Proceedings of the Sixth IFIP Conference on Data Semantics (DS-6), Atlanta, Georgia, May, 1995;
to appear in: R. Meersman, L. Mark [Ed.]: Database Application Semantics, Chapman and Hall, 1996

Flexible Design and Efficient Implementation of a Hypermedia Document Database System by Tailoring Semantic Relationships

J. Wäsch, K. Aberer
GMD-IPSI Integrated Publication and Information Systems Institute
Dolivostraße 15, D-64293 Darmstadt, Germany
Tel.: ++49-6151-{959, 935}, Fax: ++49-6151-966
Email: {waesch, aberer}@darmstadt.gmd.de

Abstract

In this paper we present the design concepts and data modeling approach that was used to define a general application framework for storing hypermedia documents in the object-oriented database management system VODAK. We exploit the capabilities of the VODAK data model to introduce new hypermedia modeling primitives at the meta level. We show that representing the hypermedia semantics within the DBMS in this way is clearly advantageous for efficiency of the design and implementation of hypermedia document storage engines. As an example, we give a concrete realization of a DBMS-based hypermedia engine for the SEPIA hypermedia authoring system developed at GMD-IPSI. With this example we show that the data model extensions provided are flexible enough to represent also complex semantic hypermedia concepts, that the development cycle of a hypermedia engine can become extremely fast and that the resulting implementation has adequate performance for interactive applications.

Keywords

Hypermedia, Object-Oriented Database Management Systems, Semantic Data Modeling, Document Databases

1 INTRODUCTION

Three layers can be identified for the architecture of hypermedia systems (Campell and Goodman 1988): the storage layer, which provides basic functionality with regard to persistent storage of data, the application layer, which provides the particular semantics of hypermedia applications, and the presentation layer which supports human-machine interaction. In first generation hypermedia systems these three layers are typically combined in one monolithic stand-alone system, where all of the functionality is realized regardless whether it is general-purpose or hypermedia-specific. This approach was adequate for certain applications and for research, but did not exploit existing solutions from other areas, and could not serve as the basis of future integrated information systems. This was soon recognized, and with regard to the storage layer, either database management systems (DBMSs) were used, like in HyperBase (Schütt and Streitz 1990) or HyperPath/O2 (Amann et al. 1993), or special-purpose storage managers were developed, like HAM (Campell and Goodman 1988) or Hyperform (Wiil and Leggett 1992). The disadvantage of the second approach is that many of the standard DBMS functionalities needed for persistent storage and sharing of data, like transaction management or declarative access, have either to be re-implemented, e.g., concurrency control, or are not available at all, e.g., declarative query languages.

Meanwhile there is a common agreement, that among the different kinds of database management systems, object-oriented DBMSs (OODBMS) are the best choice to be used for the storage layer of hypermedia systems (Lange 1993). The main reason is that their data model allows a direct representation of complex networks, and that by the encapsulation of structure and behavior much of the necessary semantics of hypermedia structures can be captured by the database management system. Also some advanced DBMS features, that can be useful for hypermedia applications, like version control, check-in/check-out mechanisms or nonstandard transaction concepts, are most often found in object-oriented database management systems.

Common to the different approaches to hypermedia system storage layers, both DBMS-based and special purpose, is the assumption of a relatively simple hypermedia core model, that has to be used by all applications. Thus, applications have to deal with the more advanced aspects of their hypermedia structures by themselves, and have to map their more complex structures and operations from the application layer, to the generic core model. A typical example of the potential complexity of the semantics of hypermedia systems is given by SEPIA, a hypermedia system for supporting authors of hypermedia documents. We will introduce SEPIA in section 2.1 and use it as a test case for our approach. A storage system based on a generic core model does not support the maintenance of the consistency of the hypermedia model with regard to the complex application semantics. Extensibility of the core model in some general way, e.g., by means of subtyping as in Hyperform, does not improve the situation. It is just a more elegant way to combine generic semantics with the additional application semantics, whereas the difficult task to maintain integrity and to provide complex operations is still delegated to the application programmer.

In order to provide the application programmers with the means to develop hypermedia systems in a flexible way, and to obtain powerful storage capabilities with little effort, the DBMS's data model has to support additional data modeling primitives for hypermedia modeling. Then the application programmer can flexibly combine those primitives to his intended, semantically rich hypermedia model, and obtains appropriate support for the primitives' semantic by the DBMS. This support can be given in the form of consistency constraint checking and provision for data structures and operations, which are fully controlled by the DBMS.

In this paper we describe how such an approach was realized with the object-oriented DBMS VODAK (Klas et al. 1994, VODAK 1995). A necessary condition in order to proceed in the way envisaged is that the DBMS's data model must be flexible enough to allow the definition of new data-modeling primitives. In case of VODAK this is achieved through a metadata layer, that allows to introduce new modeling primitives into the data model. With our approach we provide hypermedia data-modeling primitives as extensions of the DBMS's data model that, in this way, is tailored to the specific needs of hypermedia applications. Thus, we fully support the storage layer for hypermedia applications and provide the application layer with a much more powerful data model than this is the case with a hypermedia core model. As a consequence, much of the application layer semantics is realized within the DBMS. Thus, together with the data model extensions, VODAK actually turns out to be a *hypermedia engine*.

By proceeding in the way described, we obtain the following additional advantages with regard to other approaches realizing a hypermedia system storage layer:

1. Applications are relieved from defining complex hypermedia semantics themselves as they can adapt the given modeling primitives to their needs. This provides the applications with powerful predefined operations and avoids a mapping from the more complex application model to the storage model, i.e. there is no discrepancy between models. Hence, development of complex hypermedia applications is eased.
2. Complex operations can be performed completely within the DBMS more efficiently. This can, for example, massively reduce network communication costs, or relieve the application programmer from implementing complex caching strategies. DBMS-based hypermedia applications can become more efficient.
3. Consistency is enforced by the DBMS not only with regard to the hypermedia core model, but also with regard to the more complex application model. This is ensured for multi-user access – also in the case where different applications access the same hypermedia database, as there is a common data model for different applications – and in case of failure of the system by concurrency control and transaction management. The mechanism to accomplish this is to allow only meaningful and consistency-preserving operations with regard to the high-level application semantics on the database as possible transactions.
4. The declarative querying facilities the DBMS provides can fully exploit the semantic features of the application-specific model, and thus become a flexible and powerful tool for accessing the hypermedia database. The processing power that a DBMS query module provides, for example query optimization and efficient query evaluation, contributes to the flexibility and efficiency of hypermedia systems.

Additional advantages to be mentioned are the simpler interaction of hypermedia applications with other DBMS applications and the exploitation of other DBMS services, like for example the advanced multimedia features of VODAK.

We want to point out that the approach described addresses many important, although not all problems related to the access of hypermedia databases. One major research issue we want to mention is the collaborative editing of hypermedia documents, which leads, for example, to the view update problem. By defining appropriate transactions and transaction boundaries there is some flexibility in governing the behavior of the database system when multiple users access and change the same documents simultaneously. In VODAK, additionally, the concept of semantic concurrency control (Muth et al. 1992, Muth et al. 1993) allows to exploit the semantics of operations, which for the previous arguments is known to the DBMS, in order to increase concurrency. However, the role of the DBMS is restricted to isolate users from each other and to maintain consistency under all circumstances. The problem of having consistent views when editing in parallel and cooperatively will need further research efforts. However, as this issue is a hot topic in DBMS research, by providing a powerful DBMS-based storage layer also hypermedia systems will profit from new results. Through the provision of a semantically rich model in the DBMS the application of such techniques will naturally be eased.

A central question to be answered when introducing a new concept that enhances the functionality of an information system, is always whether the price to be paid in terms of additional overhead and thus system performance is acceptable. That this is a critical issue is best understood when considering the difficulties relational database systems have when used to store complex information like that occurring in hypermedia systems. For this reason, we have implemented, according to the principles discussed above, a completely functional prototype on the basis of VODAK for the SEPIA hypermedia model, including the storage, application and presentation layer. It was out of question that the functional advantages actually proved to pay off, e.g., modeling of a particular hypermedia model became extremely simply. But, first experiences show that although the system was not particularly tuned at all, the overhead for maintaining the consistency of the semantics that occurs, e.g., for checking integrity constraints, performing semantic concurrency control, maintaining additional structural information etc., is fully acceptable. The prototypes' performance allows, for example, interactive usage of the DBMS for document browsing and editing. For us this was a very encouraging result, that clearly shows that the concepts presented in this paper can really carry on in practice.

