPMENT USING A MODEL-BASED APPROACH

Daniel Schwabe* + §

Andrea Caloini*

Franca Garzotto*

Paolo Paolini * +

* Department of Electronics - Politecnico di Milano
Piazza Leonardo da Vinci 32 - 20133 Milano Italy
phone: +39-2-23993634
fax: +39-2-23993587; E-mail: relett34@imipoli.bitnet

+ A.R.G. - Applied Research Group
Via Pio La Torre 14 - Vimodrone (Milano) Italy
phone: +39-2-2650072
fax:+39-2-2650693 ; E-mail: argborll@imiclvx.bitnet
(also schwabe@icil64.cilea.it)

§ On sabbatical leave (until January 1991) from
Departamento de Informática,
Pontifícia Universidade Católica do Rio de Janeiro,
R. M. de S. Vicente, 225
22453 Rio de Janeiro, Brasil.
phone: +55-21-274 4449
fax:+55-21-274 4546; E-mail: pucrjdi@brfapesp.bitnet
Partially supported by CNPq-Brasil.

Abstract:

Hypertext development is still, for the most part, at the "handcrafting" level, where each hypertext document must be hand-designed. We present a compiler which takes hyperdocuments designed using a model-based approach and generates stacks executable in HyperCard. This compiler is implemented in standard SQL over a relational database representation of a hyperdocument designed using the Hypermedia Design Model (HDM). The compiling approach, even though illustrated with HDM, can be used with any "structured" design methodology.

1. Motivations

This article presents a tool which takes hyperdocuments designed using a model-based approach and generates stacks executable in HyperCard [Atkinson 87]. The model we used as input to the compiler is HDM (Hypermedia Design Model) [Garzotto 90a,b], but other models could have been used as well. The main point being made here is that hypertext development using this approach can be speeded up with relatively simple tools.

In this section we want to motivate the need of an application design model for hypertextual applications. Following sections will discuss which kinds of decisions have to be made to get a hypertext running under a commercial environment starting from the model specifications and how etraightforward it is.

The major challenge facing the hypertext author is that of organizing complex material in a suitable way. A systematic approach to hypertext structure is especially important in the design of large hypertexts— documents that are significantly larger than a conventional book.

The use of a model will help to discipline the authoring activity, especially for large and complex hypertexts[1] by encouraging the development of the hypertext in a structured fashion, so that its structure is designed before the actual text is actually filled into nodes.

This is very similar to what happens when developing a strongly modularized application: designing the topology and the interconnections among modules is different than writing the code for the content of the modules themselves. Most hypertext authors agree that hypertext developers face two different (but strongly correlated) tasks: developing a network of nodes and links, and filling in the nodes' content. By analogy with the Software Engineering field we use the terminology *authoring-in-the-large* to refer to the development of the structure of the network and *authoring-in-the-small* to refer to the development of the contents of the nodes.

Authoring in the large can exploit commonalities between applications o a given domain, and it can be at some extent independent from the medium - establishing a connection between two nodes is somewhat independent from the representation of the contents of the node[2]. In addition, it becomes very critical as soon as the size of the hypertext exceed a manageable limit (which is quite arbitrary and dependent, among other things, on the available technology)

Authoring-in-the-small is strongly dependent on the medium (filling in the text in a node is much different than filling in an animation, a sound, or a picture). Moreover, authoring-in-the-small is more dependent on specific application areas.

In a complex hypertext, most of the "deeper" semantics lies in the connections among the nodes, rather than in the contents of the nodes themselves, at least as far as navigation is concerned. As a consequence, many critical design decisions are made at the authoring-in-the-large level.

A model-based approach to authoring-in-the-large provides a predefined vocabulary of concepts and primitives which can be used to specify hypertext applications, with little regard for the contents of the nodes.

[1]In this article we use the term *hypertext* to denote online documents made up of a network of interconnected pieces of information (nodes).We will use the term *hypertext system* for software tools used to create a hypertext. Notecards, KMS, Intermedia, Hypergate, Guide are examples of hypertext systems. Note that a single hypertext might be published in several editions, each using a different hypertext system.

[2]This last statement is perhaps less obvious for span-to-span links, that appear to connect parts of nodes to other parts of nodes.

Most of the existing "models" for hypertext/hypermedia can be considered as "systems models", more appropriate to describe authoring-in-the-small activities. They try to identify representation structures (such as nodes, links, anchors, etc...) and functionalities that are common to most existing hypermedia systems, and to provide a common language in which to describe, compare and evaluate the various hypermedia systems [Halasz 90][Furuta 90]. By analogy, hypermedia design models do the latter for hypermedia *applications*, independently from the implementation environment in which they are developed.

It is interesting to summarize the major advantages in having a design model; some are true of any kind of model, others are mostly true for hypertext design models.

Design models provide a language in which an application analyst can *specify* a given application. Thus they facilitate the communication between the analyst and the end user (i.e., the client, in most cases); between the analyst and system designer; and between the system designer and implementor, when they are different persons. They can be used to *document* the application. This provides support for users of the application; it helps the maintenance of the system; and it serves as a common language in which to compare applications when desired. At the very least, a basis for discussing the similarities of applications exists.

An interesting related aspect, which has received little attention in the research literature, is the task of proof-reading a hypertext. It is clear that proof readers need to check links as well as text, and that spurious or accidental links can be as embarrassing (or even damaging) as more conventional typographic errors. This task should be greatly helped by the availability of a specification language.

As a matter of fact, the availability of such a language paves the way for (partial) reuse the back-bone structure of applications, since these models capture the "essential semantics" of applications, and can therefore be reused when the semantics of the two applications are similar enough.

Design models provide a framework in which the authors of hypermedia applications can develop, analyse and compare *design methodologies* and *rhetorical styles* of "hyperauthoring", at a high level of abstraction. This analysis can be done without having to resort to looking at particular visualizations (screen formats and appearances, button functionalities and the such) or to the detailed contents of units of information. At this level, it is possible to analyse the "conceptual" organization of the application domain knowledge represented, and examine its adequacy for the intended uses.

Design models can be used by *Design Tools*, much in the same way as application generators are based on languages to specify classes of applications, or as CASE tools have specification languages to describe software (at various levels of abstraction). We will show later how our design model can be used to derive a HyperCard [Atkinson 87] application.

An additional expectation is that applications developed according to a model will result in a very predictable representation structure. As a consequence, navigation environment for possible readers should also be predictable, thereby reducing the so-called "disorientation/cognitive overhead" problem [Utting 90].

To be able to provide the advantages just described, a model clearly must be *expressive*, allowing the description of concepts at the appropriate level; it must be *complete*, in the sense that it captures the most frequently occurring structures; and it must be *simple*, so that it can be easily used by authors.

The remainder of this article is organized as follows. Section 2 describes a design model that addresses the issues discussed so far; section 3 describes a compiler for this design model; and section 4 draws some conclusions.

2. The Hypermedia Design Model (HDM)[3]

HDM is an hypertext application model developed at the Politecnico di Milano. According to HDM, an application domain is seen as being composed of Entities, which in turn are formed out of hierarchies of Components. Entities belong to a Type. Entities can be connected to other Entities or Components by Links which can be either Structural or Application links. Structural links reflect the hierarchical structure of entities; Application links connect Entities or Component to other Entities or Components to reflect application domain relations. Components can be instantiated by one or more Perspectives into Units.

We discuss each of these primitives in more detail in the next sub-sections.

2.1 HDM Primitives

2.1.1 Entities and Components

We refer to an *Entity* as a concrete or conceptual real world object in the domain that is relevant for the application. Typically, an entity will denote something quite complex, whose internal structure may be further decomposed. An entity in its whole can be represented through several individual, smaller-grained pieces of information, that we call *Components*. Examples of entities are "Law 19/8/89", Dante's "Divine Comedy", Gershwins's "An American in Paris".

A Component is a piece of information describing a part of an Entity. Components are grouped into an arbitrary, application dependent, *hierarchy* to form the corresponding entity. All components of an entity are homogeneous.

Examples of possible components (with the corresponding entity from the examples above) are "Article 1" ("Law 19/8/89"), "Paradise" ("Divine Comedy"), "First Movement" ("An American in Paris").

We have chosen hierarchies as the structure of Entities because they are a frequently occurring structure, useful in specific contexts [Brown 87]. Many authors have observed that hierarchies are very useful to help user orientation when navigating in hyperdocuments [Acksyn 87, Brown 89]. HDM recognizes this via the notion of entities made up of components organized into hierarchies, but leaves unspecified the criterion to be used when breaking up an Entity into Components since it is very much dependent on the specific application.

[3] This section contains a summary of HDM, which is fully described in [Garzotto 90c].

2.1.3 Perspectives and Units

Components describe pieces of information. Due to the richness of most hypertext reading environments, information may be presented in many different ways. A natural abstraction that can be made in these situations is to allow an author to refer (and think) about the concept that is being represented by a Component independently from the way it is described. In other words, the concept can be thought of as having several "perspectives".

HDM facilitates this abstraction by having *Perspectives* for Components. By the term Perspective we mean the appearance of a piece of information. The description of a component according to a given perspective is called a *Unit*, which has a body (the information itself). HDM says nothing about bodies, as it is interested in talking about hypertext structure.

A Component may have, therefore, one or more Units (corresponding to Perspectives). For example, we may say that each component of entity "An American in Paris" has a "Textual" and a "Musical" perspective, corresponding respectively to the musical score and a digital recording of the symphony.

2.1.4 Entity Types

A common abstraction found in most data models is the notion of type. An *Entity Type* groups entities having common properties. In HDM, the properties chosen as relevant are "using the same set of perspectives", "being broken into components according to the same criteria", and "being related to other entity types in the same way".

Examples of entity types (with the corresponding entity from previous examples) are "Law" ("Law 19/8/89"), "Poem"("Divine Comedy"), "Symphony" ("An American in Paris").

In HDM the set of perspectives associated to an entity type is indicative only of *possible* perspectives for its entities, so that it is necessary to have the notion of a *Default Perspective*. The intended meaning of the default perspective is that *all* entities of this type have *at least* this perspective; further significance of this concept will be shown when we discuss the process of mapping into node-and-link structures.

2.1.5 Links and Link Types

The major advantage of the hypertext model is that one may organize an information base in a non-linear fashion. This means, loosely speaking, that pieces of information can be related to each other through links associated to them. The success or failure of a given application using the hypertext paradigm is heavily dependent, among other things, on the appropriate choice of links. The more the "meaning" of a link approximates the relationships in the application domain, the more the user will be at ease in using the corresponding hyperdocument, since links will evoke familiar associations.

HDM differentiates between three kinds of links, which we discuss next. HDM Structural Links connect components belonging to the same hierarchy. Since entities are intended to provide a kind of "navigation context", following hierarchical links has familiar meanings such as "Next", "Previous", "Up" (e.g., to move higher in abstraction level), "Down" (e.g., to get more "details"), etc... The meaning of each of these relations is dependent on the particular criterium chosen to organize the hierarchy, but the hierarchical relation should give the reader the "feeling" that she is moving "inside" the particular entity in question.

The second class of links in HDM are the Application Links. Whereas structural links capture rather "standard" semantics (structure), Application Links embody semantic relationships in the domain. That is, they represent some (arbitrary) relationship between entities that the author deems meaningful, in the sense that this relationship evokes some association between concepts useful to the user of the hyperdocument. By providing an application link, the author will be making it "natural" to the user to access some information which is "related" to the information being read at that point, by simply traversing that link.

The third type of link in HDM are the Perspective Links. Given that Components stand for an abstraction of several Perspectives of the same subject, Perspective links allow the reader to move between different Perspectives (i.e, Units) of the same Component.

To be consistent with the notions of Entity and Entity Types, HDM also defines Application Link Types as being a set of links instances whose source and destination entities are of the same entity type, respectively.

For example, the author may specify that a link of type "Is-author-of" can connect entities of type "Book" to entities of type "Person": this means that in principle all instances of "Book" (e.g., "Hamlet", "Illiad", etc...) may have a link to a corresponding "Person" (e.g., "Shakespeare", "Homer") that is its author.

Up to now we have been describing links at a conceptual level, i.e., links between entities, links between components, and links between entity and components. How these links are actually perceived by the user will be discussed together with the discussion on browsing semantics, in section 2.3.

2.1.6 Outlines

An important part of many applications is providing the initial access to the hypertext, before the user starts navigating at all. More generally, it is useful to envisage navigation patterns (including the starting points in the hypertext) which are superimposed onto the hypertext itself.

