Interoperable Databases: a Programming Language Approach

Olivier Jautzy
ENPC – CERMICS
Inria, 2004 route des lucioles F-06902 Sophia-Antipolis, France
Tel : (+33) 492 387 627
Fax : (+33) 492 387 740
E-mail : Olivier.Jautzy@sophia.inria.fr

Abstract

Even though federated multidatabase systems solve most of the issues in integrating heterogeneous data sources, they do not solve what we call data access heterogeneity: i.e. they do not allow users to structure and manage in the same way persistent, volatile and distributed data with the tools that match their needs. To solve this problem, we propose to add generic MDBS management facilities into a general purpose object oriented programming language. This extension is mainly based on new transactional and view approaches and the resulting system can be completely customized in order to match the needs of its users. An implementation of this system for Java, named JavaViews, currently allows to integrate ODBC as well as Versant or Poet databases.

1 Introduction

The growing amount of data that can be found through various media, including the Internet, had led to the heterogeneity of software throughout they are accessed, of hardware on which they are stored, and of the models used to structure these data. It is therefore hard for people to access and use simultaneously all the data they need, even though these data are online. As a matter of fact, one of the topical research activities is to model and develop softwares or database management systems that enable the integration of heterogeneous and autonomous data sources [28]. A classification of the different proposed solutions can be found in [20] and [29]. Among these solutions, the federated multidatabase approach (see [6]) has become the leading approach.

A federated multidatabase systems (MDBS) is a virtual database system that resides on top of existing data sources and presents a single database illusion to its users. One key-point of such a system is that all data are kept in the local data sources in order to guarantee their coherence. We think that it could be interesting to introduce this kind of architecture in a programming language.

After explaining, in the following of this section, the reasons of this statement, we introduce in section 2 the properties that should verify a programming language in order to allow users to access transparently heterogeneous data sources. Then in section 3, we present an implementation of such a system, named JavaViews, based on the Java programming language. The JavaViews system currently allows to easily integrate and interoperate Poet, Versant or ODBC database systems. Finally, we conclude in section 4.

1.1 Issues in Integrating Heterogeneous Database Systems

During the past few decades, file systems, hierarchical [32] and network [31] database systems, relational [9] and object oriented [7] database management systems, have been used as platforms for managing data. The problem of developing applications that require access to heterogeneous data sources that are managed separately have thus far been addressed in a few different way. The database research community has concluded that the most viable and general solution to this problem is the federated multidatabase systems. We present such systems in the following section.

1.2 Multidatabase systems

A MDBS is a database system that resides on top of existing data sources and presents a single database illusion to its users. Such a system is based on a unified metamodel, called the “global metamodel”, used to unify the heterogeneous underlying metamodels, so called “local metamodels”. Such a system can be structured as a stack of layers on top of heterogeneous and autonomous data sources. One key-point of such a system is that no data is contained in the different layers. All data are kept in the local data sources in order to guarantee the coherence of data. As a consequence, all the layers can be considered
as views on top of the local data sources or on top of other layers in the sense of the term introduced in [2].

![Multidatabase Systems Architecture](from [29])

The figure 1 shows the minimal architecture of an MDBS as it was described in [29]. The main issues in the design of MDBS concern, in the one hand, the view definitions, data access and updates, and, in the other hand, transaction processing. Most of these problems are discussed in [19], [18], [25] or [5] but are not currently completely solved.

1.3 About heterogeneity

The goal of these systems is to integrate heterogeneous data sources. In other terms, there aim at hiding some kind of heterogeneity. This latter term is yet very general. Therefore we describe below more precisely the two kind of heterogeneity that we consider in this paper.

**Data sources heterogeneity.** We have seen that systems like MDBS were designed to present a unified view of a set of local data sources. In order to achieve this goal, they try to hide what we call data sources heterogeneity. This concept includes the followings :
- Network, OS and hardware heterogeneity
- Metamodel heterogeneity : for instance, a data source can be a relational DBMS and another one can be an object oriented one.
- Model heterogeneity : for instance a same entity with the same name in two distinct local data sources can model two distinct real-world entities and vice-versa.
- Processing heterogeneity : tools used to manage local data in each data source, like query processors or transaction managers are based on heterogeneous concepts and models.

**Data access heterogeneity.** Each user or each application generally needs different kind of access to the same data or needs to manage different nature of data (i.e. persistent, volatile or distributed data). This leads to the following kinds of heterogeneity.

Heterogeneity of the nature of data.

Heterogeneity of views : each application or each user has different needs in term of data structure. Therefore, in the integration process, they need different views of the same set of data sources.

Heterogeneity of data processing. each application or each user needs different kind of tools to manage the data. For instance, one application may need to manage long transactions or object oriented query processing while another one needs only flat transactions or relational query processing.