The paper is organized as follows. In section 2 we describe the requirements for the design of the VODAK hypermedia engine, the VODAK modeling language and give an overview on our design approach. Section 3 is concerned with the modeling of semantic relationships and their combination to basic application-independent hypermedia modeling primitives. Furthermore, we explain how this data model can be tailored to application-specific need. In section 4, related work is reviewed and classified. Section 5 gives a brief summary, and further research objectives are identified.

2 DESIGNING THE HYPERMEDIA ENGINE

The SEPIA hypermedia authoring system (Streitz et al. 1989, Streitz et al. 1992) acted as a starting point for the development of the hypermedia engine. One requirement for the design of the hypermedia engine was that it should be possible to model the semantics of SEPIA's hypermedia objects. Therefore, we briefly describe in the next section the basic concepts of SEPIA and the logical components of its hypermedia model.

2.1 The SEPIA hypermedia authoring system and its data model

SEPIA supports the creation of hyperdocuments by providing the concept of activity spaces. Users create a hyperdocument by interacting with the four activity-space browsers dedicated to the tasks of content generation, planning, arguing, and writing the final hyperdocuments under a rhetorical perspective (content space, planning space, argumentation space, and rhetorical space respectively).

SEPIA's authoring-specific hypermedia model consists of the following basic hypermedia objects: atomic nodes, composite nodes, links, and atomic content objects. An **atomic node** can consist of several **atomic content objects**, e.g., text, audio and pictures objects. In general, **links** connect different nodes. Some kind of links can also point to or from other links. Links are allowed only between nodes and links that are members of the same composite or activity space.

Composite nodes are used as organizational means for clustering and nesting of related documents. Composite nodes may contain other nodes and links, forming a subgraph of the hypermedia network. They are also used within SEPIA to model the four authoring-specific activity spaces. Each space provides task-specific objects and operations to support the dedicated authoring activity. To model this SEPIA uses typed hypermedia objects*. Each activity space provides a dedicated subset of typed nodes and links. It is possible to post constraints over types of hypermedia objects to model task-specific semantics and maintain consistency of the hypermedia network. Let us look at an example to make this clearer. SEPIA's argumentation space can contain nodes of the types datum and claim and links of type supports. Support links are allowed to connect only a datum node with a claim node or a claim and with other claims.

To some extent, SEPIA supports type transformation of hypermedia objects. For example, if a node of type position is copied from the planning space into the argumentation space this operation includes a type conversion of the node to the type claim. Moreover, a node or link can have several instantiations within different composite objects in SEPIA. In the above example, SEPIA maintains a relationship between the position node and the claim node.

2.2 Requirements and design rationale for the hypermedia engine

Expressiveness of the hypermedia model

It is required that the data model of the VODAK hypermedia engine is at least as powerful as SEPIA's data model. Therefore, our hypermedia engine must

- support at least the basic hypermedia primitives described above,
- provide typed hypermedia objects, and allow to post constraints between them,
- support type transformation of objects according to some transformation rules, and
- be able to keep track of different roles of hypermedia objects within the hyperstructure.

In addition to SEPIA's hypermedia primitives it is valuable to take some other concepts into account which are proposed in the literature (Conklin 1987, Halasz 1988, Halasz 1991). The modular design of the hypermedia engine makes it easy to introduce new hypermedia primitives. Our hypermedia engine already provides some additional hypermedia concepts, e.g., it is allowed that links themselves contain information.

Flexibility of the hypermedia model

Another requirement for the design of hypermedia engine was that changes of the application-specific hypermedia model should be possible without recompiling the underlying database schema. This is useful because at the beginning of the development of the application-specific hypermedia model, the semantics is not exactly known. Hence, the development time can be reduced if new hypermedia types can be created and their constraints can be changed at run-time.

Extensibility of the hypermedia engine's database schema

The hypermedia engine should not only be able to capture authoring-specific semantics. For example, the VODAK hypermedia engine is used within the MuSE project (Lux 1993), too. MuSE aims at hypermedia support for the development process of complex technical products. For this kind of application, MuSE-specific types of hypermedia objects and constraints on them will be introduced, e.g., the four authoring spaces of SEPIA are replaced by a modeling and a validation space.

Integration of external storage systems and editing tools

It is not always possible to model the application-specific semantics of atomic content objects, e.g., video or text objects, within an OODBMS, and to build dedicated editing tools for them. In case of a proprietary text format it may be better to store text data as large bytestrings (BLOBS)

* The reader should be aware that these types are different from the object types used within the VODAK system to model the structure and behavior of objects and classes.

within an OODBMS. If we want to integrate videos, it may be better to store them outside the hypermedia engine within a specialized storage system, e.g., EOS (Biliris and Panagos 1994), and maintain only a reference to the video data. It should be possible to integrate editors within the hypermedia engine, because there already exist a variety of dedicated editing tools for viewing and manipulating different kinds of media. Therefore, in the design of our hypermedia engine generic facilities for integrating external storage services and editing tools had to be considered.

2.3 The VODAK data modeling language

Because there exist conceptual and terminological differences between different object-oriented database management systems, the relevant concepts of the used OODBMS VODAK (VODAK 1995) and its data modeling language (VML) are introduced briefly in this section.

As in other OODBMS, **objects** are used to represent material or immaterial entities, or abstract concepts. Objects are identified through unique **object identifiers** (OID). The structure (**properties**) and procedural behavior (**methods**) of objects is described through abstract data types which are called **object types**. VODAK distinguishes between object types and **classes** (dual model). Object types determine the structure and behavior of objects and hence are intentional, whereas class definitions describe class objects which act as containers for their instances (**extension** of a class). A class definition consists basically of two parts: the object type of the class object itself (`OWNTYPE`) and the object type of the class' instances (`INSTTYPE`). The initialization part allows initialization of the class' properties by calling methods of the class' own type.

```
CLASS <class name> [METACLASS <metaclass name>]
     OWNTYPE <object type of the class>
     INSTTYPE <object type of the class' instances>
     [INIT <initialization methods>]
END
```

In VODAK, classes are first-class objects, i.e., they can be treated like ordinary objects. Thus, it is possible to create classes and modify classes' properties at run-time. Because classes are treated as first-class objects, class objects are themselves instances of other classes, called **metaclasses**. Metaclasses are used in VODAK to describe the common structure and behavior of classes **and** their instances which may not be known at the time the metaclass is defined.

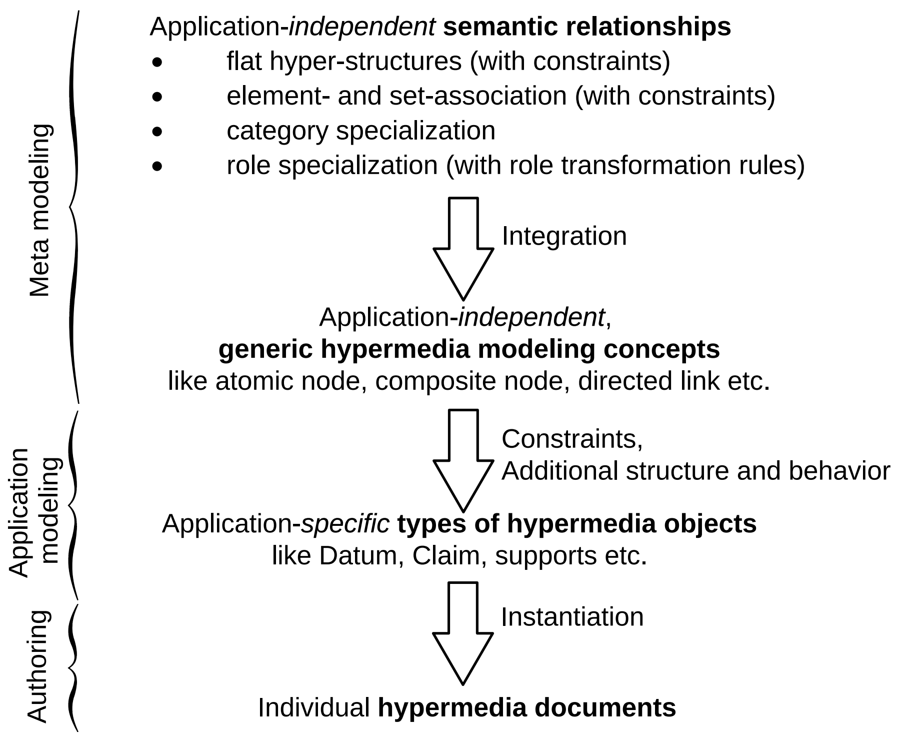
One reason to use metaclasses is to model semantic relationships between application classes. Many OODBMS offer hard-coded mechanisms to describe such relationships. But semantic relationships can have several dimensions. Thus, only a flexible mechanism like freely definable behavior and integrity constraints for metaclasses allows to model a great variety of dedicated semantic relationships, as needed for our hypermedia engine semantics. Metaclasses ensure the consistent usage of the object types defining the semantic relationships and enforce specific integrity constraints by declaring a class as an instance of a specific metaclass.