HDM recognizes this need by allowing (and encouraging) the usage of Entities in a special way, which we call Outline. An Outline is a special usage of types of entity (and therefore Outline instances have hierarchical internal structure) whose instances have leaf components which "point to" (are linked by application links to) nodes of the hypertext proper; the contents of outline components is navigational information. A typical example of an outline is a hierarchical index of the contents of a hypertext.

2.1.7 Derivation of Links

Having hierarchies as primitive concepts suggests that, at design level, an author needs to specify only a minimal set of structural links - those necessary to define the structure of an entity ("father" and "next brother ") - from which a large number of other implicit structural links (such as father-son, to-top, etc...) may be derived if desired.

Derivation rules can exist for application links too if we regard them as relations between entities (components). Properties of these relations, such as symmetry and transitivity, when existing, can be used to derive other links. The reader should note that link derivation becomes particularly powerful when used in conjunction with composition of relations.

As an example (see Fig. 1), assume that "Article 2" of a "Bank Regulation" (a component of an entity of type "Regulation") is connected to "Section 1" of a "Contract" (a component of an entity of type "Legal Document") by a link of type "Has Effects on". Then one might argue that that article of the bank regulation "Has Effects on" the whole contract too. To represent this, the author would probably set a link of type "Has Effects on" between the "Bank Regulation" and the root of "Section 1" ("Contract"). This link however can be derived straightforwardly as simple composition of the link "Has Effects on" between the two components and the structural link "to-top" between the second component and the root of its entity.

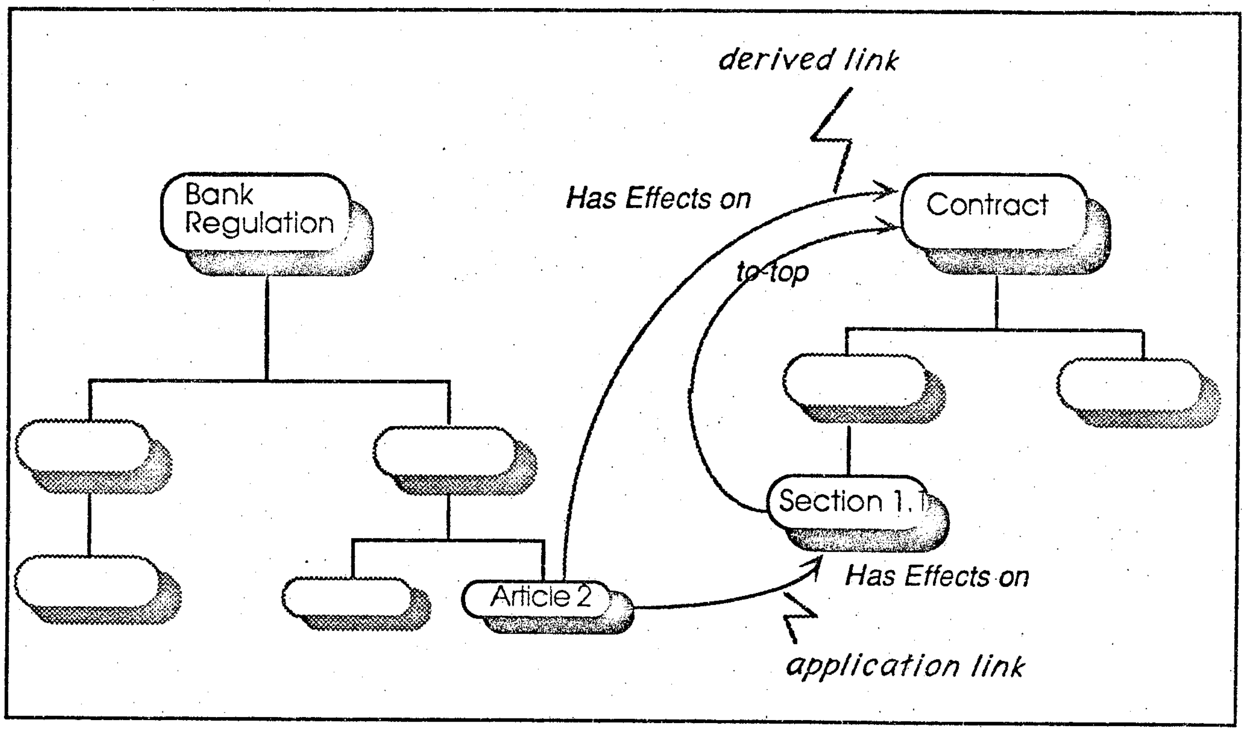


Figure 1- Example of derived link

It should be observed that defining which links are actually derived, for a given application, is a full fledged design step. The same schema can be reused by simply specifying different derivation rules; this usually will take into account delivering the same application to different kinds of users.

An important kind of derived structural link is the *Presentation Link,* which connects units of the *same* component. Presentation links allow the user to look at the same object switching between perspectives. For example, switching to the "Text" perspective of a procedure when looking at its "Graphic" perspective.

So far, HDM does not itself include a language to specify such derivation rules. Nevertheless, it is possible to have such a language in a formalism outside HDM, as will be exemplified later. Given such a language, we will see that the use of HDM can provide a great amount of conciseness to the model specification - the author needs to provide a much smaller amount of links than the number of links that will actually be present both at conceptual level and at concrete level.

2.3 Browsing Semantics

The actual appearance of a hypertext is largely defined by its *browsing semantics* [Stotts 89]. In HDM browsing semantics will determine three further *design* choices for authoring-in-the-large:

1) What are the objects for "human consumption"; in HDM terms, what can the user perceive: Entities, Components, or Units?

2) What are the perceived links between objects; in HDM terms, which links are visible?

3) What is the behaviour when links are activated; in HDM terms, what happens when one activates a one-to-many link?

HDM, as discussed so far, does not specify any particular browsing semantics for hypertexts specified with it. Work on providing primitives for defining such browsing semantics is at a preliminary stage, but simple browsing semantics can be specified with formalisms such as Petri-Nets (see [Stotts 89]); [Garzotto 90c] contains the formal definition of the browsing semantics described in section 2.3.1 using a Petri-Net based formalism.

Note also that other aspects of the browsing activity, such as the actual appearance of screens, icons appearance and other authoring-in-the-small concerns, are not discussed at all.

However, when actually compiling an HDM specification, on must choose a particular browsing semantics, which must be compatible with the target hypertext system. An important point to stress regarding the translation process is that it actually introduces another design dimension. Each choice made when deciding how to translate HDM links into concrete links affects the final hypertext the user will see. For this reason, it is quite natural to think that these choices should be made according to certain user profiles, therefore allowing the same hypertext design to be compiled for different classes of users.

In the next section, we exemplify this discussion for a class of simple browsing semantics, that one of plain *node-and-links* found in many hypertext systems, such as HyperCard.

2.3.1 A simple browsing semantics compatible with HyperCard.

The class of browsing semantics discussed in this section will serve as a basis for the compiler described in more detail in section 3.

In this class of browsing semantics, we assume that no abstract objects (entities and components) are directly visible - only units (corresponding to the usual hypertext notion of node) can be perceived by the readers as concrete objects. In consequence, readers can see links only among units and so, in the end, actual connections must be established among units. Furthermore, we also assume that only one node is active at any time, and only one link can be traversed at any time from an active node. We will call *concrete links* the connections among units, as distinguished from *abstract links*, which are defined among entities and/or components. It is necessary, therefore, to specify how concrete links can be obtained from abstract links.

A rule for component-to-component *structural* link translation is the following: if a component C1 has a structural link to a component C2, then, for each perspective P, each unit of C1 having this perspective should be linked to the unit of C2 having the *same* perspective. This rule expresses a kind of stability criterion w.r.t. the use of perspectives - if the reader is looking, say, at the "Text" perspective of an article of a law, and follows the structural link "next article", she will see its text perspective (as opposed, say, to the "Graphic" perspective, which might be the default.).

A simple choice to map abstract application links is the idea of having a *default representative* for each abstract object (component or entity). The default representative for a component is its unit in the default perspective of its type. The default representative for an entity is the default representative of its root component. This corresponds to saying that the root component of an entity (in its default perspective) "stands" for that entity. Given this notion, entity-to-entity abstract links translate into concrete links between their representatives.

A simple rule for translating component-to-component application links is one in which each abstract link corresponds to a *set* of concrete links connecting each unit of the source component to the default perspective unit of the target component.

To illustrate this rule, consider the situation (fig.2) in which there is a "Part of Law" component (belonging to an entity of type "Law'), linked to a "Step of a Procedure" component (belonging to an entity of type "Procedure"), through an "Has Effects on" application link. Consider further that the "Law" entity type has perspective types "Text" (corresponding to an informal commentary) and "Official Text", and entity type "Procedure" has "Text" and "Graphic" perspectives (the latter being a dataflow diagram for example), the "Text" one being the default for both entity types.

In this case, concrete links (corresponding to the link connecting the two components at the abstract level) will connect both the "Part of a Law:Text" and the "Part of a Law:Official Text" units to the "Step of a Procedure:Text" unit (corresponding to the default perspective).

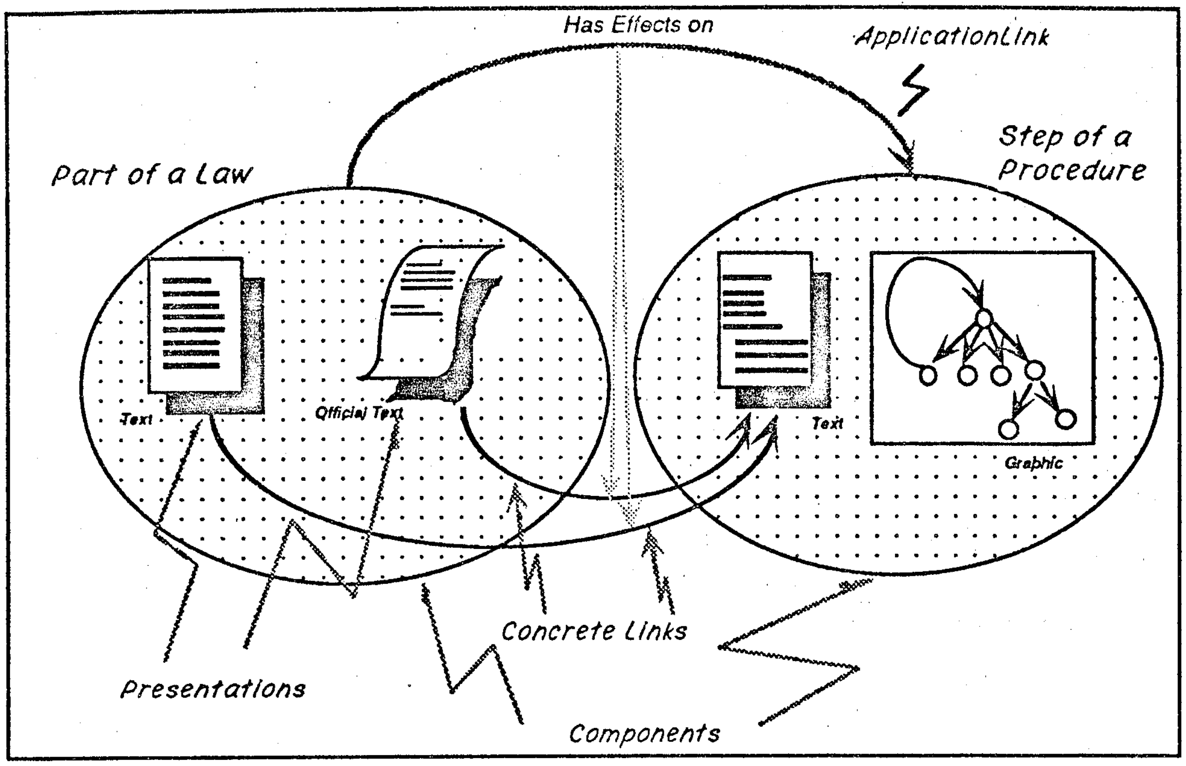


Figure. 2 - Example of concrete link generation

It should be stressed that this is only *one* of the possible choices that can be made, and HDM is not strictly prescriptive in this respect.

In hypertexts, connections among pieces of information are usually visualized through "anchors" (or "buttons"). Anchors are well identifiable areas on the screen that show the existence of a connection, and can be selected by the reader in order to get into the link target(s).

This approach suggests the need of a new primitive, which captures the notion of "anchor type". An anchor type specifies the properties of anchors that represent links of the same type - or of a group of link types. It might be often the case that what the reader actually "sees" in terms of anchors is a restricted set - the author may have chosen to group links of some types under anchors of a single anchor type.

