HyperPATH/O2: Integrating Hypermedia Systems with Object-Oriented Database Systems*

B. Amann[1], V. Christophides[2] and M. Scholl[2]

[1] INRIA F-78153 Le Chesnay Cedex France
[2] Cedric/CNAM 292 rue St Martin 75141 Paris Cedex 03 France

Abstract. We describe an integration of the hypermedia system HyperPATH with the object-oriented DBMS (OODBMS) O_2. Providing persistence to a hypertext system was the first motivation of this work. More generally, we were interested in a better understanding of the connection of hypertext systems with OODBMS. One of our goals was to define an abstract interface between HyperPATH and a variety of OODBMS's. The solution adopted shows that opening HyperPATH to different OODBMS's is not possible without major rewriting of existing C++ code.

Keywords: hypertext, hypermedia, object oriented database, system integration, persistence, C++ code reusability

1 Introduction

This paper describes an integration of the hypermedia system HyperPATH (formerly called MultiCard) [12] with the object-oriented database management system (OODBMS) O_2 [7]. Providing persistence to a hypertext system was the first motivation of this work. More generally, we were interested in 1) gaining a better understanding of the connection of hypertext systems with object-oriented database systems and 2) providing hypertext systems with typed nodes and links [11] in order to introduce new query facilities [2]. The latter issue will not be addressed in this paper. Instead we shall concentrate on providing persistence to HyperPATH by connecting it to the OODBMS O_2. One challenge was to verify whether an integration between a significant piece of code written in C++ (HyperPATH) and the OODBMS O_2 could be implemented quickly and easily. In particular, it should not require major rewriting. Another issue was to check whether there exists a general solution which is independent of the OODBMS used for the integration. In parallel with our work, an integration of HyperPATH with the OODBMS ONTOS has been done by Bull [1].

According to the HAM model [4], hypertext systems can be divided into three distinct layers : a) a *storage layer* providing persistence to the hypermedia

* Work partially supported by the French *Programme de Recherche Coordonnée BD3*.
V. Christophides and M. Scholl are also affiliated with INRIA.
Email: <Bernd.Amann|Vassilis.Christophides|Michel.Scholl>@inria.fr

objects, such as nodes, links and anchors b) a *management layer* (Hypertext Abstract Machine) providing system functionalities, such as the creation, update, deletion and display of the hypermedia objects and c) a *presentation layer* providing a high-level interface for human-machine interaction. In a further step, the DEXTER model [10] proposes the separation of the hypertext structure (storage layer) from the node contents (within-component layer). In this approach, since the range of possible document types is large (text, image, sound, etc.) and hard to be modeled in a generic way, the within-component layer is not part of the hypertext model *per se*. This architecture as well as the growing demands on the storage layer to provide concurrency control, transaction management, object versions, distribution or even efficient query language facilities [9], leads to two main research directions.

The first investigates the integration of existing hypertext systems with current database systems (relational or object-oriented). Examples of such persistent hypertext systems are described in [8, 14, 5]. The second research direction attempts to create an open hypermedia database platform for a variety of connections with existing hypertext systems. Examples of such platforms can be found in [6, 13]. Recent evolution in this direction allows new objects and operations to be added at run-time [16]. The integration of HyperPATH with O_2 pertains to the first direction. To our knowledge this is the first attempt to provide a hypermedia system with a persistence module built on top of a commercial OODBMS. Current hypertext systems are generally closed systems that cannot be easily integrated with other programs and data. On the other hand, it is evident that the extensibility of hypertext systems is crucial for interoperability reasons, in particular for the exchange of hyperdocuments stored in various systems and applications. For example, with our integration of HyperPATH with O_2 we have the possibility to visualize the structure of a hypertext by using the O_2Graph application tool. In this direction we have also investigated the definition of a minimal interface between the management layer and the storage layer in order to implement the latter by a variety of OODBMS's.

O_2 is an object-oriented DBMS developed and distributed by O_2Technology. It combines the traditional advantages of DBMS's (persistence, secondary storage management, concurrency, recovery and ad hoc query facility) with those of object-oriented programming (encapsulation, extensibility, computational completeness, etc.). HyperPATH is a set of hypermedia software tools that allows the development of interactive hypermedia applications. It has been developed at Bull as a part of the Esprit project Multiworks. In HyperPATH, a *hypertext* (hypermedia application or hyperdocument) is a graph with nodes containing multimedia documents, hereafter referred as *documents*. Following the DEXTER model, documents are strictly separated from the hypertext structure.