2.4 Overview of the design approach

In this subsection we give a brief overview of the design process of the hypermedia engine. More details are presented in section 3. The reader should get an intuitive understanding of our approach. Figure 1 indicates the design steps described below.

Definition of application-independent semantic relationships

First, we have analyzed what kind of semantic relationships (Klas et al. 1994) are necessary to model and implement a hypermedia engine that satisfies the requirements presented in section



Meta modeling
Application modeling
Authoring
Application-independent semantic relationships
flat hyper-structures (with constraints)
element- and set-association (with constraints)
category specialization
role specialization (with role transformation rules)
Integration
Application-independent, generic hypermedia modeling concepts like atomic node, composite node, directed link etc.
Constraints, Additional structure and behavior
Application-specific types of hypermedia objects like Datum, Claim, supports etc.
Instantiation
Individual hypermedia documents

Figure 1 Overview of the approach followed in the design of the hypermedia engine.

2.2. We identified the following four orthogonal, application-independent semantic relationships (which are described in section 3.1 in detail) to be sufficient:

- flat hyperstructures (e.g., different kinds of links in a hypermedia network);
- element- and set-association (e.g., for modeling of composite nodes);
- category specialization (e.g., for modeling of links that can behave like nodes);
- role specialization (e.g., for modeling type transformations).

These semantic relationships can be found in OODBMSs, but we extend them with additional constraint mechanisms and rule-based facilities. The semantic relationships are modeled by defining appropriate VODAK object types, that describe the structure and behavior for classes as well as for instances taking part in the semantic relationship.

Definition of application–independent, generic hypermedia modeling concepts

In the next step, the different semantic relationships are combined to describe the structure and behavior of single, application-independent hypermedia modeling concepts like atomic and composite nodes and different kinds of links. While the focus in the first step is to model the **intra**-semantic-relationship constraints, now the **inter**-semantic-relationship constraints have to be considered to ensure integrity of the hypermedia modeling concepts (see section 3.2).

To provide the application designer with a mechanism that allows to create dynamically application-specific instances of the generic modeling concepts, we utilize VODAK's metaclass mechanism. For each generic hypermedia modeling concept we define a metaclass describing the **application-independent semantics** of the hypermedia modeling concepts. These metaclasses constitute extensions of the data model that provide the necessary modeling primitives for the development of dedicated hypermedia applications.

The first two steps take place at the meta level and do not consider application-specific hypermedia semantics. Application-specific semantics is first considered at the application level by the definition of concrete, application-specific classes independent of the meta(class) level. This leads to a modular and reusable structure of our hypermedia data model.

Definition of application-specific hypermedia semantics

The generic hypermedia model can be tailored by an application designer to to the needs of a specific application domain, i.e., defining the application-specific hypermedia semantics. This is done by classifying objects into **application-specific** types of hypermedia objects and adding constraints with respect to the semantic relationships to these hypermedia types.

In VODAK the application-specific hypermedia types are defined by application classes which are instances of appropriate metaclasses describing the basic semantics for hypermedia objects. To specialize these semantics it is possible to add constraints to the application classes and to use additional object types in the definition of application classes. This extension of structural and behavioral semantics of hypermedia types would not be possible if types of hypermedia objects were simply represented as labels of objects.

The modular assembly concept in the design of the hypermedia engine enables a developer of an application-specific hypermedia model to reuse existing object type definitions and to refine and change the semantics of the model at run-time which results in reduced development time of the application-specific hypermedia model. The hypermedia engine ensures at the same time the generic and application-specific integrity constraints without programming efforts.

3 FROM APPLICATION-INDEPENDENT TO APPLICATION-SPECIFIC SEMANTICS

3.1 Modeling semantic relationships

In this section we present a (semi)formal description of the semantic relationships used to model the application-independent hypermedia primitives.

Flat hypermedia structures

A flat hypermedia structure can be viewed as a **graph** $G=(N_0,L_0)$ where the set of nodes N_0 corresponds to vertices of the graph and the set of binary links $L_0 \subseteq N_0 \times N_0$ corresponds to the edges that connect the nodes of G. If the graph contains only directed links we obtain a digraph. Because it is allowed that a hyper network contain both binary directed and bidirectional links, we get $G=(N_0, DL_0, BL_0)$ where DL_0 denotes the set of directed links and BL_0 denotes the set of of bidirectional links.

Because we allow also links on links, we have to extend this basic definition as follows: Let $N_{i+1} = N_i \cup DL_i \cup BL_i$, $DL_i \subseteq N_i \times N_i$, $BL_i \subseteq N_i \times N_i$. Using this, we can define $G = (N, DL, BL)$ where $N = \bigcup_{i \in \mathbb{N}} N_i$, $DL = \bigcup_{i \in \mathbb{N}} DL_i$, $BL = \bigcup_{i \in \mathbb{N}} BL_i$.[*] For flat hypermedia structures we consider the following application-independent constraints:

(i) Loops are not allowed in the graph, i.e. $(n, n) \notin DL \cup BL$.
(ii) Dangling links are not allowed by definition.

Additionally, our hypermedia engine allows the usage of typed links and nodes. Such a hyperstructure can be described as a **typed** graph $TG = (N, DL, BL)$. We introduce three total typing functions $type_N: N \rightarrow NT$, $type_{DL}: DL \rightarrow DLT$, $type_{BL}: BL \rightarrow BLT$ to obtain the types of nodes and links. The sets NT, DLT, BLT contain the possible types of the elements of N, DL, BL and are disjoint. DLT' and BLT' are subsets of DLT and BLT and denote the types of links that can behave

[*] Typically, BL_i and DL_i will be empty for values $i > i_0$, where i_0 is relatively small, e.g., in SEPIA $i_o = 1$.

like nodes. For the sake of simplicity we introduce a function $type_{HT}$ that combines the three functional relations $type_{N}$, $type_{DL}$ and $type_{BL}$. For typed graphs we have the following additional constraint:

(iii) Multiple links between vertices are allowed under the provision that the links have different types.

To model constraints on the connectivity of links and nodes we introduce a constraint function $cons_{HT}$: $DLT \cup BLT \rightarrow \wp((NT \cup DLT' \cup BLT') \times (NT \cup DLT' \cup BLT'))$ that determines if a link of a specific type is allowed to connect two typed objects $o_1, o_2 \in N$.* For a bidirectional link type $blt \in BLT$ it is required that $cons_{HT}(blt)$ is symmetric. With this formalism we are able to test if a link is allowed to connect some objects in the hypermedia structure:

(iv) A link $l = (o_1, o_2)$ can connect two objects o_1, o_2, if $(type_{HT}(o_1), type_{HT}(o_2)) \in cons_{HT}(type_{HT}(l))$.

Example 1: Let us consider an example obtained from SEPIA: The binary directed link type supports is only allowed to connect a node of type datum with a node of type claim or a claim with a claim node. Therefore, the result of the function $cons_{HT}$(supports) is equal to {(datum, claim), (claim , claim)}. ❑

Element- and Set-Association

To introduce the composition mechanism into the generic hypermedia model, we use the association concepts described below. Additionally, we introduce a constraint mechanism to describe application-specific constraints of the hypermedia model.