Consider for example, that entity "Procedure" have a link of type "Formal Legal Justification" to a "Law" and a link of type "Informal Justification" to a an "Informal Regulation". The author might want to provide to the user a single reading link, maybe labeled "Justification", since the distinction might not be important for the reader. This can be achieved by specifying the anchor type "Justification" to be the union of link types "Formal Legal Justification" and "Informal Justification". It should be noted that the anchor mechanism also allows the author to selectively hide some links for some perspectives.

Anchor types, therefore, provide a mechanism for the author to present groups of link types together, also renaming or hiding them as well, as desired. By assigning a set of link types to each anchor type, the author can control link visibility. In this way, the actual topology seen by the author may be much more complex than the one presented to the reader.

From the previous discussion, it is clear that there are many situations in which an anchor actually refers to several possible destination nodes. It must be defined, therefore, what happens when the user activates one such anchor. Clearly, this is part of the particular browsing semantics being used, which in turn is partially dependent on the particular system being used to implement the hypertext - if it supports multiple active windows, for instance, a possible choice may simply be to show all destinations, each in a separate window. This solution, however, is often not acceptable, and oftentimes not even possible in many systems.

One possible alternative approach that can be adopted to deal with this situation is to introduce the notion of *chooser*. A chooser is a mechanism associated with a single-source-multiple-target anchor that allows the selection of one of the multiple targets of the anchor. As such, choosers are implementation structures that can be generated automatically from the specification of the hypertext, using any available mechanisms present in the implementation environment (e.g., menus).

3. Compiling HDM

3.1 The Structure of the Compilation Process

Given an HDM specification of a hypertext, we would like to translate it into an actual runnable system, preferably using some already commercially available delivery platform.

A compilation scheme for HDM is given in Fig. 3. The first step, which strictly speaking is not really compilation (since it does not really change abstraction levels), implements the "link derivation engine", i.e., explicates the derived links specified in the schema instance. The second step implements most of the browsing semantics, generating an Intermediate Model (IM) at the node-and-link level of abstraction. Finally, step three takes the IM and generates a running system, using a commercially available system, HyperCard.

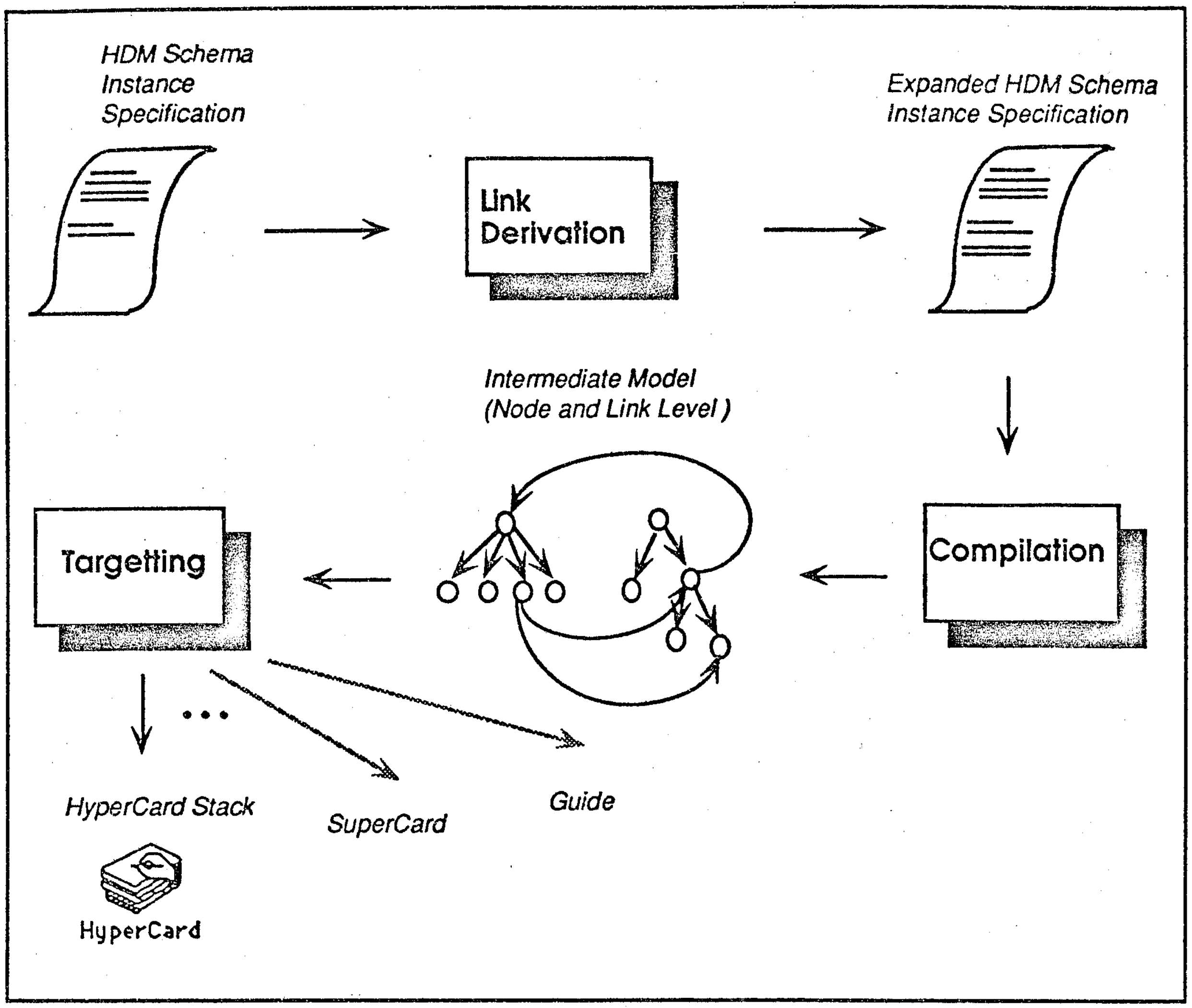

HDM Schema Instance Specification
Link Derivation
Expanded HDM Schema Instance Specification
Intermediate Model (Node and Link Level)
Targetting
Compilation
HyperCard Stack
SuperCard
Guide
HyperCard

Figure 3. Schema of HDM Compilation Process.

Step 1) Link Derivation

This step is responsible for explicating all links that were specified intensionally in the schema. As said previously, this is not strictly a compilation step, in the sense that its output is still an HDM schema instance (which we call the *Expanded HDM Schema*) at the same level of abstraction. The only difference is that all links, structural and application, are now explicitly included.

Once again, it should be remembered that this derivation may be done taking into account specific classes of users and tasks. For example, it may be decided that next-brother structural links are not interesting for this class of users; similarly, no propagation of application links to the root component (as in example in fig. 1) may be desirable.

Step 2) Translation into Node-and-Link Level

Here the Expanded HDM Schema is actually compiled into a lower level representation (which we call the *Intermediate Model* - IM). This step is the most dependent on the particular browsing semantics chosen.

Typically, the level of the IM is at the node-and-links level, the same as that of the Storage Layer of the Dexter Reference Model [Halasz 90] (indeed it can be shown that IM is a subset of the Dexter Model). The main advantage of this choice for the IM is that one can benefit from the fact of the IM being indendent of any particular hypertext system, and thus being able to generate different versions of the same hypertext, using diverse delivery environments.

Step 3) Translation into the Delivery Environment

An IM instance is translated into a specific delivery environment. Here some further design decisions are made, e.g., what is the mechanism used to implement choosers, how are anchors visualized, etc...

An important aspect to highlight is the compatibility between the particular browsing semantics chosen and the delivery environment. For example, it is of little use to have a browsing semantics in which multiple windows are allowed, and have HyperCard as the delivery environment. Conversely, it may still be useful to have a browsing semantics allowing only one window, and have Intermedia or SuperCard (which allow multiple windows) as delivery environments.

3.2. The HF Compiler

In this section we present HF, a compiler for HDM, developed at the Politecnico di Milano, based on a relational dabatase representation of the schema instance, following the compilation schema of the previous sub-section. This means that this implementation takes an HDM schema instance cast as a relational database (which we call the HF Database), applies some derivations rules specified as relational database operations, compiles the schema instance into a node-and-links level description (IM description) and finally takes the IM and translates it into HyperCard. The compilation we will illustrate refers to the simple browsing semantics discussed in section 2.3.

It should be observed that the database described here is *not* intended to support *development* in HDM, but rather to support the compilation process; in other words, the output of the development environment could be the HF database. There are several possible approaches to supporting the complete development cycle, and the HYTEA Esprit II project [ARG 90] is focusing on providing an integrated environment in which this could be done.

3.2.1. The HF Data Base

This subsection shows in detail the relationships between the objects of HDM and the schema of the relational database that models them; appendix I contains the SQL code for some of the steps of the compiler. Below we describe each table of the database and its attributes. An attribute marked with an "*" indicates the primary key to the table; names starting with "#" are identifiers of objects.

```
ENTITY (#Entity*                  unique identifier
        Entity_type
        Default_perspective
       );
```

Note that here we have incorporated entity type information into the entity description, instead of having a separate table to describe entity types.

```
COMPONENT ( #Component*      unique identifier
            #Father          null for root components
            #Entity          null for non root components
            Title            short description of this Component
            Order_number .   ordering specification
          );
```

`Order_number` represents the *next* relation necessary to define an entity instance.

```
UNIT ( #Unit*              unique identifier
       #Component          the Comp.this Unit belongs to
       Perspective         "style" of this Unit
       Body                pointer to the Unit's body
       Template            Information for the delivery environment
     );
```

The `#Component` attribute specifies the component the Unit belongs to and `Perspective` specifies which Perspective of `#Component` the unit implements. Since the (set of) Perspectives are associated to Entities types, all the components of a given entity will have the same number of Units (one for each perspective associated to the entity's type). The `Body` field contains a "pointer" (e.g. a filename) to the unformatted information to be held by the unit. The body of the unit is stored in the Delivery Environment, not in the HF database; `Body` ties the HF version and the Delivery version. The `Template` attribute is not interpreted here, and is meant to describe how the node should be processed in the Delivery Environment; it may have values such as "HyperCard template card", "a chooser template", and so on.

```
APLINK ( #Source        Source component Id
         #Target        Target component Id
         link_type      describes link semantic
       );
```

The `AP`plicative `LINK`s table represent explicit connections between pairs of components. Note that the database does not explicitly describe HDM structural links, since they can be computed; the same is true for derived application links.

```
LINKAN ( link_type*     This link_type must be mapped
         anchor_type    into this anchor_type
       );
```

The `LINK` and `AN`chors table groups many link types under the same anchor type; a given `link_type` can be associated to one `Anchor_type` only.

```
ANPERSP ( Anchor_type
          Perspective
        );
```

The table `AN`chors and `PERSP`ectives states which perspectives should have which anchors, since some anchor types make sense only for certain perspectives and not for others.

3.2.2. An application example

As an example[4], suppose we are designing a hypertext that contains the descriptions of loan procedures in a bank. Besides describing the procedures themselves (which steps to follow, what forms to compile, etc...), the hypertext also represents all the relevant regulations that determine why certain forms look the way they do, and why certain steps must be taken instead of others.

The bank manipulates *Documents* according to *Procedures*. The reasons why *Documents* and *Procedures* are the way they are can be explained by looking at *Laws, Regulations* and *Informal Norms.*

Laws are issued by the state to discipline and control credit granting and taking activity. Most of the times laws are too broad, and must be made more specific by *Regulations* issued by some authority, on the basis of the text of the law. Finally, these regulations are interpreted within an organization with the addition of *Informal Norms,* which are of course valid only for that organization.

The above state of affairs is captured in the schema described in Fig. 4., where Entity Types and Application Link Types have been specified. Since in this case all application links are by-directional (representing a relation and its inverse) - this is frequently (but not necessarily) the case - we have drawn the links only once.

[4]This example is a subset of a larger prototype application (named Expert Dictionary) [Garzotto 89] developed by ARG SpA and Politecnico di Milano within the european Esprit projects INDOC and SUPERDOC.