It is obvious that MDBS do not solve this second kind of heterogeneity. Moreover, it looks like they augment it since they are homogenizing heterogeneous data sources in the same way for all users or applications. These ones must access data through MDBS concepts even though they need properties that are present in local data sources but hidden by MDBS ones. For instance, let us assume that a MDBS integrates two data sources. The first one allows processing of flat classical transaction while the second one allows long transactions processing. Moreover, suppose that the MDBS only manages flat transaction. Then, users and applications working on the MDBS must use this latter kind of transaction even though they want to use long transactions and would have used them if they were working only on the second data source. A similar issue concerns the global schema of MDBS which are commonly defined a priori without the knowledge of the needs of users or applications. As a consequence, this definition is often the result of a trade-off between the size of the schema, which has lot of consequences on the system performance, and the supposed needs of users or applications working on the MDBS.

Until now, we have used both the terms of user and application to describe the different entities that can access a MDBS. However, since most of the users of an MDBS are accessing data through an application, we will concentrate in the following on applications and therefore we will put the stress on programming languages used to develop them. In fact, we will consider that “real” users of MDBS are programmers and that other users are only working on applications written by these ones. This leads us to look at the different approaches used to access persistent data through a programming language (see figure 2). These approaches are the loosely coupled approach where the persistent system is external to the programming language, the tightly coupled approach where the persistent system is tightly included in the programming language and a mix of the two preceding approaches, that we call mixed coupling approach, where a persistent sys-
tem is tightly included in the programming language and where it can also used other external persistent systems.

![Diagram showing coupling approaches]

**figure 2 : Coupling approaches**

DBMS are clearly accessed by following the first approach. Since the access to such systems is often normalized, this approach is independent of programming languages. However, the programmer needs to know and/or learn the new semantic and concepts of each DBMS or MDBS that he wishes to use. Moreover, it is very hard to specialize data access in order to match application needs. The second approach is commonly used when designing persistent programming languages like for instance PJava [11]. It allows the programmer to use only one semantic in order to manage data access. If the programming language can be specialized, it also allows to solve data access heterogeneity. However, this approach is dependent on one programming language. Moreover, it does not solve data sources heterogeneity since users must save their data in one particular kind of database. The third approach, that we have chosen, combines the advantages of both the first approaches. The programming language includes a persistent architecture and persistent data management tools but also knows how to delegate some treatments to external systems. Therefore, even though this approach is dependent of one particular programming language, it could be used to solve both data sources heterogeneity and data access heterogeneity.

2 Design choices

In this section, we present how to add general MDBS facilities into a general purpose object oriented language by following a mixed coupling approach.

2.1 Views, Transactions, Persistency and queries

The first choices we have to make is to determine which concept or which processing tools arising from MDBS should be included in the programming language. We have already stated that MDBS are “only” complex views systems since data are kept in their own local data sources. From the programming language point of view, it implies that the language should not be aware of stocking data explicitly like it must be done, for instance, in persistent programming language. In fact, it should only allows programmers to create, in the one hand, proxies for persistent data that correspond to the wrapper layer of MDBS and views that correspond to the federated layer of MDBS and that can be created on top of such proxies, or on top of other views, or even on top of other classical structures of the programming language. This architecture allows to manage classical structures, view structure or persistent structures in the same way. Moreover, since data are kept in local data sources, library containing views and/or proxies can be shared by many applications, leading to the illusion of a virtual global data source. Accessing persistent data, saved in DBMS, must be done in a transactional environment and often by the mean of query processing. Since our system allows managing persistent and volatile data in the same way in order to solve the data access heterogeneity, it must include a transaction manager and a query processor.

To sum up, our system has to allow, in the one hand, view definition, transaction management and query processing for all kind of data (volatile, persistent and distributed) and in the other hand, the creation of proxies for persistent data.

2.2 Object entities

Let us consider the transaction aspect. The API of most classical transactional models contains some operators, for instance begin, commit and abort, that allow the programmers to mark the code that is implied in a transaction. Other API, defined for more advanced transaction models, include more behavior. For instance, the split operator, defined by [15], must be included in order to allow dynamic restructuring of transactions. Or, in order to use the two phase commit protocol (2PC [14]), a prepare_to_commit operator must also be included. Moreover, the semantic of the same operator can change from one transactional model to another. For instance, in the flat transactional model [14], the commit operator leads to
make visible for every other transactions the effects of the current transaction. In the nested transactional model [26], however, the commit operation leads to make the effects visible only for the ancestors and sisters of the current transaction. Therefore, since we want to allow each programmer to refine our system in order to work with the transactional model that matches the needs of its application, our transactional system as well as, more generally, our view or query system, must be extensible and polymorphic. These two properties are the base properties of every object oriented programming language. As a result, our system is based on such a language by including each new entities or concepts as classes, objects or behavior.