There are two types of associations, namely element- and set-association (Mattos 1988). Element-association introduces a set object to describe properties of a group of element objects. It suppresses the details of the element objects while emphasizing the properties of a group as a whole. Element-association establishes an **element-of** ($e \in S$) relationship between the element objects and the set objects, forming a 1-level hierarchy. Set-association introduces set object (superset) in order to describe properties of a group of set objects (subsets). Set-association establishes a **subset-of** ($S' \subseteq S''$) relationship between subsets and supersets. It may be applied recursively, building an n-level hierarchy. Of course, set-association requires element-association in order to instantiate basic non-empty set objects.

According to the Dexter hypermedia reference model (Halasz and Schwartz 1994), we decided to separate the nodes and links from their contents (Dexter within-component layer). The internal structure of the content objects are described separately from the hypermedia structure (see Figure 2). Thus, content objects can be viewed at the hypermedia abstraction level as atomic. These elementary content objects can be grouped together in a node, which establishes an element-of relationship between an atomic content object and nodes, which act as containers (sets) for their content objects. To model composite nodes we use the concept of set-association. Composite nodes can contain atomic nodes, links and recursively other composite objects. Therefore, we can view composite nodes as supersets which are composed out of other nodes and links (subsets) and establish a subset-of relationship. Of course, superset objects can also contain atomic content objects (element-of relationship). The only application-independent constraints for our concept of set-association is as follows:

(i) The set objects related by the subset-of relationship must correspond to a tree structure, i.e., set-association is acyclic and it is not allowed that the same set object is in a subset-of relationship with two different composite (set) objects.

* $\wp$ is used to denote the powerset.

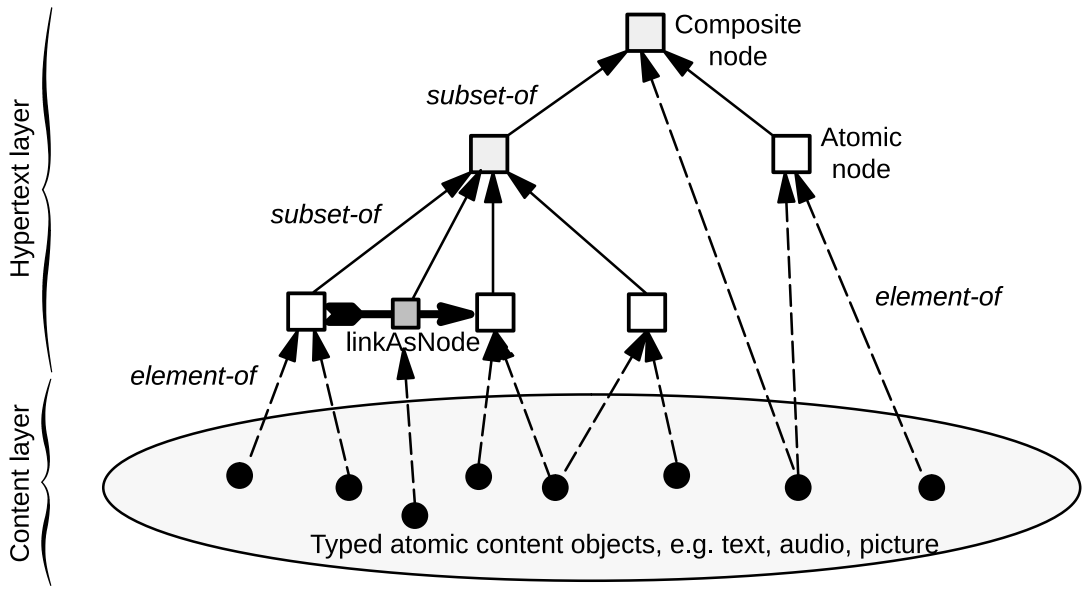


Figure 2: Example of the usage of element- and set-association in the hypermedia engine.

Because it is useful to share atomic content objects we do not obey this constraint in the case of element-association. Therefore, an atomic content object can be an element-of different atomic or composite nodes and links (see Figure 2).

In addition to this restriction, we combined a constraint mechanism with the concept of association for tailoring the hypermedia model to application-specific needs. As with nodes and links, elements and sets are typed. Therefore we introduce a typing function $type_{ASS}$: $EO \cup SO \rightarrow ET \cup ST$, $ET \cap ST = \oslash$ where EO and SO denote the set of atomic (element) and composite (set) objects, respectively. ET and ST represent the possible types.

(ii) Using this typing function we can introduce a constraint function to post constraints over the structure of association: $cons_{ASS}$: $ST \rightarrow \wp((ET \cup ST) \times \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\}))$.

These constraints are assigned to set objects. The first component describes which type of element and (sub)set objects can be included within the set object. The last two components are used to constrain the cardinality of occurrences of objects of a specific type within a set object.

Example 2: In SEPIA a composite node of type path has to contain exactly one start node and is allowed to contain an arbitrary number of content nodes and followedBy links and at most one atomic TextContent object which acts as an annotation of the path node. These constraints are expressed by $cons_{ASS}$(path) = {(start, 1, 1), (content, 0, ∞), (followedBy, 0, ∞), (TextContent, 0, 1)} where start, followedBy, content $\in ST$ and TextContent $\in ET$. This implies that if a path node instance is created, a start node has to be created automatically by the hypermedia engine. ❏

Category specialization

On of the most frequently used semantic relationships is specialization of classes. Classes can be declared as specializations (subclasses) of other classes, such that properties and methods of the superclass are inherited to their subclasses. In this sense specialization is similar to subtyping. The conceptual difference between subtyping and class specialization is that subtyping is a notion related to type definition and hence intentional. Class specialization is an extensional notion, where classes are used as containers for sets of objects and the specialization relationship leads additionally to extension inclusion between subclasses and superclasses.

We denote that class S is a **specialization-of** class T by $S \prec T$. We assume the following consistency constraints on the classes as well as on the instances of these classes, that participate in the specialization relationship:

(i) The relation $\prec$ must be a partial order.
(ii) Additionally we allow for a subclass only one direct superclass. Thus, we obtain a tree of classes (single inheritance).
(iii) Let ext(S) denote the extension of class S. The following properties regarding the extension must hold: If $S \prec T$ then $\text{ext}(S) \subseteq \text{ext}(T)$. This implies that $\bigcup_{S \prec T} \text{ext}(S) \subseteq \text{ext}(T)$ holds.
(iv) Furthermore, it is assumed that the extensions of two classes S, S' which are not in a specialization relationship are disjoint, i.e., $\neg(S \prec S' \lor S' \prec S)$ implies $\text{ext}(S) \cap \text{ext}(S') = \oslash$.

Klas et al. call this kind of semantic relationship **category specialization** (Klas et al. 1994). Property (iv) can be rephrased: Let *MSC*(o) denote a most special class of an object o, i.e., a class T with $o \in \text{ext}(T)$ and for all classes $S \prec T$: $o \notin \text{ext}(S)$. An important property of category specialization is that *MSC*(o) is unique: for all S with $MSC(o) \prec S$: $o \in \text{ext}(S)$ and for all T with $\neg(MSC(o) \prec T)$: $o \notin \text{ext}(T)$. We also use another kind of specialization called role specialization in the hypermedia engine. Role specialization does not require that *MSC*(o) is unique. This role-of relationship is discussed in the next section.

Example 3: Category specialization is used in various ways within our hypermedia engine. E.g., in some applications, certain links can behave almost like nodes, as they have a content and they may be referred to by other links. In SEPIA, a support link can be referred to by an explain link and has textual content. To avoid the introduction of additional concepts for this kind of links, we use category specialization to model this behavior. An application designer simply has to declare the link class supports as a specialization of a class like linkAsNode that is an instance of the metaclass AtomicNode and hence captures the node semantics of a link that can behave like a node. As a side effect, we get a common domain for all links of this kind. If a link class has no node semantics, no category specialization is used. ❑

Role specialization

As mentioned in the requirements section, nodes and links can occur in SEPIA as different types in different (types of) composites. Furthermore, it should be possible to transform typed nodes and links into other types. Such a situation occurs if a position node created in SEPIA's PlanningSpace is copied into the ArgumentationSpace. This results in the creation of a node of type claim within the ArgumentationSpace. Moreover, there is a relationship between the two nodes in the different spaces (see Figure 3). Some properties of these two nodes are shared, e.g., their name, but there exist also some properties that differ, e.g., the coordinates of the nodes in the authoring spaces or the links that point to or from the nodes.