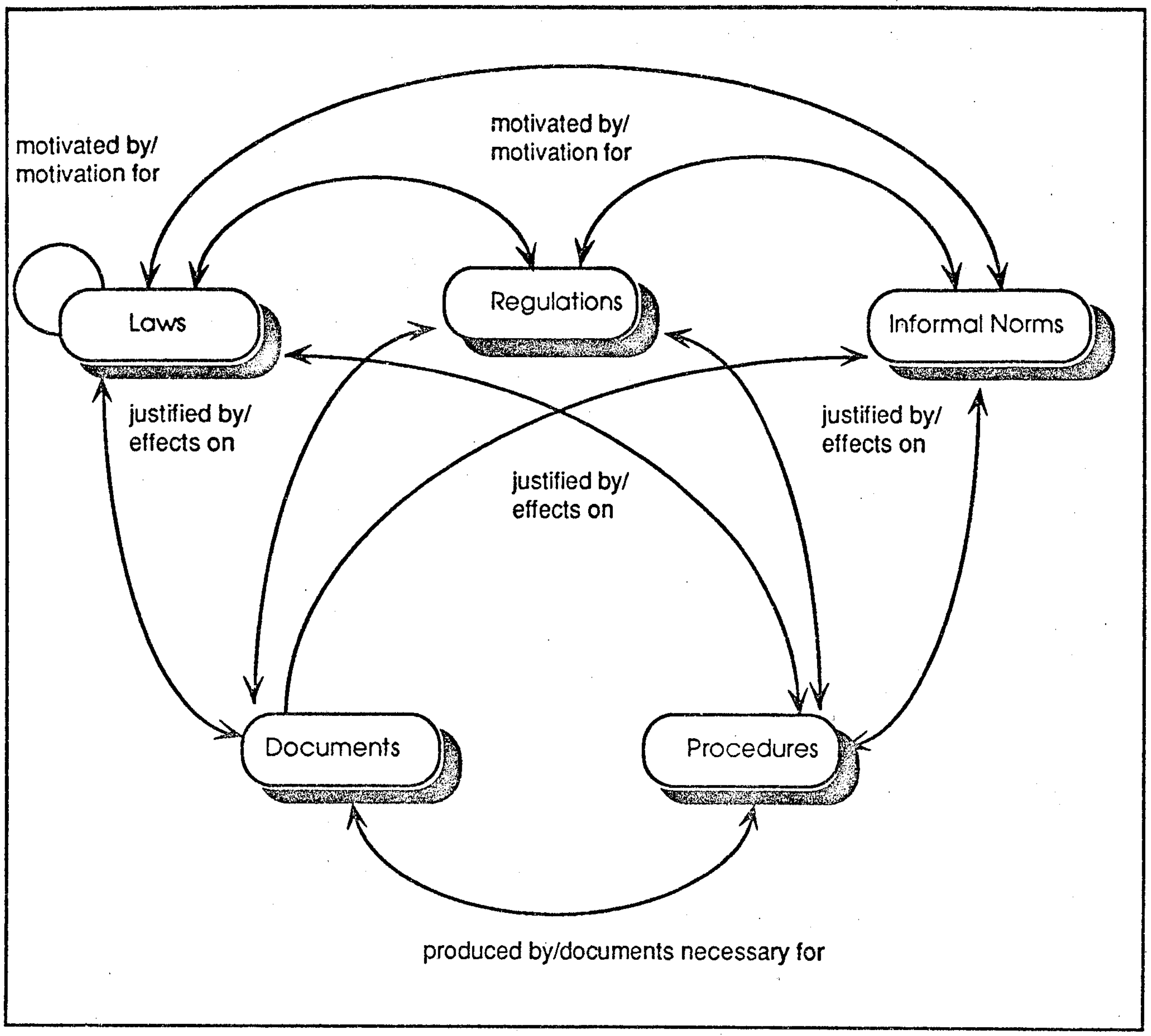


Figure 4. Schema of Entity Types and Application Link Types

Each of these Entity Types has a set of *Perspectives* associated to it. We do not enumerate all of them here, but simply mention that the default perspective for "Regulations" is "Official Text" and for Procedures is "Description".

Let us look now at a small part of an instance of the schema above, depicted in Figure 5. In the figure, shaded areas denote entity instances whose internal structure has been further detailed. We have simplified links and perspectives, keeping only what is essential to illustrate the compilation process. Note that the boxes representing the components of entity "Mortgage Loan Procedure" have a dark gray shadow, to indicate they have two perspectives.

There is an entity of type "Procedure" named "Mortgage Loan Procedure", which is made up of several steps. One of the intermediate steps is named "Request Acceptance and Entry". This step is, in its turn, also made of several steps, the third being named "Request Data Entry". As we can see, the "Mortgage Loan Procedure" is really a hierarchy of sub-procedures, each represented by a component.

The entity "Circular HyperBank 21/10/89" represents some internal bank norm; some of its parts are shown as nodes attached to the root. Between the components of these two entities there are two application links, "Has Effects on" and "Motivated by"; the structural links inside each of the entities are represented in gray.

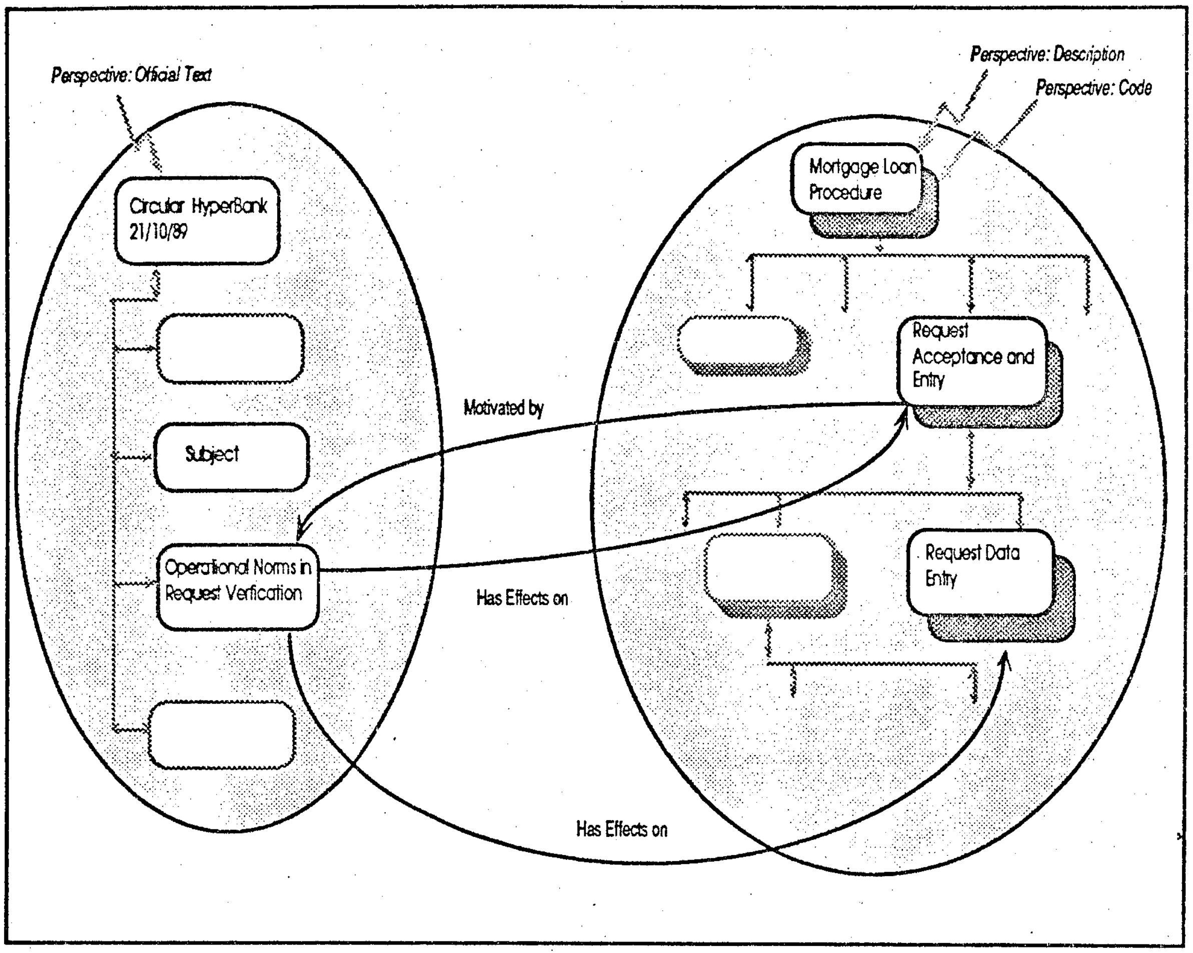


Figure 5 - An example hypertext fragment. Grayed nodes are shown to give an idea of an overall meaning of the components actually included in the example.

3.2.3 An example of HF Data Base

The HF database instance given below represents the hypertext fragment of figure 5.

ENTITY

#Entity	Entity_type	Default_perspective
Circular_HyperBank	Regulation	Official_Text
Mortgage Loan Procedure	Procedure	Description

COMPONENT

#component	#Father	#Entity	Title	Order_number
C1	nil	Circular HyperBank	Circular HyperBank 21/10/89	1
C2	C1	nil	Subject	1
C3	C1	nil	Operational Norms in Request Verif	2

C4	nil	Mortgage Loan Procedure	Mortgage Loan Procedure	1
C5	C4	nil	Request Acceptance and Entry	1
C6	C5	nil	Request Data Entry	1

UNIT

#Unit	#component	Perspective	Body	Template
N1	C1	Official_Text	file1	CHTempl
N2	C2	Official_Text	file2	CHTempl
N3	C3	Official_Text	file3	CHTempl
N4	C4	Description	file4	MLTempl-1
N5	C5	Description	file5	MLTempl-1
N6	C6	Description	file6	MLTempl-1
N7	C4	Code	file7	MLTempl-2
N8	C5	Code	file8	MLTempl-2
N9	C6	Code	file9	MLTempl-2

APLINK

#Source	#Target	link_Type
C2	C5	Has_Effects_on
C2	C6	Has_Effects_on
C5	C3	Motivated_by

LINKAN

link_Type	Anchor_Type
Has_Effects_on	Effects
Motivated_by	Motivation
root/Motivated_by	Global Motivation

ANPERSP

Anchor_Type	Perspective
Effects	Official_Text
Motivation	Description
Global Motivation	Description

3.3. The Compilation Process

3.3.1 - Derivation of links

Even though HDM does not yet include a language for specifying derivation of links, we illustrate here how this can be done using a relational query language to do so. It should be clear that we do not, by this, advocate that this is the most suitable language to do so; in fact, we are convinced that there are better (and more powerful) ones. What is described here is simply an exercise to show the viability of the ideas discussed so far.

3.3.1.1 - Structural links derivation

This stage makes explicit structural links left implicit in the HDM hypertext definition. Five structural link types will be made explicit:

FATHER: links a component to its direct ascendent

SON: links a component to its first descendant

RIGHT BROTHER, LEFT BROTHER: link in a linear fashion all the sons of the same father component

ROOT: links a component to the root component of its entity

These new links have the same structure as application links: they are described by a source component, a target component and a link_type. The output of this stage is a hypertext where entity trees of components are not any more a data structure embedded in the hypertext, but are implemented explicitly by links.

The database describing the hypertext is the same as the input database, where table APLINK has been modified as follows. (bold show new or modified records).

APLINK (including structural links)

#Source	#Target	link Type
C2	C5	Has_Effects_on
C2	C6	Has_Effects_on
C5	C3	Motivated_by
C1	**C2**	**son**
C2	**C1**	**father**
C3	**C1**	**father**
C2	**C1**	**root**
C3	**C1**	**root**
C2	**C3**	**rBrother**
C3	**C2**	**lBrother**
C4	**C5**	**son**
C5	**C6**	**son**
C6	**C5**	**father**
C5	**C4**	**father**
C5	**C4**	**root**
C6	**C4**	**root**

3.3.1.2 - Application links derivation

Next, we compute the derived application links; they can be computed on the basis of both structural and application links. A rule for generating derived links we will use as an example is:

- for each application link connecting two components of different entities generate a derived link to connect the source component to the root of the target component.

This derivation rule reflects the assumption that if a component of an entity is linked for any reason to a component of another entity, then it should be linked to its root (which stands for the whole entity) too.