Prior to our integration, the persistence of the HyperPATH hypermedia objects was provided by an *ad hoc* distributed system, called *mwdbms* (Multiworks Data Base Management System). The developers' objective was not to implement a complete database system but an open and flexible module that allows hypertexts to be stored and retrieved on different hosts. In order, then, to extend HyperPATH with full database functionalities (query facilities, security, recovery, etc.) as well as for a better understanding of the relationship between hypermedia systems and object-oriented database systems, we use the O_2 DBMS for implementing the persistence of HyperPATH objects.

Section 2 presents the architecture and the hypertext model of HyperPATH as well as its initial persistence mechanism. The C++ interface of O_2 which is used to connect both systems is described in Section 3. In Section 4 we present different levels of integration among which one was chosen for the integration of HyperPATH and O_2. Section 5 points out the main issues of an "open" integration of HyperPATH with OODBMS's while Section 6 describes the solution that was finally adopted.

2 HyperPATH

2.1 The HyperPATH Architecture

A general overview of the architecture of HyperPATH [12] is given in Figure 1. A hypermedia application can be constructed by using the following modules:

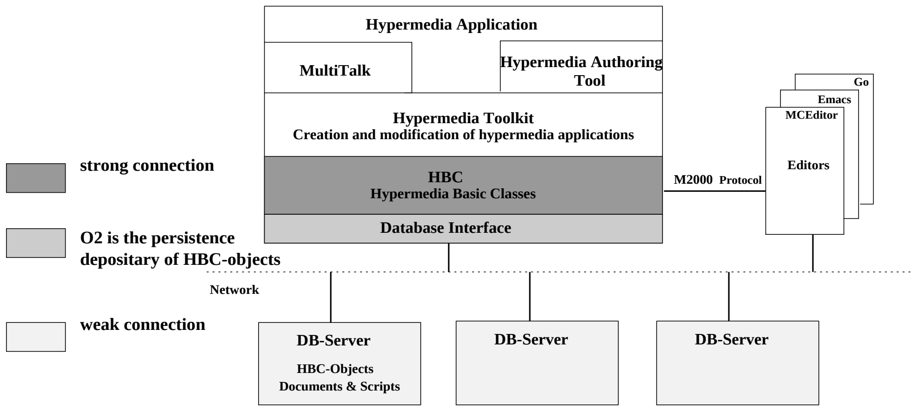

Fig. 1. HyperPATH Architecture

1. The *Hypermedia Authoring Tool* is a `Motif`[3] application built on top of the *Hypermedia Toolkit* (see below) and allows a) authors to create and modify hypertexts by using specific editors, and b) readers to browse through existing hypertexts. Different documents are separately prepared with specific editors and then linked together interactively.
2. With the programming language *MultiTalk* it is possible to write simple programs, called *scripts*. These are attached to objects like nodes and anchors, and are triggered by events such as a display of a node or an anchor selection.

[3] **Motif** is a registered trademark of the Open Software Foundation, Inc.

3. The *Hypermedia Toolkit* is a library of C functions for the creation and manipulation of persistent distributed hypermedia objects (such as nodes, anchors and links).
4. The *M2000 Protocol* and the *Compliant Editors* allow a user to edit the information contents of a node. Each node is associated with a specific editor (for example MCEditor, GO, Emacs, etc.) that is not included in HyperPATH but is "extended" in such a way as to communicate with HyperPATH via the predefined communication protocol *M2000*. It is then possible to create HyperPATH applications with any kind of node contents (text, image, audio, graphics, etc.) whose editor is compatible with the *M2000 protocol*.

The *Hypermedia Basic Classes (HBC)* module, written in C++, manages all objects representing the structure of a hypertext. Their persistence is provided by *mwdbms* running on several servers (DB-Server) which are connected by a network. An application represents a client that communicates with mwdbms using remote procedure calls (RPC).

2.2 The Data Model of HyperPATH

In HyperPATH, hypertext data is represented as a *graph*. A *node* of the graph can be either a single document (*atomic node*) or a group of nodes (*composite node*). *Links* (edges of the graph) can be defined between nodes without any restriction (e.g. there may be a link between an atomic node and a composite node). Finally, sequences of links can be stored as *trails* in order to provide a guided navigation mechanism to HyperPATH readers. A hypertext is *weakly typed*. Any of its hypermedia objects has an arbitrary list of properties. A property has a name and a value. Two "similar" nodes, e.g. two nodes containing documents with the same structure, can have different property lists. In addition, properties can be added to and removed from an object at any moment during the hypertext's life-time. This flexibility, typical for hypermedia applications, is in sharp contrast with *strong typing* in database models and in particular querying by (properties) values.