To model this we use **role specialization** (Klas et al. 1994). The role-specialization relationship need not fulfill property (iv) of category specialization. The general object contains the shared properties, whereas the different roles contain the non-shared properties. In addition to that, the role objects can have different structure and behavior, defined by their own object types. Between the different role objects and their common general object a **role-of** relationship is established.

To specify the possible type transformation of hypermedia objects we introduce a function $trans_R: R \rightarrow \wp(R)$ where R is the set of classes which can appear as a role of a general object. This function describes all possible transformations of an object from one class to another. We assume that the object's state does not change during the transformation. However, it would be

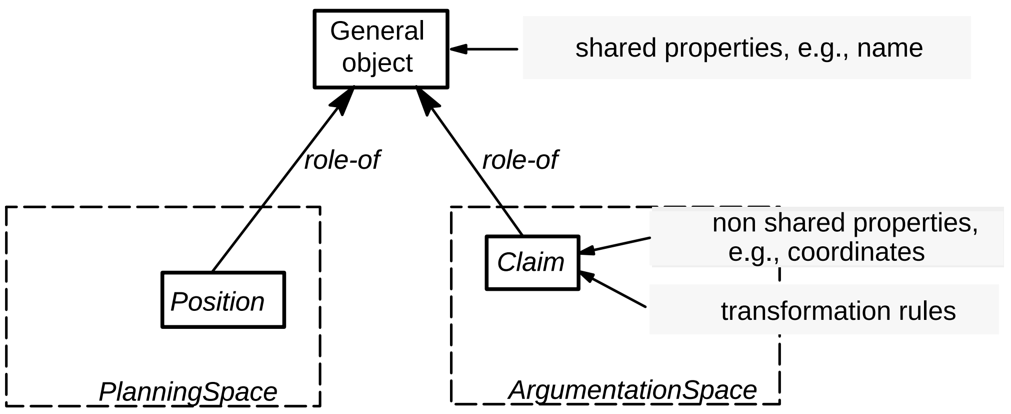


Figure 3 Example of the usage of role specialization.

possible to adopt concepts from object migration (Li and Dong 1994) to introduce more flexibility into role changes.

Example 4: For example, in SEPIA we have *trans*(position) = {claim, neutralNode} which denotes that a position node can be converted to a claim or a neutralNode. ❑

3.2 Combining semantic relationships to generic, application-independent hypermedia modeling primitives

In the last section, we discussed the modeling of semantic relationships and intra-semantic-relationship constraints. As mentioned in section 2.4, these semantic relationships are combined to several generic hypermedia modeling concepts, e.g., atomic nodes or composite nodes.

When combining semantic relationships, additionally the **inter**-semantic-relationship constraints have to be considered. It is not enough to combine the definitions of semantic relationship given by object types via subtyping. The semantic relationship must be integrated in a meaningful way to ensure the integrity of the hypermedia network.

Considering the different typing functions used in the hypermedia structure, association and role specialization relationships it is obvious that they have to share the same types for their constraint functions, such that each object in our hypermedia model has exactly one type.

Example 5: Figure 4 shows an example of how the semantic relationships are combined within the implementation in VODAK. A directed binary link class object type inherits the basic link semantics from DirectedBinaryLink_ClassType, the set semantics of element-association from SetAssoc_ClassType and other object types, e.g., for modeling presentation information. Nodes and links in SEPIA are always enclosed in a composite and links connect only objects within the same composite. Therefore, we first have to check if the link class (which represents the hypermedia type) is allowed in the composite, if the source and destination objects are contained in the same composite and then check if the link is allowed to connect the given objects. If all of these constraints are fulfilled we can create the link object (including its connection to the source and destination objects) and finally add the link to the composite object. ❑

In addition to the above mentioned combination of semantic relationships we had to provide additional functionality that is only meaningful with the combination of particular semantic relationships. For example, type transformations of nodes and links depend on the target composite object where the object should appear. Therefore, the type transformation function used in the role specialization relationship described above has to be extended to a function *trans*: $(NT \cup DLT \cup BLT) \times ST \times ST \rightarrow NT \cup DLT \cup BLT$, when combining role specialization

```
OBJECTTYPE Combined_DirectedBinaryLink_ClassType
    SUBTYPEOF DirectedBinaryLink_ClassType, SetAssoc_ClassType, CatSpec_ClassType,
              RoleSpec_ClassType, Presentation_ClassType, . . . // semantic relationships and other object type
    IMPLEMENTATION . . .
        createLink(from: OID, to: OID, inComposite: OID, ...): OID
            { . . .
            IF ((inComposite->canContain(SELF) AND // test for set-association constraints
                (to->isContainedIn(inComposite)) AND // test if dest. node is contained in same composit
                (from->isContainedIn(inComposite)) AND // dto. for source node
                (SELF->canConnect(from->class(), to->class())) // test for linking constraints
            THEN {newLink := SELF->createLink(from, to); // create the link  between 'from' and 'to'
                  inComposite->addSubsetObject(newLink)} // AND add it into the composite object
            ELSE {RETURN NULL}; // constraint violation – no link is created
            . . .} . . .
END
```

Figure 4 Example of a combination of semantic relationships.

and set-association. The arguments of *trans* are the class of the hypermedia object to be transformed and the classes of the source composite object and the target composite object. The result of this function is the identifier of a single class. A restriction is that a node cannot be transformed to a link and vice versa.

This type transformation mechanism together with role specialization enables us to transform objects within the hypermedia network in a flexible way. Of course, before doing the actual type transformation, the hypermedia and association constraints have to be checked.

For each hypermedia modeling concept we provided two combined object types: one object type defines the semantics of the individual objects, the other describes the class' structure and behavior. These two object types are used in the metaclass definition.

These metaclasses describe the **application-independent semantics** of the hypermedia modeling concepts and extend the OODBMS VODAK to a hypermedia engine. They provide the necessary modeling primitives for the development of dedicated hypermedia applications and ensure the consistent usage of the semantic relationship. Moreover, they allow the dynamic creation of classes at run time. The current implementation of the hypermedia engine supports the following metaclasses:

- **Nodes:** AtomicNode, CompositeNode, and Node;
- **Links:** DirectedBinaryLink, BidirectionalBinaryLink;
- **Composite contents:** CompositeContent and Organizer;
- **Atomic contents**: AtomicContent, BytestringAtomicContent, and ExternalReferenceAtomicContent.

The Node and CompositeContent metaclasses together allow the modelling of objects that can behave like atomic or composite hypertext objects. Organizer classes are used to organize complete hyperdocuments in a directory-like way. The AtomicContent metaclass is used to implement classes that model different kinds and format of media within VODAK. The BytestringAtomicContent metaclass and ExternalReferenceAtomicContent metaclass support storage of multimedia data (e.g., text, pictures, audio, video) as BLOBs in VODAK or in external storage systems. These metaclasses include also generic mechanisms for the integration of external application programs.

3.3 Tailoring the hypermedia model to application-specific needs

The metaclasses shown above can be used by a schema designer to tailor the hypermedia model to the needs of a specific application domain, thus, defining the **application-specific hypermedia semantics**. Application classes simply are declared as instances of an appropriate metaclass to provide them with the intended behavior. As mentioned before, different types of hypermedia objects are modeled as different classes within the hypermedia engine.

Tailoring the generic hypermedia model to the application-specific semantics can be done by asserting constraints regarding the semantic relationships to the application classes. The constraints can be changed at run-time because constraint insertion and deletion is done by ordinary method calls. Moreover, it is possible to create new classes as instances of an existing metaclass at run-time because classes are treated as first class objects in VODAK.

```
CLASS supports METACLASS DirectedBinaryLink  // declaration of class as an instance of the metaclass
      INIT SELF->subclassOf(linkAsNode);
           SELF->addLinkConstraints([datum, claim],[claim, claim]);
           SELF->addElementConstraints([TextContent,0,1],[AudioContent, 0, 1], ...);
           SELF->addTransformationRule(ArgumentationSpace, PlanningSpace, neutralLink)
           . . .
END
```

Figure 5 Example of tailoring an application-specific class by adding constraints.