After this stage the original hypertext has been transformed into an equivalent one where all links that were left unexpressed in the original HDM description have been explicated. In the sequel we will refer to "component links", disregarding whether they are structural, application or derived. The database describing the hypertext is the same as for the previous section except for table `APLINK` which has the following two new entries added

#Source	#Target	link_Type
C2	C4	root/Has_Effects_on
C5	C1	root/Motivated_by

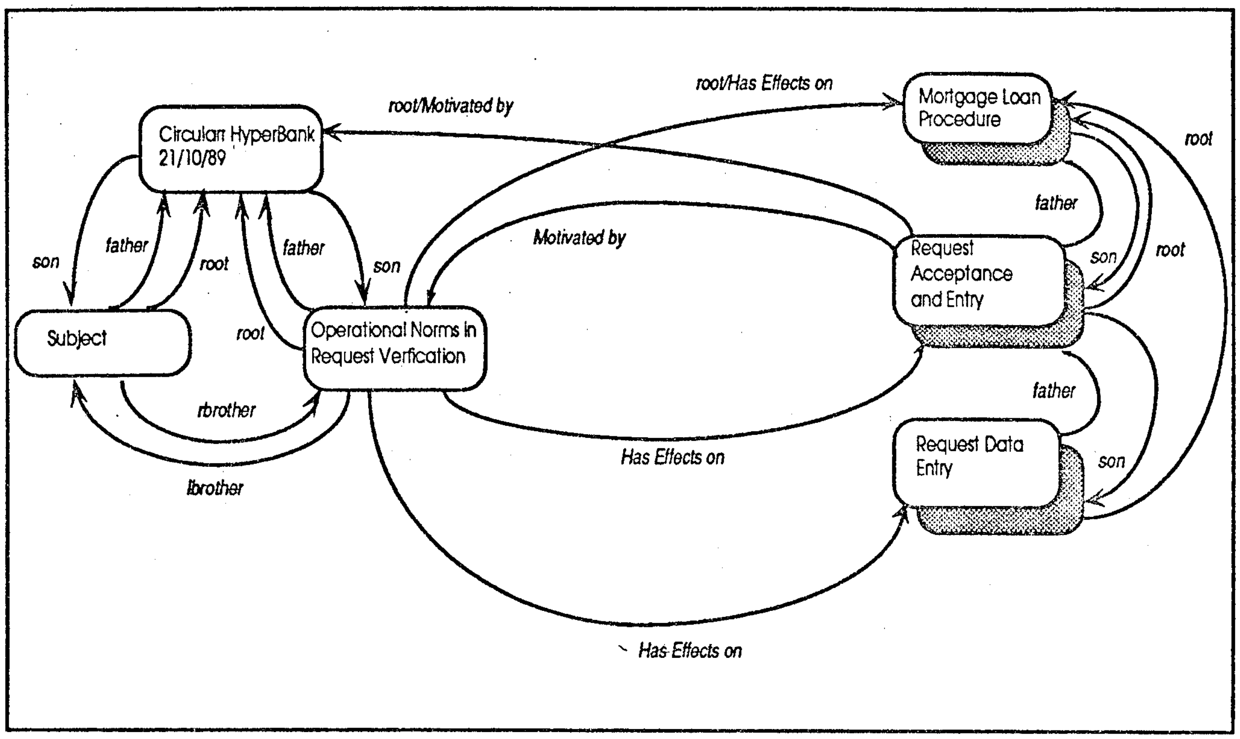


Figure 6 - Derived links for example in fig. 5

Fig.6 represents the result of a link derivation applied to the HDM hypertext fragment of fig. 5. Entities are not shown for the sake of simplicity.

3.3.2 - Mapping entities and components into units

Next, we map the concepts of component and entities into units. It should be remembered that we are using the browsing semantics in which Units are the only visible HDM objects; they hold the bulk information actually viewed by the user of the hypertext and "physically" correspond to a "node" in the common hypertext terminology. Links between components will be mapped to their associated units accordingly to the following rules:

1) for each pair of linked components belonging to the same entity place a link between each pair of their units having the same Perspective;

2) for each pair of linked components belonging to different entities place a link between each unit of the source component and the default perspective unit of the target component;

3) for each unit of a given component place a link of type Change_perspective to cycle between all the perspectives of that component.

4) The type of a new unit link (except for the third rule) shall be the same as the type of the component link it originated from.

The structure of unit links is isomorphic to that of component links, being defined by a source unit, a target unit and a link_type. Fig. 7 shows the output of this step. It is composed of structures very similar to those of fig. 6, where components Circular HyperBank 21/10/89, Operational Norms in Request Verif,and Subject and their structural links have been transformed into an isomorphic structure composed of nodes N1, N2, N3 together with their structural links. Components Mortgage Loan Procedure, Request Acceptance and Entry, and Request Data Entry have been transformed into two isomorphic structures (one for each perspective), composed by nodes N4, N5, N6 and N7, N8, N9. Isomorphic nodes (for example pair <N4, N7> of these structures are linked by Change_perspective unit links.

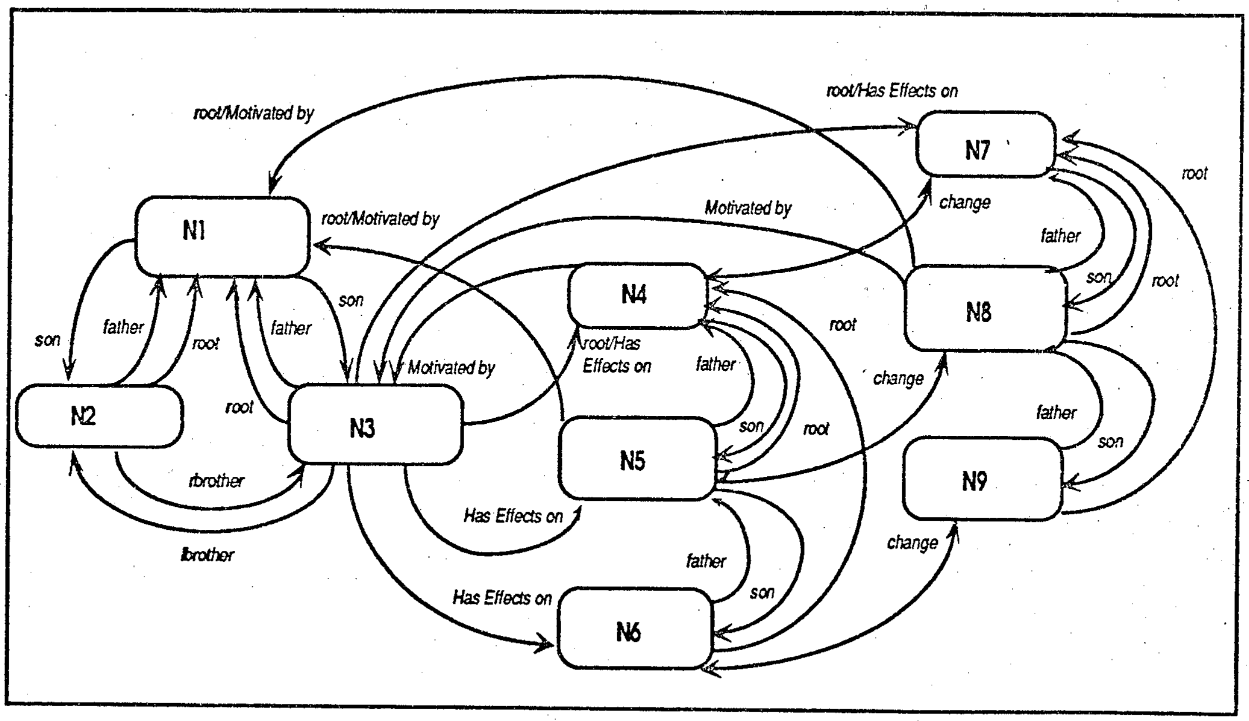


Figure 7 - Unit level links generated from hypertext in fig. 6

The reader should note that links of type Has_Effects_on point to the nodes belonging to the *default* perspective (Description) of the entity originally constituted by components Mortgage Loan Procedure, Request Acceptance and Entry, Request Data Entry.

Our hypertext is now composed of units and unit links only; it can be described by the following tables:

```
UNIT ( #unit*                   unique identifier
       #component               the Comp.this Unit belongs to
       Perspective              "style" of this Unit
       Body                     pointer to the Unit's body
       Template                 Information for the delivery environment
     );

LINK ( #source_unit             Source unit Id
       #target_unit             Target unit Id
       link_Type                describes link semantic
     );
```

Table UNIT is the same as the original UNIT table. Table LINK has the same structure as the original table APLINK, but now describes links between units instead of describing links between components.

UNIT

#Unit	#component	Perspective	Body	Template
N1	C1	Official_Text	file1	CHTempl
N2	C2	Official_Text	file2	CHTempl
N3	C3	Official_Text	file3	CHTempl
N4	C4	Description	file4	MLTempl-1
N5	C5	Description	file5	MLTempl-1
N6	C6	Description	file6	MLTempl-1
N7	C4	Code	file7	MLTempl-2
N8	C5	Code	file8	MLTempl-2
N9	C6	Code	file9	MLTempl-2

LINK

#source	#target	link_type	comments
N2	N5	Has_Effects_on	application links
N2	N6	Has_Effects_on	
N5	N3	Motivated_by	
N8	N3	Motivated_by	
N1	N2	son	links outgoing from nodes of entity Request_Data_Entry
N2	N1	father	(Official_Text perspective)
N3	N1	father	
N2	N1	root	
N3	N1	root	
N2	N3	rBrother	
N3	N2	lBrother	
N2	N4	root/Has_Effects_on	
N4	N5	son	links outgoing from nodes of entity Mortgage Loan Procedure
N5	N6	son	(Description perspective)
N6	N5	father	
N5	N4	father	

N5	N4	root	
N6	N4	root	
N5	N1	root/Motivated_by	
N8	N1	root/Motivated_by	
N7	N8	son	links outgoing from nodes of entity Mortgage Loan Procedure
N8	N9	son	(Description perspective)
N9	N8	father	
N8	N7	father	
N8	N7	root	
N9	N7	root	
N4	N7	change	Change_Perspective links
N5	N8	change	(toggle between Description and Code)
N6	N9	change	
N7	N4	change	
N8	N5	change	
N9	N6	change	

3.3.3 - From links to anchors

Now we substitute links by anchors. This operation is just a renaming and regrouping of the types of the links. The rule for changing link types is the following:

- substitute the type of each unit link with the corresponding anchor type (look for this in table LINKAN).

The table representing anchors is similar to table LINK: the only differences are the renaming of the field link_type (that now becomes anchor_type) and the change of the values it contains to the corresponding anchor types (records affected are written in bold), as follows:

Has_Effects_on → Effects

root/Has_Effects_on → Effects

Motivated_by → Motivation

root/Motivated_by → Global Motivation

ANCHOR

#source	#target	anchor_type	comments
N3	**N5**	**Effects**	application links
N3	**N6**	**Effects**	
N5	**N3**	**Motivation**	
N8	**N3**	**Motivation**	
N1	N2	son	links outgoing from nodes of entity Request_Data_Entry
N2	N1	father	(Official_Text perspective)
N3	N1	father	
N2	N1	root	
N3	N1	root	
N2	N3	rBrother	

N3	N2	lBrother	
N3	**N4**	**Effects**	
N4	N5	son	links outgoing from nodes of entity Mortgage Loan Procedure
N5	N6	son	(Description perspective)
N6	N5	father	
N5	N4	father	
N5	N4	root	
N6	N4	root	
N5	**N1**	**Global Motivation**	
N8	**N1**	**Global Motivation**	
N7	N8	son	links outgoing from nodes of entity Mortgage Loan Procedure
N8	N9	son	(Code perspective)
N9	N8	father	
N8	N7	father	
N8	N7	root	
N9	N7	root	
N4	N7	change	Change_Perspective links
N5	N8	change	(toggle between Description and Code)
N6	N9	change	
N7	N4	change	
N8	N5	change	
N9	N6	change	

3.3.4 - Anchor filtering

Finally we delete the anchor links whose type is not allowed for the perspective of their source unit (look for this in table ANPERSP). The result is that, in the final database, anchor types Motivation and Global Motivation have been removed from perspective Code by removing the corresponding anchors connecting unit N8 to N1 and N3.

3.3.5 - Chooser generation

The particular browsing semantics we have chosen requires the generation of a chooser mechanism for each multiple destination anchor (see section 2.2.1.2). This is done by scanning the table ANCHOR, looking for anchors with the same source node (same #source and anchor_type field values). For each set of such anchors, a new chooser node is generated, the destination node (#target field) of the original anchors is changed to point to the new chooser, and new records are created, to include anchors from the chooser to the original destinations. The table below shows the updates to the ANCHOR relation of the example.