The HyperPATH data model is implemented in the C++ programming language. Objects are defined through composition and inheritance mechanisms resulting in the schema shown in Figure 2 : dashed lines represent inheritance relationships (e.g. the class `Link` is a subclass of `HyperObject`) and solid arcs represent composition. The labels of these arcs are the attribute names. Multi-valued attributes are distinguished from single-valued attributes by an additional <> symbol. For example, the arc from `Anchor` to `Node` labeled with *node* denotes that each anchor has an attribute with name *node* and whose value is the `Node` object containing the anchor. On the other hand, each node has an attribute *anchors* with the list of its anchors (class `Anchor`).

All classes are subclasses of the class `HyperObject`. The user semantics of each object, i.e. its list of properties, is represented by the attribute *propertylist* which is inherited from `HyperObject`. HyperPATH hypertexts are objects of the class `Hypergraph`. Atomic nodes (class `Node`) and composite nodes, hereafter

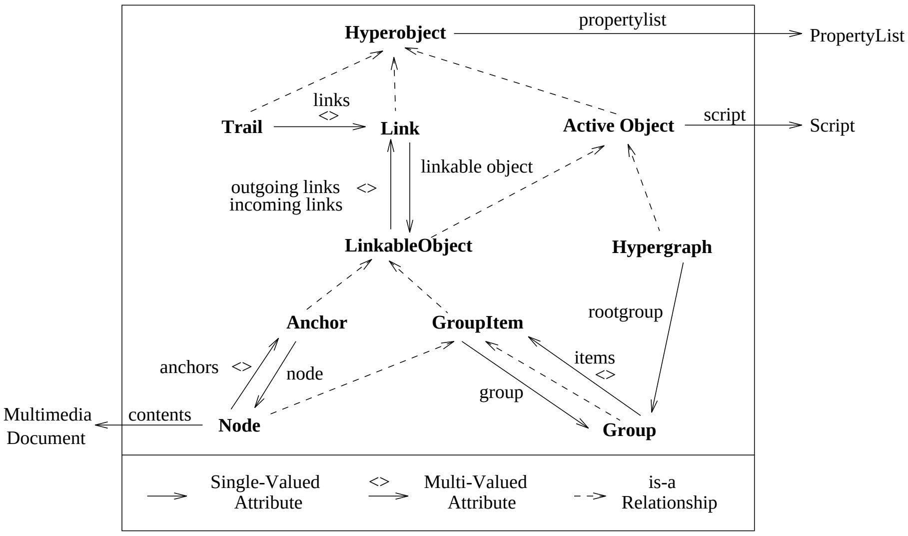

Fig. 2. Hypermedia Basic Classes (HBC)

called groups (class `Group`), form the instances of the class `GroupItem`[4]. The components of a group (nodes and groups) are stored in the *items* attribute. HyperPATH links (class `Link`) can be defined between groups, nodes and anchors (common superclass `LinkableObject`) without any restriction. A HyperPATH trail (class `Trail`) is then a list of links followed during user navigation in the hypertext. Objects of the class `ActiveObject` can receive events triggering the execution of an attached *script* (attribute *script*). Finally, the contents of a HyperPATH atomic node is not an object of the class hierarchy but is stored, for example as a UNIX file, by the corresponding compliant editor of the node.

The HyperPATH data model contains not only objects from the Hypermedia Basic Classes (HBC) hierarchy but also values such as scripts and property lists. From now on, we will distinguish between *HBC-objects* and *classes* and *HBC-values* and *types*. Both have to be rendered persistent. More precisely,

a) HBC-objects are created and managed by the persistence module (*internal interface*) and have a "global" identifier, called the *MCIdentifier* of the object. The MCIdentifier is also used by the *Compliant Editors* and the *Hypermedia Toolkit* (*external interface*) and it is composed of a) the *host* name, b) a hypertext identifier, called the *stack*, and c) an identifier for the object inside the hypertext, called the *local identifier* of the object.
b) HBC-values represent non sharable and non encapsulated attribute *values* of HBC-objects. However, they are also implemented as C++ objects (e.g. class `Script` and `PropertyList`). Since C++ does not provide bulk data types,

[4] Atomic nodes can be compared with files and groups with directories containing files and other directories.

lists are represented as Abstract Data Types (ADT) which are implemented as classes with appropriate methods for insertion, deletion, etc. As opposed to HBC-objects which are sharable and can be accessed from the external modules (*M2000* and *Hypermedia Toolkit*) by their MCIdentifiers, objects of the above classes do not have any logical identifier.