Example 6: An example is shown in Figure 5.First, the class supports is declared as an instance of the metaclass BinaryDirectedLink which describes the general behavior of this kind of links. Then it is declared as a subclass-of the class linkAsNode which is an instance of the metaclass AtomicNode and therefore inherits the node-like behavior to link instances of class supports. Moreover, application-specific constraints on the connectivity and the possible content of the link are asserted to the class by means of an `INIT` clause and a transformation rule is added which express that if a link of class supports is copied from an ArgumentationSpace to a PlanningSpace object, it is automatically converted to an object of class neutralLink.❑

In this example, it was not necessary to enrich the definition of the application-specific link class supports with additional functionality. It is possible to provide additional functionality for an application class by adding object types in the class definition. This semantic enrichment for dedicated hypermedia types would not have been possible if types of hypermedia objects were simply represented as labels (strings) of objects. This was the reason why we decided to model different types of hypermedia objects as different classes.

```
CLASS AudioContent METACLASS AtomicContent // declaration of class as an instance of the metaclass AtomicConten
      OWNTYPE VODAK_Audio_ClassType      // adding  additional object types that model audio class' and
      INSTTYPE VODAK_Audio_Type          // audio instance's structure and behavior
END
```

Figure 6 Example of adding application-specific structure and behavior to a class.

Example 7: The class AudioContent (see Figure 6) is defined as an instance of the metaclass AtomicContent which describes the general semantics of content objects in our hypermedia engine. Additionally, an `OWNTYPE` and an `INSTTYPE` are defined for the class AudioContent and its instances, which provide the properties and methods for storing and manipulating audio objects within VODAK. Therefore, the structure and behavior of the class AudioContent and its instances is composed out of the generic object types inherited by the metaclass **and** the application-specific object types. There are no constraints assigned to the class AudioContent because constraints regarding the element-of relationships are assigned to the set objects.❑

4 SYSTEM ARCHITECTURE AND EVALUATION

In this section we describe the system architecture of the implemented VODAK hypermedia engine. Moreover, we present some results of the evaluation we have done on a SEPIA tailored hypermedia engine.

4.1 System architecture of the hypermedia engine

The overall system architecture is shown in Figure 7. The core of the hypermedia engine consists of the object-oriented database management system VODAK. VODAK contains the metaclasses that implement the basic hypermedia engine functionality, e.g., the checking of generic and application-specific integrity constraints and consistency-preserving operations for the manipulation of hypermedia structures. Moreover, VODAK stores and manages the application-specific hypermedia models described by application classes, their application-specific constraints, and additional object types.

External storage systems, e.g., EOS for storing large videos, can be connected to VODAK by instantiating the ExternalReferenceAtomicContent metaclass. Accesses and manipulations to these external storage systems are managed by the VODAK hypermedia engine, and, hence, are transparent for applications running on top of it. The invocation of external application programs is handled by the VODAK hypermedia engine, too.

Applications of the VODAK hypermedia engine are implemented using the C++ based VODAK client interface. The VODAK clients may run on arbitrary nodes in the network and communicate via the VODAK server interface with the VODAK hypermedia engine. Basically, the VODAK client interface can be considered as a remote API to the VODAK OODBMS; it offers VODAK data types (VML) and the VODAK query language (VQL) that can be used to build applications programs like graphical user interfaces. Moreover, it offers support for visualization and manipulation of multi–media data stored within VODAK. Client applications communicate with VODAK via a generic interface which consists of the following functions:

- getting the OID of a class by sending the class' name;
- begin, commit and abort of a VODAK top-level transaction;
- submitting arbitrary method calls to VODAK and transferring back the results;
- submitting declarative queries to VODAK and transferring back the results.

The complex, consistency-preserving operations offered by the VODAK hypermedia engine are invoked by the application using the method call interface. Each of those operations is executed as a single VODAK top-level transaction by default. Using the transactional commands offered by the interface, an application programmer can build new complex transactions, e.g., macros, consisting of several consistency-preserving operations. Utilizing VODAK's open nested transaction model (Muth et al. 1992, Muth et al. 1993) and the commutativity predicates defined for the hypermedia engine's operations, each operation of an application-defined transaction can be executed as a subtransaction, increasing the degree of concurrency without loss of ACID properties. Moreover, an application programmer can use declarative VQL queries (including the hypermedia engine's operations), thus, enabling set-oriented access to hypermedia documents.

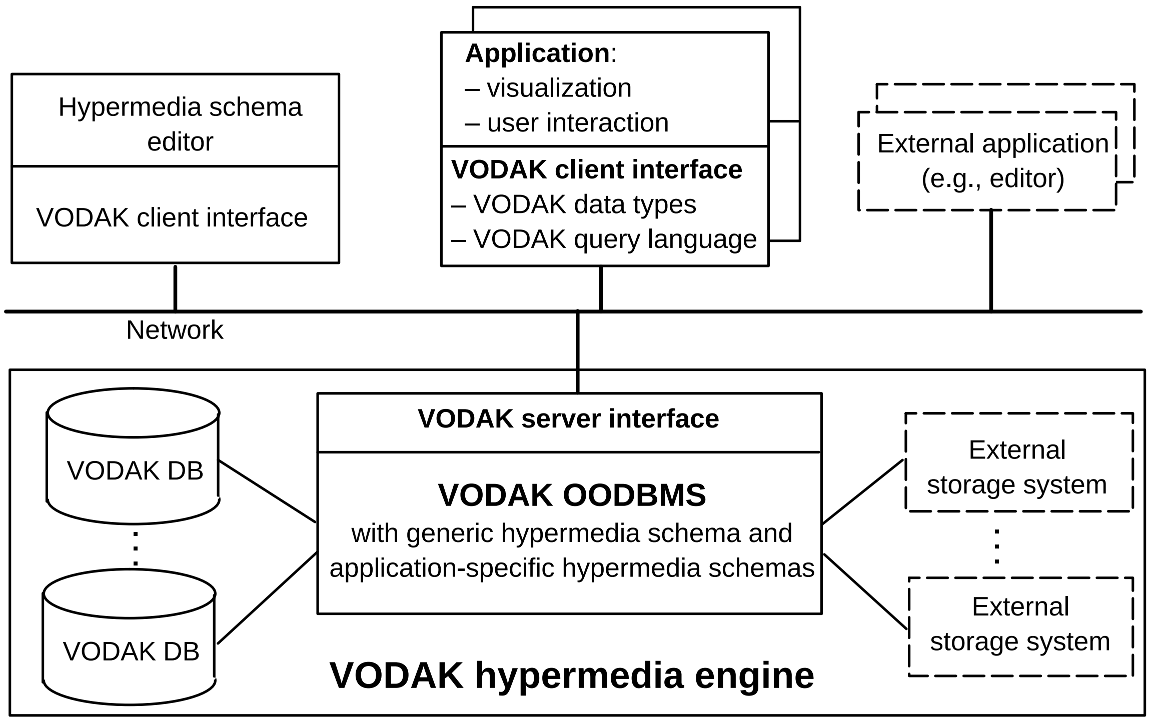


Figure 7 System architecture of the VODAK hypermedia engine.

4.2 Performance and experiences

As shown in the previous sections, our approach of modular design enables the efficient development of dedicated hypermedia database schemas. In this section we present some experiences and results obtained by an evaluation of the implemented hypermedia engine.

The results of the evaluation are based on a hypermedia engine that is tailored to the application-specific needs of the SEPIA hypermedia authoring system. The database schema consist of 10 metaclasses and 56 application-specific classes. Modeling the SEPIA-specific semantics by the 56 application-specific classes on top of our generic hypermedia schema took only **one** day. And by previous arguments this model ensures the maintenance of the generic and application-specific integrity constraints of the hypermedia model within the database hypermedia engine.

In the case of SEPIA, only classes corresponding to the available hypermedia types in SEPIA had to be defined and initialized with SEPIA-specific constraints. The structure and behavior of atomic content objects that represent audio and pictures was modeled by reusing existing VODAK object types for storage and manipulation of these kinds of media.