ANCHOR

#source	#target	anchor_type	comments
N3	**CH1**	**Effects**	previously pointing to *N5*
N3	**CH1**	**Effects**	previously pointing to *N6*

•			
•			(same as before)
•			
N3	**CH1**	**Effects**	previously pointing to *N4*
•			
•			(same as before)
•			
			(below are the new chooser anchors)
CH1	**N5**	**1**	chooser anchor for first node
CH1	**N6**	**2**	chooser anchor for second node
CH1	**N4**	**3**	chooser anchor for third node

A new record for each generated chooser is also created and included in table DEUNIT. The #Unit field has an automatically generated value; the Body field has a constant value "chooser" (which allows the recognition of chooser nodes); and the Template field contains a reference to a standard chooser template. This is exemplified in table DEUNIT (DElivery UNITs) shown below.

The whole hypertext is now represented by the tables ANCHOR and DEUNIT. Table DEUNIT differs from UNIT in that the attribute Perspective has been replaced by Title, whose contents are derived from the Title field of the Component table the unit belonged to and the Perspective field of the Unit table. For choosers, the Title field has its value composed with the values of the Title field of the source of the chooser and the anchor_type of the chooser.

The final output of the first compilation step, which is called the Intermediate Model, is the representation of an oriented graph whose nodes (units) are the same as those in the original hypertext, and links connect nodes instead of connecting the original HDM components. This simplified version is described by the following two tables (changes of table Anchors with respect to table Links are shown in bold).

DEUNIT (Delivery Unit)

#Unit	Title	Body	Template
N1	Circular HyperBank 21/10/89:Official Text	file1	CHTempl
N2	Operational Norms in Request Verif:Official Text	file2	CHTempl
N3	Subject:Official Text	file3	CHTempl
N4	Mortgage Loan Procedure:Description	file4	MLTempl-1
N5	Request Acceptance and Entry:Description	file5	MLTempl-1
N6	Request Data Entry:Description	file6	MLTempl-1
N7	Mortgage Loan Procedure:Code	file7	MLTempl-2
N8	Request Acceptance and Entry:Code	file8	MLTempl-2
N9	Request Data Entry:Code	file9	MLTempl-2
CH1	Circular HyperBank 21/10/89:Effects	chooser	ChsTempl

ANCHOR

#source	#target	anchor_type	comments
N3	**CH1**	**Effects**	application links; note anchor
N3	**CH1**	**Effects**	for these nodes points to choosers

N5	**N3**	**Motivation**	
(deleted)			(old *N8 → N3: Motivation*) - not allowed for this perspective
N1	N2	son	links outgoing from nodes of entity Request_Data_Entry
N2	N1	father	(Official_Text perspective)
N3	N1	father	
N2	N1	root	
N3	N1	root	
N2	N3	rBrother	
N3	N2	lBrother	
N3	**CH1**	**Effects**	
N4	N5	son	links outgoing from nodes of entity Mortgage Loan Procedure
N5	N6	son	(Description perspective)
N6	N5	father	
N5	N4	father	
N5	N4	root	
N6	N4	root	
N5	**N1**	**Global Motivation**	
(deleted)			(old *N5 → N1: Global Motivation*) - not allowed for this perspective
N7	N8	son	links outgoing from nodes of entity Mortgage Loan Procedure
N8	N9	son	(Code perspective)
N9	N8	father	
N8	N7	father	
N8	N7	root	
N9	N7	root	
N4	N7	change	Change_Perspective links
N5	N8	change	(toggle between Description and Code)
N6	N9	change	
N7	N4	change	
N8	N5	change	
N9	N6	change	
CH1	**N5**	**1**	chooser anchor for first node
CH1	**N6**	**2**	chooser anchor for second node
CH1	**N4**	**3**	chooser anchor for third node

3.3.5 - Translation into HyperCard

The purpose of this step is to generate an application in HyperCard, the input being the tables ANCHOR and DEUNIT; the problem then becomes the translation of delivery nodes and anchor links into HyperCard objects.

The most natural choice for what concerns nodes is to map them onto HyperCard cards. Unfortunately there is no such simple choice for anchors since HyperCard does not directly implement links as such - they are actually coded as part of scripts associated to buttons. When a button is "clicked", the associated script is executed, and if it contains a "go to card ..." command, the effect is the same as a link traversal.

The first alternative for compiling HDM anchors is to generate one script for each corresponding button. Another possibility is to have only one "generic" parameterized script for all buttons, where the destination is given by the parameter, usually found in a table. In HyperCard this can be achieved by placing the script to handle mouse clicks at the stack level.

We have opted for the second approach simply because it allows for a simpler compiler, and, secondarily, because it makes the resulting HyperCard stack easier to maintain within the HyperCard environment. It should be noted that, in any case, either option is possible; had we chosen the first alternative above, only a small change to the overall compiler would be needed. Furthermore, with a simple utility program written in HyperTalk (the scripting language of HyperCard) it is possible to convert one representation into the other.

Thus, we have implemented a small engine implemented in Hypertalk which detects user navigation requests, retrieves the name of the target node from a table, and finally instructs the HyperCard engine to show the target card. A simplified fragment of the script handling button activation is given in figure 10. The name of the target node is retrieved by the engine from a table (conceptually identical to table `ANCHOR`) that describes links in terms of triplets `<anchor_type, source_card, target_card>`; this table is stored in a hidden field of a card in the stack. From the user's point of view, anchor links are mapped onto HyperCard buttons placed in source cards.

The final thing that must be translated from IM into HyperCard are chooser nodes, which have a different structure from other nodes. We solve this problem, in the case of HyperCard, by connecting the source card to a special type of card generated from a chooser template, containing fields with a short description (title) of the target cards; the reader selects the actual target by clicking on its title. Choosers are generated automatically by the HF compiler.

If we had chosen a different browsing semantics, choosers could have been implemented as a pull-down menu whose items would be target node titles; this solution would have freed some screen space on the card but seemed to us to be less natural for the typical HyperCard user.

Now we shall examine in detail the stages of this compilation step.

Stage 1 - Nodes generation

For each delivery node HF generates an HyperCard node (card).

The node is generated cloning (copying) a prototype card using the card whose ID is given in the `template` field of table `DEUNIT`, and whose content is specified in the `body` field.

Prototype cards can be freely designed, except for the following constraints:

- They must have a button for each outgoing anchor link, so that users can activate all anchors;

- They must have a field to hold the title of the node. Whereas the first restriction above is logically necessary for consistency, this restriction is rather of a stylistic nature. It is logically necessary for chooser templates only (since the user selects target nodes clicking on their titles), but requiring it for all templates imposes some uniformity on the visual identification of all cards. This also helps to convey the notion of "component".

After having been cloned from its prototype the newly created card is assigned an unique identifier and its title (respectively, the `#unit` and `title` fields of the `DEUNIT` table.

For example, the card corresponding to unit `N1` in that table will have (HyperCard) name `N1` and the value of the HyperCard field "`Title`" equal to "`Circular HyperBank:Official Text`". If the card shown in figure 8 is the template for that node (i.e., the card with name `CHTempl`, taken from field `template` of that record in table `DEUNIT`), then the cloned card for unit `N1` will be the one shown in figure 9.

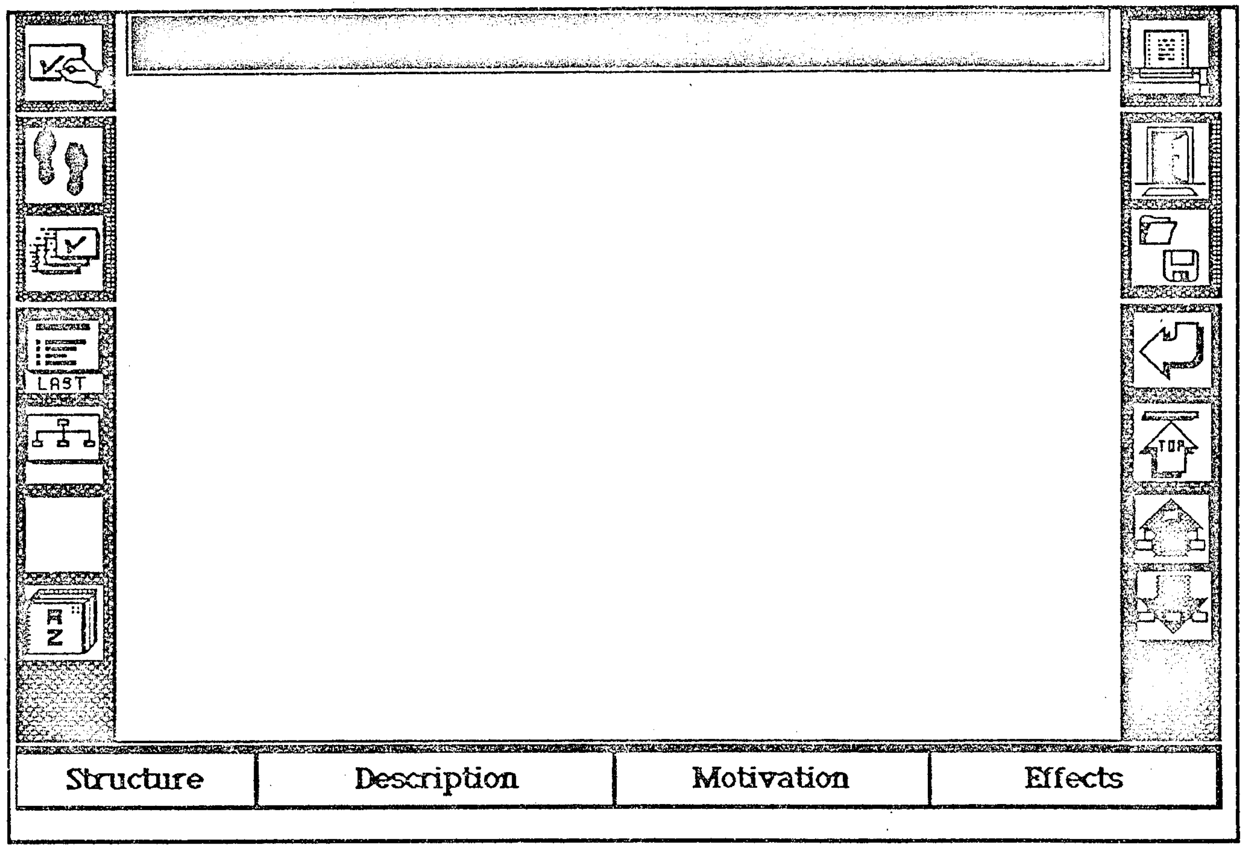
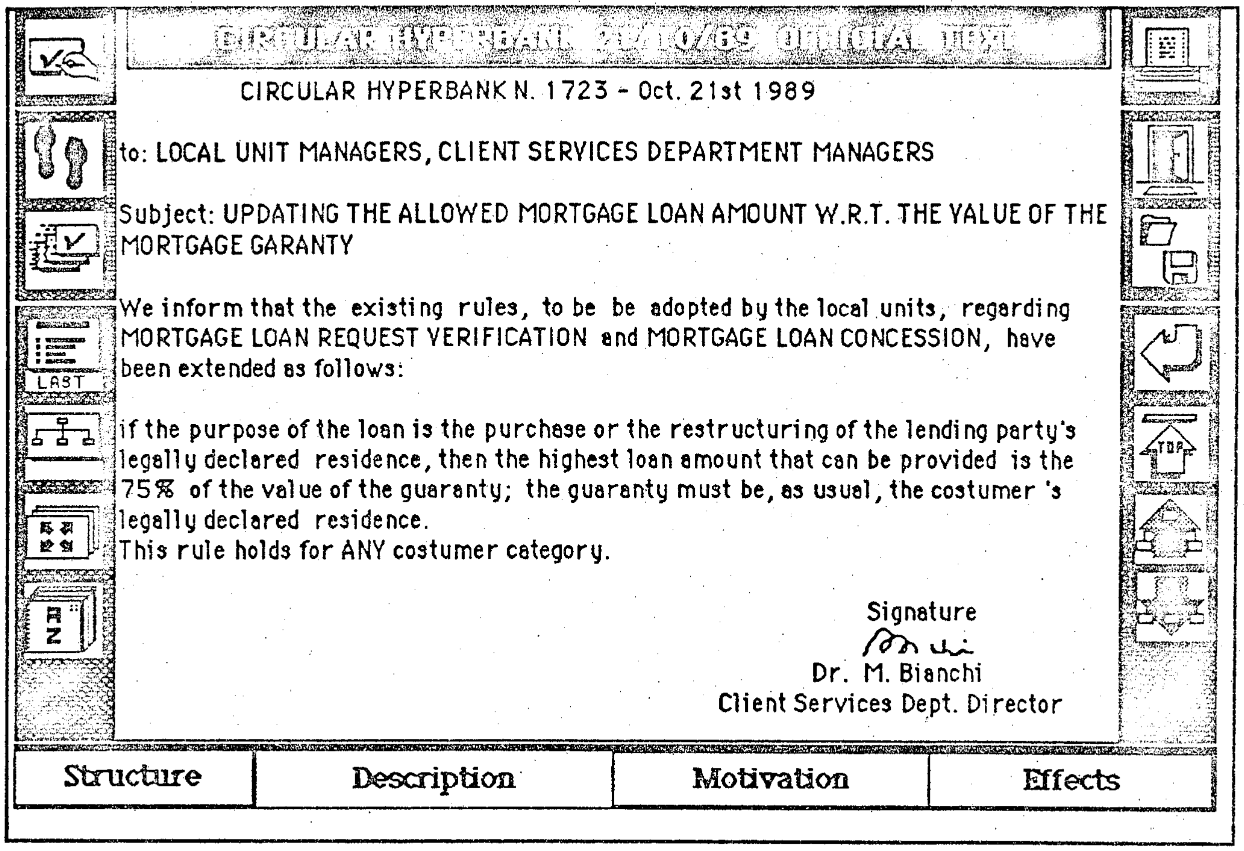


Figure 8 - Template card corresponding to `template` field value `CHTempl` in table `DEUNIT`

A few remarks about figures 8 and 9 are in order. First of all, since we are illustrating just a portion of a more complex application, there are buttons which correspond to anchors we have not mentioned ("Motivation", "Description", "Structure"). The group of buttons (icons) at the bottom right hand of the screen correspond to anchors for structural links. All the other buttons on the sides correspond to functions either related to navigation or to system operations. Navigation functions can be "Trace", "Navigation Map", "Mark Node", etc...; system functions allow, for example, access to files, printing and the such.