Besides forward references, implementing composition links in the HyperPATH data model, for efficiency reasons, there also exist inverse references from HBC-values back to HBC-objects. Since these inverse references are C++ temporary pointers they cannot be stored with the corresponding HBC-objects and must be initialized each time their owner objects are loaded from the database.

2.3 The HyperPATH Persistence Module

The persistence of HBC-objects and values is provided by the ad-hoc distributed database system *mwdbms*. A server-client architecture allows several database servers on different hosts for one HyperPATH application (client).

In each client, a *dbmsserver* (object of class `DbmsServer`) guarantees the persistence of all HBC-objects of one hypertext. It communicates with mwdbms by remote procedure calls (RPC). An *object manager* (object of class `ObjManager`) renders distribution of objects transparent to the above layers by handling a list of dbmsservers.

Each time the object manager needs an HBC-object with MCIdentifier *mcoid*, it sends a request to the dbmsserver. The latter initializes a new object with the database value obtained from mwdbms unless there is already an object with the same identifier *mcoid* in the *dictionary* of loaded objects. Each object has a reference to its dbmsserver. This allows faster access to an HBC-object of the same hypertext (e.g. a link accessing its source node) by calling directly the related dbmsserver, i.e. without passing by the object manager.

A hypertext can be saved only in its entirety. Each HBC-object receives from the dbmsserver the message to save itself. An object is saved only when its state has been modified. Saving a node also includes saving its contents. Obviously, an HBC-object is saved with its HBC-values.

3 C++ Interface of O_2

O_2 provides a C++ programming interface [15] which allows an O_2 database to be linked with an external application written in C or C++. By this, writing and reading to and from the database are transparent to the programmer. The *import* command takes an existing C++ declaration file as an input and generates its image schema in O_2. The *export* command renders O_2 objects visible to external applications. In general, each C++ class is extended by two methods *read* and *write*, which allow an object to be read/written from/to the O_2 database. Additionally, for each C++ class K a twin C++ class (*persistent pointer class*) o2_K is created and whenever an application wishes to access a persistent object, it might use the persistent pointer instead of the C++ temporary pointer.

C++ member functions (methods) are not imported. In contrast to O_2C methods, they are neither changed nor stored in the O_2 database. Nevertheless, it is possible to call O_2C methods from a C++ method. A named object (O_2 persistence root) is automatically available to the external C++ application when its class is exported. In the imported classes, temporary pointer declarations are automatically replaced by persistent pointer declarations. Non-imported classes are not modified. For example, if we define the `class A {B *b;}`, `class B {C *c;}` and `class C {A *a;}`, the importation of class `A` and `B` implies the following changes in the upgraded class definitions and methods : the attribute `B *b` in class `A` is replaced by the O_2 persistent pointer `o2_B b`. On the other hand the attributes `C *c` in class `B` and `A *a` in the non-imported class `C` remain unchanged. The methods of both imported and non imported classes are not modified. For an exhaustive description of O_2 persistent pointers structure and management see [3].

4 HyperPATH/O_2 : Three Levels of Integration

One of our requirements was to make as few changes as possible in the HBC module and definitely no changes in the *Hypermedia Toolkit*. Additionally, the node's contents (documents) are not stored in the O_2 database since in HyperPATH the persistence of the documents is under the responsibility of the compliant editors. We have examined the following solutions for the integration of HyperPATH with O_2 (see Figure 2):

1. **Weak connection** : O_2 only replaces some DB-Server and communicates with a HyperPATH client via remote procedure calls (RPC).
2. **O_2 is the persistence depositary of the HBC-objects** : We modify the C++ schema of the HBC-classes as little as possible, i.e. only the methods concerning the persistence module.
3. **Strong connection** : The HBC schema and thus the hypertext management itself is implemented in O_2. Note that most of the hypermedia functionalities of HyperPATH are implemented outside HBC (*M2000 protocol*, *Hypermedia Toolkit*).