Of course, we have to raise the question whether this efficiency and flexibility in design and safety in execution can be compliant with reasonable run-time performance. For this reason we performed some preliminary experiments with our DBMS-based hypermedia engine. The results of this evaluation are shown in Figure 8. Performance measurement was done on a SUN Sparc 10 workstation running the VODAK hypermedia engine and a single client application. All of the operations shown in Figure 8 are covered by transaction management.

These results show that an adequate performance for interactive hypermedia applications that are built upon our hypermedia engine is achieved although all the manipulations on the hypermedia structure are done within the hypermedia engine. The response times are such that interactive editing operations, like createLink or changeName, are possible. Additionally, one has to consider that the numbers where obtained from the first fully functional prototype of the hypermedia

Method	Submitting method to VODAK	Executing method within VODAK	Transferring results to clients	Average time in ms
getAllClassDefinitions	0 % (8 ms)	81.8 % (18430 ms)	18.2 % (4098 ms)	22 536 ms
createNode	10.5 % (41 ms)	83.3 % (325 ms)	6.2 % (24 ms)	390 ms
createLink (link with node behavior)	7.8 % (47 ms)	88.4 % (533 ms)	3.8 % (23 ms)	603 ms
createLink (constraint violation – no link created)	44.2 % (46 ms)	34.6 % (36 ms)	21.2 % (22 ms)	104 ms
changeName	25 % (12 ms)	54.2 % (26 ms)	20.8 % (10 ms)	48 ms
copyObjectsToClipboard (5 objects)	13.8 % (30 ms)	79.4 % (173 ms)	6.9 % (15 ms)	218 ms
pasteObjectsFromClipboard (type transformation of 5 objects)	0.4 % (9 ms)	92.4 % (2073 ms)	7.2 % (162 ms)	2244 ms
openComposite (15 objects contained in composite)	0.8 % (11 ms)	78.9 % (1057 ms)	20.3 % (272 ms)	1340 ms
createAtomicContent	5.2 % (19 ms)	88.4 % (320 ms)	6.4 % (23 ms)	362 ms

Figure 8 Performance of selected methods offered by the hypermedia engine.

engine without any particular optimizations. Checking constraint within the client applications can increase performance, too. This is possible since the hypermedia engine offers a method getAllClassDefinitions to retrieve the classes' constraints and other information. Nevertheless, consistency of the hypermedia network is always ensured, because all semantics and constraints of the hypermedia objects are captured by the VODAK hypermedia engine.

5 RELATED WORK

According to the HAM model (Campell and Goodman 1988), most hypermedia systems can be divided into three functional layers: a storage layer providing persistence to the system, an application layer providing the functionality of the system and a presentation layer enabling the users to interact with the system. The storage layer in most systems is geared towards the specific needs of the application and presentation layer of the particular hypermedia system and usually provides a fixed hypermedia data model.

In recent years, several general purpose hypermedia "database" systems (often called **hyperbase** systems) were developed, e.g., HAM (Campell and Goodman 1988), GMD-IPSI's HyperBase (Schütt and Streitz 1990), the Danish HyperBase (Wiil and Østerbye 1990), DeVise/DHM (Grønbaek and Trigg 1994). These systems are either based on a file system or built on top of a (relational or object-oriented) database management system. The HAM (Hypermedia Abstract Machine) used in Neptune provides generic hypermedia system functionality like create, delete, get and update of hypermedia objects. It uses a file system for persistent storage of the hypermedia objects. GMD-IPSI's HyperBase was built as a general interface between the application layer and the storage layer on top of the RDBMS Sybase. It is based on a fixed, application-independent hypermedia data model. As the HAM, HyperBase provides a fixed set of generic operations, e.g., create, delete, copy, retrieve and modify of hypermedia components.

Our approach differs from the above mentioned systems in the sense that we built our hypermedia engine not upon a storage system but extended an OODBMS with functionality for the management of hypermedia structures, combining the traditional advantages of DBMS like transaction management and query facilities with the advantages of object-oriented data modelling. This was done by designing an appropriate set of metaclasses. The extended OODBMS is not only concerned with the persistent storage of hypermedia objects, it also captures the structure **and** behavior of the hypermedia objects related to the application layer. Thus, it can be categorized as a hypermedia engine, not only as a (passive) hyperbase system. Capturing all of the semantics within our hypermedia engine enables us to maintain application-independent and application-specific integrity constraints within the OODBMS whereas the other systems are only able to manage these integrity constraints on top of the storage system. Furthermore, we are able to provide a set of semantically rich and consistency-preserving operations that can be used by the application systems running on top of the hypermedia engine.

All of the systems mentioned above, provide a fixed hypermedia data model. An exception is Hyperform (Wiil and Leggett 1992) which implements basic hyperbase services (a small class library) that can be tailored via subtyping to provide specialized hyperbase support. The Hyperform server is based on an internal computational engine and an object-oriented language written in C and an extension of Scheme. Unfortunately, it is not clear from the literature if Hyperform provides true database functionality. Our hypermedia engine supports extensibility of the hypermedia model, too. We are able to assert application-specific constraints to classes and create classes as instances of given metaclasses at run-time. Moreover, an application designer can enrich the semantics of application classes by adding additional object types to their class definition. If some new hypermedia primitives are needed, additional metaclasses can be designed. This does not require great effort because the existing semantic relationships can be reused.

In addition to the HAM model, the DEXTER hypermedia reference model (Halasz and Schwartz 1994) proposes the separation of the hypermedia structure from the node contents (within-component layer). Since the range of possible types is large (text, image, sound) and hard to model in a generic way, the within-component layer is not part of the model per se. DeVise/DHM is an example of an object-oriented hypermedia framework for the Dexter concepts. We have adapted this separation but the data modeling facilities of VODAK (together with VODAK's multimedia extensions) are powerful enough to model the internal structure and behavior of this kind of media. The modular design of the hypermedia engine makes it easy to enrich the semantics of the content objects with additional functionality by plugging in object types that model the semantics of within-component layer objects. We have used this extensibility to model audio and picture content within our hypermedia engine. Nevertheless, it is possible to store the content of nodes and links as binary large objects within the hypermedia engine or as references to an external storage systems leaving the interpretation of the data to the applications.

Another research direction focuses on the integration of existing hypermedia systems with current database technology instead of building general purpose hyperbase systems. E.g., HyperPath/O2 (Amann et al. 1993) and MultiCard/O2 are built on top the OODBMS O2. HyperPath/O2 and MultiCard/O2 utilize O2 only as the persistent depository of their so called hypermedia basic classes. O2 provides a minimal interface (create, load, save, delete) to the persistence module of the application layer. As opposed to our hypermedia engine, the hypermedia management is not implemented within O2 but is part of the applications. No additional semantics to the above read/write operations is captured by the database management system.

MultiCard/O2, HyperPath/O2 and DeVise/DHM offer weakly-typed nodes and links, i.e., the nodes and links can have an arbitrary list of properties. Strong typing, i.e., the assertion of additional functionality to typed hypermedia objects is not possible. Furthermore, up to our knowledge all of these systems provide no mechanism to post constraints to tailor the hypermedia model to the application-domain-specific needs. In our approach, different types of hypermedia objects are represented as classes which are themselves instances of metaclasses that describe the general semantics of classes and their instances. Therefore, it is possible to assert additional structure and behavior to types (classes) of nodes and links.

6 CONCLUSION AND FUTURE WORK

In this paper we presented the design of a hypermedia engine that is implemented within the object-oriented database management system VODAK. Our intention was not to provide a new hypermedia data model, but to develop an open, tailorable hypermedia engine combining the advantages of an OODBMS, like multi-user access, transaction management and declarative query access with advanced object-oriented data modeling facilities. The hypermedia engine as described in this paper is fully implemented.

In the design of the hypermedia engine we follow a modular assembly concept. Several dedicated semantic relationships are implemented which are combined in a meaningful way to describe the structure and behavior of hypermedia modeling primitives. Afterwards, several well-defined, application-independent metaclasses were built to ensure the consistent use of the combination of the semantic relationships. A designer of an application-specific hypermedia schema only has to declare his application classes as instances of appropriate metaclasses. Furthermore, he can tailor the data model to application-specific needs by adding constraints to the classes and additional functionality if needed.