CIRCULAR HYPERBANK 21/10/89 OFFICIAL TEXT

CIRCULAR HYPERBANK N. 1723 - Oct. 21st 1989

to: LOCAL UNIT MANAGERS, CLIENT SERVICES DEPARTMENT MANAGERS

Subject: UPDATING THE ALLOWED MORTGAGE LOAN AMOUNT W.R.T. THE VALUE OF THE MORTGAGE GARANTY

We inform that the existing rules, to be be adopted by the local units, regarding MORTGAGE LOAN REQUEST VERIFICATION and MORTGAGE LOAN CONCESSION, have been extended as follows:

if the purpose of the loan is the purchase or the restructuring of the lending party's legally declared residence, then the highest loan amount that can be provided is the 75% of the value of the guaranty; the guaranty must be, as usual, the costumer 's legally declared residence.
This rule holds for ANY costumer category.

Signature
[illegible]
Dr. M. Bianchi
Client Services Dept. Director

Figure 9 - Card corresponding to node N1 in table DEUNIT. Buttons "Motivation", "Description" and "Structure" should be ignored for this example.

Stage 2 - Links generation

Since we do not use the standard linking technique of HyperCard, link generation simply consists in generating a table of triplets `<anchor_type, source_card, target_card>`, which gets encoded into invisible fields of an HyperCard card. Figure 10 contains part of the HyperCard script that handles button clicks, thereby implementing the small hypertext engine.

```
--  This script is placed at the Stack level of the Hypercard
hierarchy
-- ************************************************************
--  ********* Captures mouse clicks on buttons ***************
on mouseup

  if "button" is in the name of the target¬
  or "field" is in the name of the target
  then
    set cursor to busy
    lock screen
    set lockMessages to true
    ActivateAnchor the short name of the target &","&¬
    the short name of this cd
  else
    exit mouseup
  end if
  unlock screen
end mouseup

--  *********************************************
--  ************* ACTIVATE ANCHOR ****************
on  ActivateAnchor anchor
  put the name of this card into CurrentNode -- for error recovery
  go to cd "delink"       -- Contains table "delink" in its fields
  -- Now find the target node of this anchor by looking in the fields
  -- "DELINKn" of card "delink". This done to simply to cirmcumvent
  -- Hypercard's 32K limitation on field sizes
  put 4 into maxtabdelink       -- Number of fields to encode table
                                -- delink, in card "delink".
  put false into found
  repeat with n=1 to maxtabdelink
    set cursor to busy
    do "find string anchor in fld delink"&n&&"of cd delink"
    if the result is empty and the short name of this cd is "delink"
    then
      put true into found
      put item 3 of the value of the foundline into targetnode
      exit repeat
    end if
  end repeat
  if not found then
    -- Error - should not happen since it is checked at compile time
    ErrorHandler(CurrentNode)
    exit  ActivateAnchor
  end if
  go to targetnode
end  ActivateAnchor
```

Figure 10 - Script for handling button clicks. It is placed at the stack level.

Stage 3 - Choosers

The purpose of this stage is to implement the support for one-to-many links. In HyperCard this is implemented generating some cards (choosers) which allow the reader to select one of the (many) target cards. Chooser cards are cloned from a standard chooser template card, whose `title` field is assigned the title of the source node concatenated with the type of the anchor leading to that chooser. Each chooser and has as many clickable fields as the destinations. Each destination field contains the title of a target node, and the user chooses the node she wants to see by clicking on its title.

If the value "`ChsTempl`" of field `template` in the record for chooser `CH1` in table `DEUNIT` is the name of the card shown in figure 11, the corresponding chooser cloned from it is the one shown in figure 12.

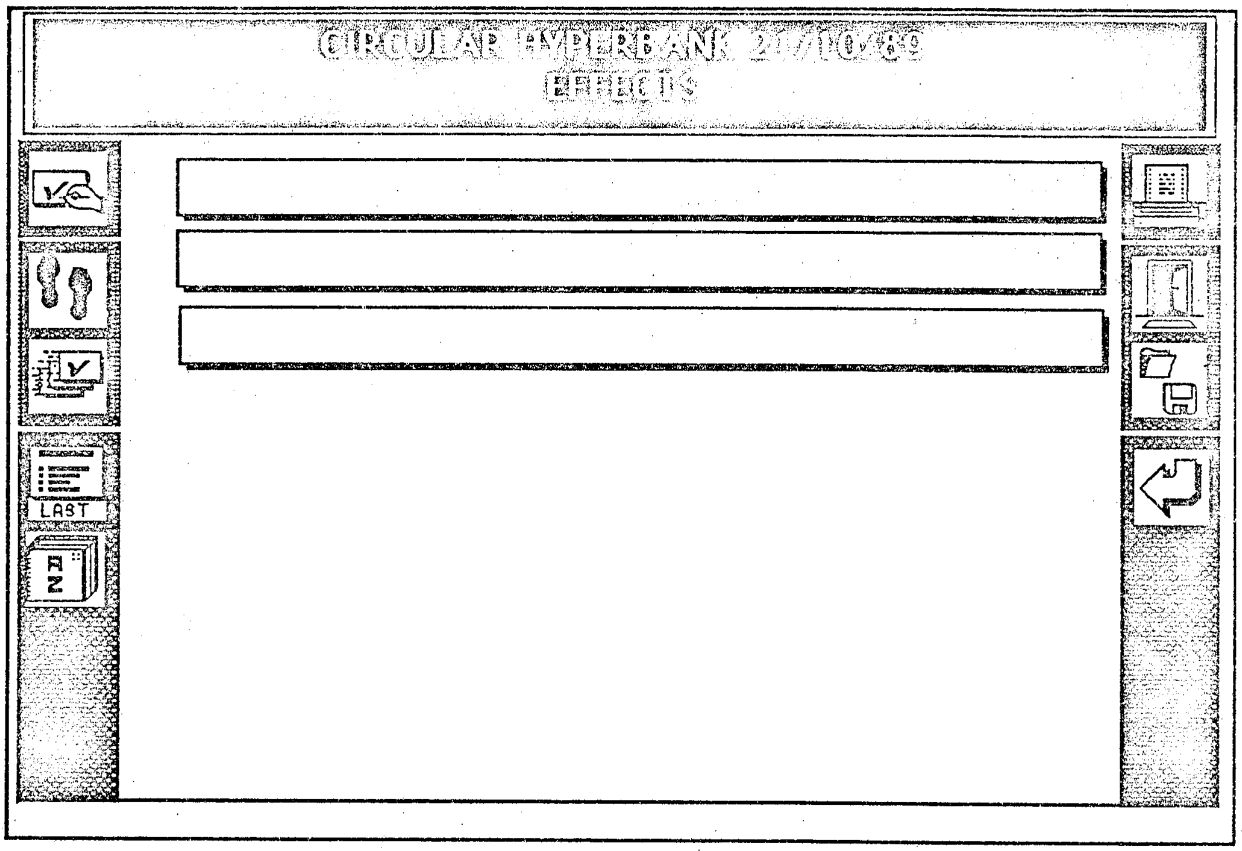


Figure 11 - Template card corresponding to the `template` field value `ChsTempl` in table DEUNIT.

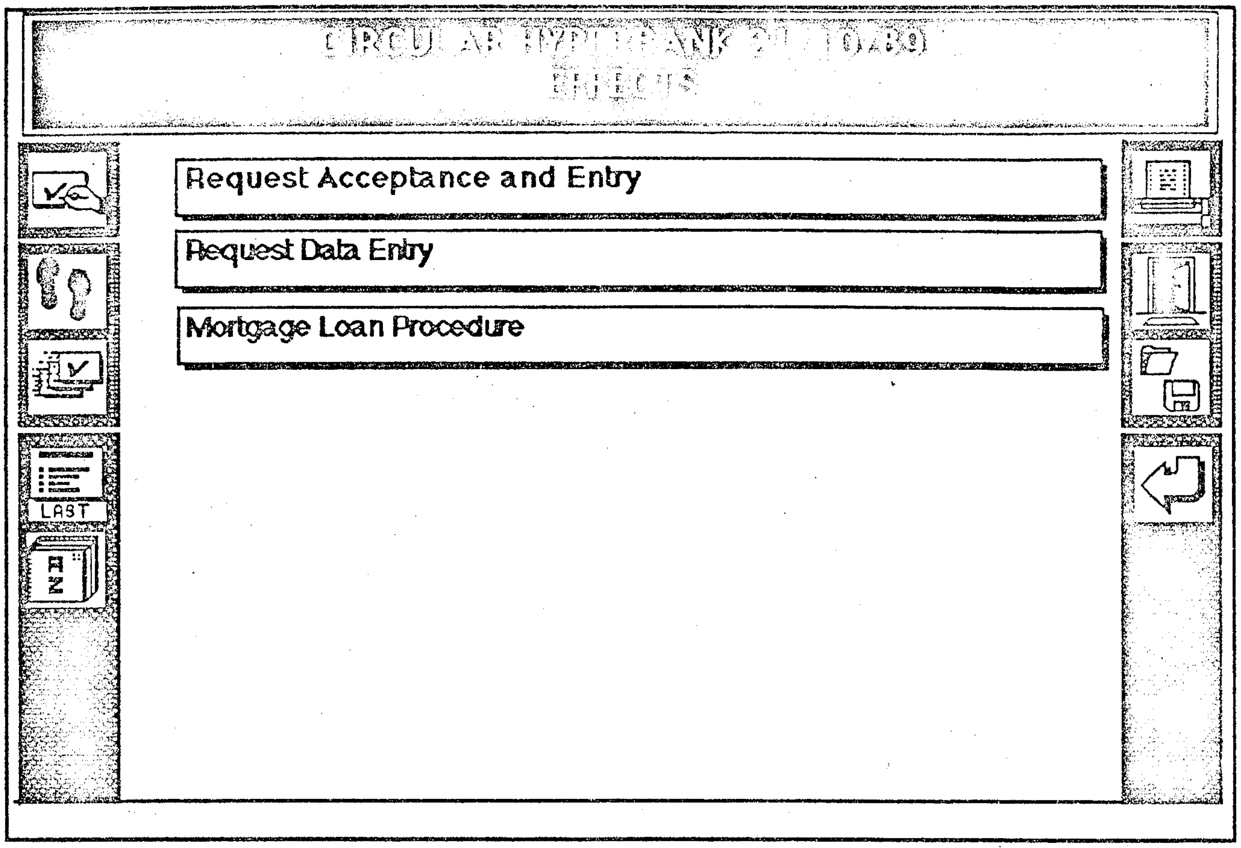


Figure 12 - Chooser corresponding to node CH1 in table DELINK.

Once this stage is finished, the resulting stack contains the final application, which can be run using HyperCard.