The simplest solution for storing the HBC-objects is the substitution of mwdbms by an "O_2 database server" that simulates the interface of mwdbms (weak connection). The disadvantages of this solution become clear when we look at the steps necessary for storing an HBC-object. The database interface a) copies the C++ structure into a C structure and b) sends it by a remote procedure call (RPC) to the database server that c) receives the RPC, d) copies the C structure into a C++ structure and e) stores the structure in the O_2 database. Such a mechanism results in an extremely low performance (at least two data transformations each time) and low reliability (the transaction mechanism of O_2 is not used).

The second solution suggests that an O_2 database is the persistence depositary of the HBC-objects. The main idea is to build a persistent module that uses

O_2 in a transparent way to the rest of the HyperPATH architecture. This solution requires a redefinition of the front-end (database interface), i.e. the classes and the methods implementing the persistence of the HBC-objects.

The third solution (strong connection) makes possible to avoid the traditional type mismatch between programming languages and database systems. However it was not chosen in a first step since it requires a major rewriting of the existing code and because it is in contradiction with the objective of opening HyperPATH to various OODBMS's.

Finally, the second solution was adopted because of the encountered problems which will be described in detail in the following section. In the sequel we shall see that even in this case, the integration was not trivial and led us to three possible implementations of the HBC module.

5 "Open Integrations"

We first planned to provide persistence to the HBC hierarchy with different DBMS's (O_2 and ONTOS [1]). This heterogeneity should have been transparent to the above HyperPATH layers. The idea was to define for each DBMS and HBC-class a *persistent subclass*. For example, to the HBC class `Node` we add two subclasses `NodeO2` and `NodeONTOS`, which allow a node to be stored either in an O_2 or an ONTOS database. Then, after the importation of the existing HBC-class hierarchy into O_2, a HBC-class schema with multiple inheritance, as shown in Figure 3, is created.

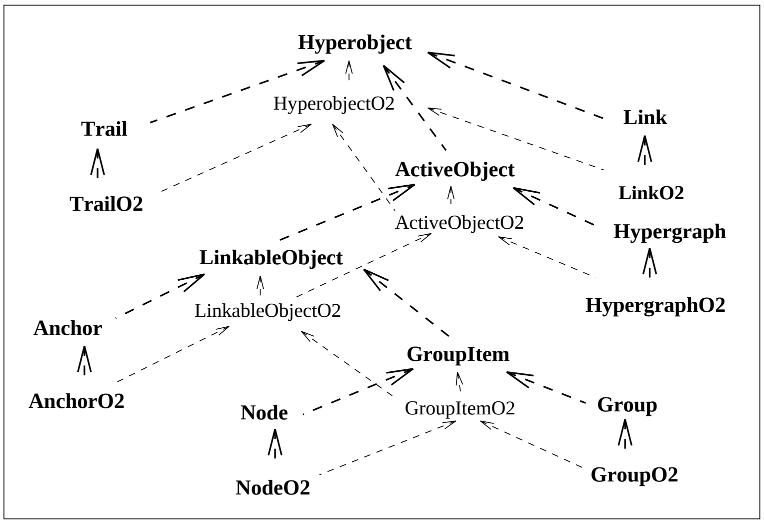

Fig. 3. HBC-Class Schema with Multiple and Simple Inheritance

Even if the use of multiple inheritance increases the complexity of the resulting C++ schema, it leads to a very natural solution for coupling HyperPATH with a variety of OODBMS's. It simply requires the definition of a minimal abstract interface between the HBC classes and their persistent subclasses whose implementation depends on the OODBMS used. But existing C++ restrictions for explicit casting rendered the implementation of this solution practically impossible. For a better understanding of the C++ inheritance mechanism and the difficulties encountered with this solution see [3].

In order to avoid the difficulties with multiple inheritance and keep the persistent module extensible to other OODBMS's we have investigated a schema based on simple inheritance where all HBC attributes that have to be stored are redefined in persistent subclasses. Looking at the resulting class hierarchy (bold classes and inheritance arcs in Figure 3) we can see that, at the O_2 level, we do not have any inheritance hierarchy and we create only 6 persistent subclasses for each of the HBC-classes that can be instantiated (`Link`, `Hypergraph`, `Anchor`, `Node`, `Group` and `Trail`). Likewise, HBC-values become persistent by adding the subclasses `ScriptO2`, and `PropertyListO2`.