The VODAK hypermedia engine fulfills the requirements mentioned in section 2.2 and the requirements for hypermedia storage mechanisms stated in (Lange et al. 1992) (openness, sharing, integrity, multimedia, querying, extensibility, versioning), except that we have not integrated VODAK’s versioning mechanism yet. Preliminary results show that an adequate performance for interactive use is achieved although we use a flexible design approach and the whole semantics of the model is captured by the hypermedia engine, i.e., all the manipulations and constraint checking on the hypermedia structure are done within VODAK.

The design of additional hypermedia modeling primitives is demand driven. If some applications need additional modeling primitives, new metaclasses will be implemented. For example, if the hypermedia engine will be used within the meeting-room system DOLPHIN (Streitz et al. 1994) a metaclass for handling persistent scribbles has to be developed.

Other future extension will address the integration of the VODAK hypermedia engine with the SGML (ISO-8879 1992) database schema developed at our institute (Aberer et al. 1994), the mapping to and integration of the linking architectural forms of the HyTime ISO standard (ISO/IEC-10744 1992), and the tight integration of declarative VQL queries (Aberer and Fischer 1995) with the hypermedia engine.

ACKNOWLEDGEMENTS

We would like to thank Ajit Bapat for his contribution in implementing the presentation layer of the prototype and many valuable discussions for analyzing the SEPIA model. We also like to thank Gisela Fischer for her valuable comments on an earlier version of this paper.

REFERENCES

K. Aberer, K. Böhm, C. Hüser (1994) The Prospect of Publishing Using Advanced Database Concepts, *Proceedings of the Conference on Electronic Publishing, Document Manipulation and Typography (EP) ’94, Darmstadt, Germany,* John Wiley & Sons.

K. Aberer and G. Fischer (1995) Semantic Query Optimization for Methods in Object-Oriented Database Systems. *Proceedings of the 11th IEEE Conference on Data Engineering (ICDE '95), Taipei, Taiwan.*

B. Amann, V. Christophides and M. Scholl (1993) HyperPath/O2: Integrating Hypermedia Systems with Object-Oriented Database Systems. *Proceedings of the 4th International Conference on Data and Expert Systems Applications (DEXA '93), Prague, Czech Rebublic,* 709–720.

A. Biliris and E. Pangos (1994) EOS User's Guide, Release 2.2, *Technical report AT&T Bell Laboratories*.

B. Campell and J.M. Goodman (1988) HAM: A general purpose Hypertext Abstract Machine, *Communications of the ACM*, Vol. 31, No. 7, 856–861.

J. Conklin (1987) Hypertext: An Introduction and Survey. *D. Marca and G. Rock [Eds.]: Groupware: Software for Computer Supported Cooperative Work*, IEEE Computer Society Press, Los Alamos, CA, 236–260.

K. Grønbaek and R.H. Trigg (1994) Design Issues for a Dexter-Based Hypermedia System, *Communications of the ACM*, Vol.37, No. 2, 40–49.

F.G. Halasz (1988) Reflections on Notecards: Seven Issues for the Next Generation Of Hypermedia Systems, *Communications of the ACM,* Vol. 31, No. 7, 836–852.

F.G. Halasz (1991) Seven Issues: Revisited, *Hypertext '91, Third ACM Conference on Hypertext, San Antonio, Texas.*

F.G. Halasz and M. Schwartz (1994) The Dexter Hypertext Reference Model, *Communications of the ACM,* Vol. 37, No. 2, 29–39.

ISO 8879–1986 (E) (1992) Information Processing – Text and Office Systems – Standardized Generalized Markup Language (SGML), *International Organization for Standardization.*

ISO/IEC 10744–1992 (E) (1992) Information Technology – Hypermedia/Time-based Structuring Language (HyTime), *International Organization for Standardization.*

W. Klas, K. Aberer and E.J. Neuhold (1994) Object-Oriented Modelling for Hypermedia Systems Using the VODAK Model Language. *A. Dogac, T. Özsu and A.Biliris [Eds.]: Advances in Object-Oriented Database Systems, NATO ASI Series F,* Springer Verlag Berlin, 389–433.

D.B. Lange, K. Østerbye and H. Schütt (1992) Hypermedia Storage, *Report R 92-2002,* Dept. of Math. and Comp. Sci., Aalborg University, Denmark.

D.B. Lange (1993) Object-Oriented Hypermodeling of Hypertext Supported Information Systems. *Proceedings of the 26th Hawaii International Conference on System Sciences,* Vol. 3, 380–389.

Q. Li and G. Dong (1994) A framework for object migration in object-oriented databases, *Data & Knowledge Engineering* 13, 221–242.

G. Lux (1993) MuSE – A Technical Systems Engineering Environment, *Technical Report,* Department of Computer Science, Technical University of Darmstadt.

N.M. Mattos (1988) Abstraction Concepts: The Basis for Data and Knowledge Modelling. *Proceedings of the 7th International Conference on Entity-Relationship Approach, Rome, Italy,* 331–350.

P. Muth, T.C. Rakow, W. Klas and E. J. Neuhold (1992) A Transaction Model for an Open Publication Environment. *A. K. Elmagarmid [Ed.]: Database Transaction Models for Advanced Applications,* Morgan Kaufman, San Mateo, California, 169–218.

P. Muth, T.C. Rakow, G. Weikum, P. Brössler and C. Hasse (1993) Semantic Concurrency Control in Object-Oriented Database Systems. *Proceedings of the 9th IEEE Conference of Data Engineering, Vienna (ICDE '93), Austria* 232–242.

H. Schütt and N.A. Streitz (1990) HyperBase: A Hypermedia Engine Based on a Relational Database Management System. *Proceedings of the European Conference on Hypertext (ECHT '90), Versaille, France,* 95–108.

N.A. Streitz, J. Hannemann and M. Thüring (1989) From Ideas and Arguments to Hyperdocuments: Travelling through Activity Spaces, *2nd ACM Conference on Hypertext (Hypertext '89), Pittsburgh, P.A.,* 343–364.

N.A. Streitz, J.M. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schütt and M. Thüring (1992) SEPIA – A Cooperative Hypermedia Authoring System. *Proceedings of the ACM Conference on Hypertext (ECHT '92), Milano, Italy,* 11–22.

N.A. Streitz, J. Geißler, J.M. Haake and J. Hol (1994) DOLPHIN: Integrated Meeting Support across LiveBoards, Local and Remote Desktop Environments. *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '94), Chapel Hill, N.C.,* 345–358.

U.K. Wiil and J.J. Leggett (1992) Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems. *Proceedings of the ACM Conference on Hypertext (ECHT' 92), Milano, Italy* 251–261.

U.K. Wiil and K. Østerbye (1990) Experiences with HyperBase – A multi-user back-end for hypertext applications with emphasis on collaboration support. *Technical Report R 90-38,* CS Dept., University of Aalborg, Denmark.

VODAK Manual (1995) Release 4.0, *Technical Report, Arbeitspapiere der GMD No. 910,* GMD, Germany.

BIOGRAPHY

Jürgen Wäsch is a member of the database research group VODAK at the Integrated Publication and Information Systems Institute of the German National Research Center for Information Technology (GMD-IPSI), Darmstadt. He is also involved in European ESPRIT research projects. His research activities and interests include cooperative transaction management, global transaction management for ODMG-compliant multi-database systems, object-oriented database system support for cooperative hypermedia systems, and mobile information systems.

He received his diploma degree in computer science and economics in 1993 from the University of Kaiserslautern. After working at the University Hospital in Heidelberg he joined GMD-IPSI in November, 1993.

Dr. Karl Aberer is department manager of the database research group VODAK at the Integrated Publication and Information Systems Institute of the German National Research Center for Information Technology (GMD-IPSI), Darmstadt. He is conducting projects in hypermedia document modelling and bioinformatics. His research interests include object–oriented and multimedia database systems, data modelling, query processing, and foundations for database management systems.

He received his Ph.D. in mathematics in 1991 from the ETH Zürich where he was from 1987 to 1991 research assistant. From 1991 to 1992 he was postdoctoral fellow at the International Computer Science Institute (ICSI), Berkeley. In 1992 he joined GMD-IPSI.