4. Conclusions

We have shown how a model-based approach can enhance the design of an hypertext; although the example shown is only a small part of a large application, it should be clear how the systematic approach allows to handle a quite complex structure in an organized way. The existence of a model allows relatively simple tools to considerably speed up hypertext structure development, as can be verified, for example, by comparing the complexity of the structures depicted in figures 5 and 7. Moreover, this complexity is managed in such a way that should render it easier on readers that must "navigate" in such hypertexts.

It is clear that this approach is less advantageous in application domains where there are few structural regularities, or where such regularities should not be exploited too much (in certain games, for instance). Even in such cases, though, the approach described here can serve as a starting point from which the author may continue embellishing the hypertext structure.

The compilation schema presented here allows several degrees of "reusability" of the code implementing the compiler. The first degree is when, maintaining the same browsing semantics, one generates the hypertext for a different system. In this case, only the last step of the compilation process (targetting of the IM) must be implemented again. The second degree is when the browsing semantics is changed; if the only thing that is changed is the behaviour upon link activation , then again most of the compilation process can be reused.

The approach we have described takes a hitherto new HDM specification and generates an empty HyperCard stack to be filled in. Clearly, if the author changes her mind about the hypertext structure after having compiled its specification, she would not want to re-specify unchanged nodes. In order to accomodate this need our approach should be extended to allow for incremental compilation. Here, rather than starting with a completely new design, the author starts with a previously defined specification. In this situation, the compiler must be extended in to keep track of altered information (mainly link information), so that only new additions and changes to previously defined objects must be generated.

As a more general comment, model specifications and node content editing should ideally be processed within an *integrated* environment, allowing the author to move "seamlessly" from the specification level to the delivery environment level. As mentioned previously, such an environment is being defined and developed within the Esprit II Project HYTEA [ARG 90], of which Politecnico di Milano is a partner.

5. References

[ARG 90] ARG, "HYTEA Technical Annex", Esprit Project P5252, June 1990

[Akscyn 87] Akscyn, R.; McCracken, D.; Yoder, E.; "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations", Proc. Hypertext '87, ACM, Baltimore, 1987, pp. 1-20

[Atkinson 87] Atkinson, W.; HyperCard, software for Macintosh computers, Cupertino, Apple Computer Co, 1987.

[Brown 87] Brown, P.J.; "Turning Ideas Into Products: The Guide System", Proc. Hypertext '87, ACM, Baltimore, 1987. pp. 33-40

[Brown 89] Brown, P.J.; "Do we need maps to navigate round hypertext documents?", Eletronic Publishing-Origination, Dissemination and Design 2, 2 (July 89).

[Furuta 90] Furuta R., Stotts D., The Trellis Reference Model", Proc. 1st Hypertext Standardization NIST Workshop, Gaithersburg, MD, Jan. 1990

[Garzotto 89] Garzotto F., Paolini P., " Expert Dictionaries: Knowledge Based Tools for Explanation and Maintenance of Complex Application Environments", Proc. 2nd ACM Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Tullahoma , TN., Aug. 1989

[Garzotto 90a] Garzotto F., Paolini P., Schwabe,D., Bernstein, M. "Tools for Developers", Chapter 5 of "Hypertext/Hypermedia Handbook", Devlin, J. ; Berk, E. (eds), McGraw Hill 1990.

[Garzotto 90b] Garzotto F., Schwabe,D.; Paolini P.; Caloini, A. Mainetti, S.; Borroni,S, "HDM - HYPERMEDIA DESIGN MODEL", Tech. Report.m 90-41, Dept. of Electronics, Politecnico di Milano, Oct. 1990

[Garzotto 90c] Garzotto F., Schwabe,D.; Paolini P.;, "HDM - A Model Based Approach to Hypermedia Application Design", submitted to ACM - TOIS, November 1990. Also available as Technical Report 90-??, Dipartimento di Elettronica, Politecnico di Milano, Nov. 1990.

[Halasz 90] Halasz F., Schwartz M, " The Dexter Refernce Model", Proc. 1st Hypertext Standardization NIST Workshop, Gaithersburg, MD, Jan. 1990

[Stotts 89] Stotts, P.D.; Furuta, R., "Petri-Net-Based Hypertext: Document Structure with Browsing Semantics", ACM Transactions on Office and Information Systems, 7(1), January 1989.

[Utting 90] Utting K; Yankelovich, N.; "Context and Orientation in Hypermedia Networks", ACM Trans. on Information Systems, 7. (1990) pp. 58-84

6. Appendix: SQL Code

In this appendix we show the SQL code corresponding to the compilation steps described in the article. The implementation of the chooser cloning mechanism requires the use of embedded SQL, and we have therefore chosen to implement it in HyperCard, rather than having it in yet another programming environment. This part is not shown in the appendix.

```
REM /* CREATE Tables */

CREATE TABLE ENTITY (#ENTITY               CHAR(4)    NOT NULL,
                     ENTITY_TYPE           CHAR(15)   NOT NULL,
                     DEFAULT_PERSPECTIVE   CHAR(15)   NOT NULL
                     );

CREATE TABLE COMPON (#COMPONENT            CHAR(4)    NOT NULL,
                     #FATHER               CHAR(4),
                     #ENTITY               CHAR(4)    NOT NULL,
                     TITLE                 CHAR(45),
                     ORDER_NUMBER          NUMBER     NOT NULL
                     );

CREATE TABLE APNODE (#UNIT                 CHAR(4)    NOT NULL,
                     #COMPONENT            CHAR(4)    NOT NULL,
                     PERSPECTIVE           CHAR(15)   NOT NULL,
                     BODY                  CHAR(45)   NOT NULL,
                     TEMPLATE              CHAR(45)
                     );

CREATE TABLE APLINK (#SOURCE      CHAR(4)    NOT NULL,/* APplication LINK */
                     #TARGET      CHAR(4)    NOT NULL,
                     LINK_TYPE    CHAR(15)   NOT NULL
                     );

CREATE TABLE LINKAN (ANCHOR_TYPE      CHAR(15)  NOT NULL,      /* ANchor LINK */
                     LINK_TYPE        CHAR(15)  NOT NULL
                     );

CREATE TABLE ANPERSP (ANCHOR_TYPE          CHAR(15)   NOT NULL,
                      PERSPECTIVE          CHAR(15)   NOT NULL
                      );

REM /* Structural Link Derivation - Generates Table COMP_STRUCT_LINK */

CREATE TABLE COMP_STRUCT_LINK (#SOURCE     CHAR(4)      NOT NULL,
                                                  /* COmponent STructural LinK */
                               #TARGET     CHAR(4)      NOT NULL,
                               LINK_TYPE   CHAR(15)     NOT NULL
                                                      );

COMMIT;
```

```
INSERT INTO COMP_STRUCT_LINK
        SELECT #COMPONENT, #FATHER, 'FATHER'    /*FATHER Relation */
        FROM COMPON
        WHERE #FATHER != 'NULL' ;

COMMIT;

INSERT INTO COMP_STRUCT_LINK
        SELECT #FATHER, #COMPONENT, 'SON'       /* SON Relation */
        FROM COMPON
        WHERE #FATHER != 'NULL'
          AND ORDER_NUMBER=1 ;

COMMIT;

INSERT INTO COMP_STRUCT_LINK                    /* RIGHT BROTHER Relation */
        SELECT COLB.#COMPONENT, CORB.#COMPONENT, 'RBROTHER'
        FROM COMPON COLB, COMPON CORB      /* COLB = COmponent Left Brother */
                                           /* CORB = COmponent Right Brother */
        WHERE COLB.#FATHER=CORB.#FATHER
          AND COLB.ORDER_NUMBER=CORB.ORDER_NUMBER-1 ;

COMMIT;

INSERT INTO COMP_STRUCT_LINK                    /* LEFT BROTHER Relation */
        SELECT CORB.#COMPONENT, COLB.#COMPONENT, 'LBROTHER'
        FROM COMPON COLB, COMPON CORB
        WHERE COLB.#FATHER=CORB.#FATHER
          AND COLB.ORDER_NUMBER=CORB.ORDER_NUMBER-1 ;

COMMIT;

INSERT INTO COMP_STRUCT_LINK                    /* ROOT Relation */
        SELECT COMP.#COMPONENT, ROOT.#COMPONENT, 'ROOT'
        FROM COMPON COMP, COMPON ROOT
        WHERE COMP.#ENTITY = ROOT.#ENTITY
          AND ROOT.#FATHER IS NULL;

COMMIT;

REM /* Application Links. Table COAPLK */

CREATE TABLE COAPLK (#SOURCE,                /* COmponent APplication LinK */
                     #TARGET,
                     LINK_TYPE
                    )
AS
        SELECT *                             /*Copy application links */
        FROM APLINK ;

COMMIT;

INSERT INTO COAPLK                           /* Generate derived links */
        SELECT DISTINCT #SOURCE, ROOT.#COMPONENT, LINK_TYPE
        FROM  APLINK, COMPON TARG, COMPON ROOT
        WHERE APLINK.#TARGET = TARG.#COMPONENT
          AND TARG.#ENTITY = ROOT.#ENTITY             /* for each link  */
          AND ROOT.#FATHER IS NULL;                   /* find its root  */

COMMIT;
```

```
REM /* Temporary Table for Units */

CREATE TABLE NOLINK (CD_NOS,            /* NOde LINK - Links between Units*/
                     CD_NOT,
                     LINK_TYPE
                    )
AS
        SELECT DISTINCT NODES.#UNIT, NODET.#UNIT, COMP_STRUCT_LINK.LINK_TYPE
/*Structural Links*/
        FROM COMP_STRUCT_LINK, APNODE NODES, APNODE NODET  /*Remember       */
                  /* structural links connect components of the same entity */
        WHERE COMP_STRUCT_LINK.#SOURCE = NODES.#COMPONENT  /* Connect units */
          AND COMP_STRUCT_LINK.#TARGET = NODET.#COMPONENT  /* with the same */
          AND NODES.PERSPECTIVE        = NODET.PERSPECTIVE; /* perspective, */
                  /* if their components are connected by structural links  */

COMMIT;

INSERT INTO NOLINK
  SELECT DISTINCT NODES.#UNIT, NODET.#UNIT, COAPLK.LINK_TYPE
/*Application Links*/
  FROM COAPLK, COMPON, APNODE NODES, APNODE NODET, ENTITY  /*Remember      */
            /* application links connect components of different entities  */
  WHERE COAPLK.#SOURCE     = NODES.#COMPONENT  /* Connect components linked */
    AND COAPLK.#TARGET     = NODET.#COMPONENT  /* by application links.     */
    AND COAPLK.#TARGET     = COMPON.#COMPONENT /*  Links go from units of   */
    AND COMPON.#ENTITY     = ENTITY.#ENTITY    /* each perspective to the   */
    AND NODET.PERSPECTIVE = ENTITY.DEFAULT_PERSPECTIVE; /* default persp.   */

COMMIT;

INSERT INTO NOLINK                              /*Change Perspective links  */
  SELECT SOURCE.#UNIT,TARGET.#UNIT,'PERPSPECTIVE'
  FROM APNODE SOURCE,APNODE TARGET
  WHERE SOURCE.#COMPONENT =  TARGET.#COMPONENT /* Connect each persp to all */
    AND SOURCE.#UNIT != TARGET.#UNIT;          /* other perscpectives       */

COMMIT;

REM /* Generated Delivery Link Table */

CREATE TABLE DELINK (ANCHOR_TYPE,           /* DElivery LINK */
                     CD_NOS,
                     CD_NOT
                    )
AS
SELECT ANPERSP.ANCHOR_TYPE, CD_NOS, CD_NOT
FROM ANPERSP, LINKAN, NOLINK, APNODE
WHERE ANPERSP.PERSPECTIVE = APNODE.PERSPECTIVE    /* if node n with persp.p */
AND APNODE.#UNIT = NOLINK.CD_NOS /* is source of link of type lt to node NOT*/
AND LINKAN.LINK_TYPE = NOLINK.LINK_TYPE /* get anchor_type at corresp to lt */
AND ANPERSP.ANCHOR_TYPE = LINKAN.ANCHOR_TYPE;     /* if at is allowed for p */

COMMIT;
```

```
REM /* Generate Delivery Units */

CREATE TABLE DEUNIT (#UNIT,                /*DElivery NODE */
                     TITLE,
                     BODY,
                     TEMPLATE
                    )
AS SELECT #UNIT, TITLE | ":" | PERSPECTIVE, BODY, TEMPLATE
   FROM APNODE, COMPON
   WHERE APNODE.#COMPONENT = COMPON.#COMPONENT;

COMMIT;
```