The resulting C++ structure of each persistent subclass contains pairs of HBC-values with the same semantics : one attribute is defined at the database level (i.e. is an O_2 object) and one is inherited from the corresponding superclasses in the C++ schema (i.e. is a C++ temporary pointer). In order then to ensure consistency between non persistent and persistent HBC-values we are obliged to impose supplementary constraints during the object's creation and management. This solution turned out to be impractical because of the mismatch in object construction and management between C++ and O_2 [3].

In conclusion, the two possible solutions for providing HyperPATH with the possibility of using a variety of OODBMS's were abandoned. The following section describes the final solution that was chosen for the integration. It has the advantage that no rewriting of the *Hypermedia Toolkit* is necessary but it loses the generality of the "persistent subclass" solutions. The solution is specific to O_2 and does not represent an abstract interface to different DBMS's.

6 An O_2 Tailored Integration

6.1 The Hypermedia Basic Classes Schema

This solution consists in using the result of the importation of the whole HBC hierarchy as such. This implies that all HBC-classes become persistent via the O_2 persistent pointers. The C++ schema finally constructed is the same as the initial one given in Figure 2. After importation, the HBC-classes include the necessary methods for implementing the minimal interface with the rest of the HyperPATH modules as well as the methods for the persistence functionalities (`create, load, save` and `delete`). Thus the database interface and the persistence module still remain independent from the other HyperPATH modules.

6.2 Mapping of Object Identifiers

The mapping between object identifiers (MCIdentifiers) used in a hypermedia application and object identifiers of the database is another important issue. Since HBC-objects are accessed through their MCIdentifiers, one solution for a uniform integration of the two systems was the substitution of the MCIdentifier by the *OID* provided by O_2. In that way, we could "directly" access the HBC-objects in the O_2 database. This solution was rejected for two reasons :

1. The MCIdentifier is not only used by the HBC module but also to a large extent by the external interface. Thus, the uniform usage of the O_2 *OID* implies important modifications of HyperPATH modules that were not intended to be modified (e.g. *Hypermedia Toolkit* and *M2000 protocol*).
2. With a replacement of the MCIdentifiers with O_2 *OID*'s, we are faced with the same problem when attempting to read other hypertexts stored in different OODBMS (managing different *OID*'s).

Therefore, at least for reasons of openness, to each OODBMS system $\mathcal{S}$ associated with HyperPATH corresponds a table (dictionary) mapping MCIdentifiers to *OID*'s of system $\mathcal{S}$. When an object needs to be accessed, either there is an entry in the dictionary, or it has to be loaded from the database. Since each object has an attribute whose value is the MCIdentifier, loading is performed by querying the objects on their MCIdentifier attribute.

6.3 The Persistence Module

We have seen in Section 2.3 that the persistence of the HBC-objects in a hypertext is guaranteed by an object of the class `DbmsServer`, i.e. the dbmsserver loads and saves the HBC-objects. One way to represent this functionality in O_2 is to render the dbmsserver persistent and extend it such that it contains all HBC-objects of the corresponding hypertext. Since it is not possible to create an O_2 name for each new dbmsserver (C++ interface restriction), we define a named variable which contains a list of dbmsservers. In this way and because of the O_2 persistence transitivity, all HBC-objects become persistent together with their dbmsserver.

Now, all hypertexts can be accessed by the object manager through the persistent list of dbmsservers. Note that for HBC-values that are always attached to some HBC-object, access is provided implicitly by O_2. Each time we load a persistent HBC-object, its corresponding persistent HBC-values are loaded automatically by the C++ interface of O_2. Since we have replaced all temporary C++ pointers by persistent pointers, inverse references also become persistent and do not need to be reinitialized each time the dbmsserver loads an object from the database.

7 Conclusion and Future Work

One objective of the integration HyperPATH/O_2 was to deeply understand both products, in order to supply HyperPATH with query facilities. The integration has turned out to last much longer than expected and is not satisfactory as far as the performance is concerned. The major issues encountered during this experiment are related to the difficulties in reusing C++ code. Since HyperPATH uses C++ both as an object-oriented programming language and as an extension of C, we first encountered difficulties related to incompatibilities between object-oriented and imperative programming. In particular the coexistence of strong typing as implied by object-oriented models and explicit casting of C++ objects led to unexpected dead ends. We think that one major drawback of C++ is the lack of consistency in the type system conversion.

In addition, the mapping between two object identifiers (one for HyperPATH objects and one for O_2 objects) results in performance discrepancy. There is also some mismatch in object construction and management between C++ and O_2[3]. Thus, guidelines to let HyperPATH support both C/C++ and C++/OODBMS interfaces are quite ambitious and currently not easily implementable. This is why we advocate for a higher level interface between hypertexts and OODBMS's. For a more powerful integration between HyperPATH and O_2 (strong connection, see Section 4), the HBC layer of HyperPATH should be written in O_2C. Currently, the document's content is not stored in the O_2 database.For that, we plan to add a compliant editor able to communicate both with O_2 and HyperPATH. This experiment will also be useful for our next goal of providing HyperPATH with query capabilities [2].

In conclusion, constructing a HyperPATH persistence module extensible to a variety of commercial OODBMS's turned out to be a difficult task. Nevertheless, we believe that it was a necessary step towards the specification of common hypertext storage requirements for current database technology.

Acknowledgments

We are grateful to O_2Technology, Bull and Euroclid France for their support during this integration. Thanks also go to D. Plateau, A. Rizk, L. Sauter and A.M. Vercoustre for helpful discussions and to M. Ahedo and P. Dagand from Bull for fruitful exchanges on HyperPATH and their integration of HyperPATH with ONTOS.

References

1. M. Ahedo, P. Dagand, and L. Sauter. private communication, 1992.
2. B. Amann and M. Scholl. GRAM: A Graph Model and Query Language. In *Proc. of the ACM Conference on Hypertext (ECHT'92)*, pages 201–211, Milano, Italy, December 1992.
3. B. Amann, V.Christophides, and M. Scholl. HyperPATH/O2: Integrating Hypermedia Systems with Object-Oriented Database Systems (extended version). Technical Report 93-07, CEDRIC CNAM, Paris, France, April 1993.
4. B. Campbell and J. M. Goodman. HAM: A General Purpose Hypertext Abstract Machine. *Communications of the ACM*, 31(7):856–861, July 1988.

5. J. C. Chen, T. W. Ekberg, and C. Thompson. Querying an Object-Oriented Hypermedia System. In R. Mc. Aleese and C. Catherine, editors, *Hypertext: State of the Art*, pages 231–238. Billing & Songs, 1990.
6. N. Delisle and M. Schwartz. Neptune: A Hypertext System for CAD Applications. In *Proc. of the ACM Conf. on the Management of Data (SIGMOD'86)*, pages 132–143, Washington, D.C., May 1986.
7. O. Deux et. al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1989.
8. L. Gallagher, R. Furuta, and P. D. Stotts. Increasing the Power of Hypertext Search with Relational Queries. *Hypermedia*, 2(1):1–14, 1990.
9. F. G. Halasz. Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7):836–852, July 1988.
10. F. G. Halasz and M. Schwartz. The Dexter Hypertext Reference Model. In *Proc. of the NIST Hypertext Standardization Workshop*, pages 95–133, Gaithersburg, National Institute of Standards and Technology, January 1990.
11. J. Nanard and M. Nanard. Using Structured Types to Incorporate Knowledge in Hypertext. In *Proc. of the ACM Conference on Hypertext (HYPERTEXT'91)*, pages 329–343, December 1991.
12. A. Rizk and L. Sauter. Multicard: An Open Hypermedia System. In *Proc. of the ACM Conference on Hypertext (ECHT'92)*, pages 4–10, Milano, Italy, December 1992.
13. H. A. Schütt and N. A. Streitz. *Hyperbase*: A Hypermedia Engine Based on a Relational Database Management System. In *Proc. of the European Conference on Hypertext (ECHT'90)*, pages 95–108, Versailles, France, November 1990.
14. K. E. Smith and S. B. Zdonik. Intermedia : A Case Study of the Differences between Relational and Object-Oriented Database Sytems. In *Proc. of the Conf. on Object-Oriented Programming (OOPLSA'87)*, pages 452–465, October 1987.
15. O_2 Technology. *The O_2 User's Manual, version 3.4*, June 1992.
16. U. K. Wiil and J. J. Leggett. *Hyperform*: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems. In *Proc. of the ACM Conference on Hypertext (ECHT'92)*, pages 251–261, Milano, Italy, December 1992.

This article was processed using the LaTeX macro package with LLNCS style