2.3 Atomic properties

To solve the data access heterogeneity issue, we must allow programmers to design applications that verify only needed properties for each managed objects and that are optimally structured. For instance, following the transactional point of view, an application can be composed of different parts, some of them being controlled by a transactional execution and other not. Moreover, it seems interesting to allow the programmer to chose for each object, the kind of transactional properties applied to them and the algorithms used to enforce these properties. As a consequence, we have designed our system by considering that each new concept like views or transactions must be include in the language as many separated and distinct atomic concepts. These atomic concepts can then be grouped together during the application design time in order to form classical concepts like ACID transactions, and can also be used alone.

The result of this part of our work currently concerns only transaction and view concepts. It is summed up below:

**Views**: A lot of work has already been done one adding view abilities to object oriented DBMS (see for instance [27] or [3]). At the difference of these previous works, we introduce the concept of view as two distinct concepts. In our system, we call virtual class the structure part of the view. We then introduce the mapping concept that contains how a virtual class is mapped to other structures, let us say base classes, of the application. A mapping is associated at run time with a virtual class. The set of virtual instances or the extension of each virtual class is the union of the extension of each of its associated mapping. These ones are calculated from the extension of the base classes of the mapping.

**Transactions**: In our system, we have reduced the concept of transaction to its minimal size. In fact, the role of our transactions has been reduce to the control of the execution of the application like any other control structure. Enforcement of transactional properties like atomicity or concurrency control is delegated to objects themselves or to external transactional systems for persistent objects. During the design of each class, the programmer describe which default properties will be verified by each instance of such classes. These properties can be changed at run-time for each object. The system ensures that each object implied in a transactional execution verify the properties that are associated with it. Each object is then completely independent of the others in terms of transactional properties. Let us note that this architecture is actually valid for volatile (i.e. classical), persistent (i.e. proxies) and virtual classes.

2.4 Implicit Semantic

We would like to allow users to use our system to access heterogeneous data sources even though they are not expert in database management. In other words we would like to allow programming application that access heterogeneous data sources in the same way as programming classical and non persistent applications. Such design must allow users to implicitly manage transactions, query and views for the following reasons:

Programmers are then relieved of erroneous and error prone tasks such as setting locks or notifying updates explicitly. This improve safety and reduce development time.

The majority of the code remains unchanged compared to a classical development context. It then greatly increases code-reuse.

This choice has some consequences on transaction management, view and query processing. This is sum up below.

**Views**: we have designed our system to allow creation of views in the closest possible way from the creation of classes. The programmer defines virtual classes as any other classes, with field and method declarations. Then he must define at least one mapping. This task is very close to the definition of classes as we will see in the 3.6 section even though some minimal new syntax has been added to the programming language in order to describe mappings. After the views and mappings description phase, users then manage virtual classes as any other classes.

**Queries**: Since our goal is to integrate heterogeneous data sources that may be accessed using heterogeneous query languages, we have to choose one particular query language that has to allow, in order to help to solve data access heterogeneity, to query in the same way persistent classes (i.e. proxies of our system) as well as virtual classes and even volatile classes. Since we have chosen to extend an object oriented programming language, it should have been obvious to choose the OQL [7] or SQL3 query languages. However, with the idea to permit non expert users to use our system, we have preferred to design the query syntax using only the programming language syntax. A query in our system is in two parts, a
projection that is composed of a classical expression of the programming language and a restriction that is composed of a classical boolean expression of the programming language. Our query system allows to query in the same way, classes, proxies and virtual classes. Finally, each request is evaluated on only one class at a time. In particular we actually do not allow joins.

**Transactions**: Most of current DBMS transactional API permits to explicitly manage transactions by sprinkling codes with explicit locking claims, transaction begin and end operations. Even though this approach is very powerful, it means that the application logic may be obscured or that it is easy to misinform the transactional engine by making erroneous explicit calls. Transactions are execution control structures that can be stopped in order, for instance in the case of an atomic transaction, to abort all the effects of the transaction. This mechanism is more or less already included in some object oriented programming language: the execution is controlled by message sending and can be stopped by the exception manager.

We have based our transactional system on these properties of object oriented languages. We have introduced in our system the concept of transactional message sending (see figure 3). In order to allow implicit transaction management, objects implied in a method that is call with such transactional message are managed by a transaction automatically created before the begin of the method. If the method runs until its end, the transaction is committed, else if a precise kind of exception is thrown during the method execution, the transaction is aborted. Like classical message sending, we allow transactional message sending to be nested and multithreaded.

### 2.5 Metaobject protocol

Two distinct concepts can be found in object oriented programming languages: class and object. Classes generate or create objects. Hence, the programmer faces incoherence since he must use message sending to communicate with objects while he must use a procedural approach to create objects because classes are not objects.

The metaclass concept was introduced to solve this problem [17]. In programming languages including metaclasses (see for instance [10], [12], [16]), each class as well as each metaclass is an instance of a metaclass as well as each object is an instance of a class. Therefore classes and metaclasses are objects and can be managed as such. A MetaObject Protocol (MOP) is a set of metaclasses with their behavior.

We have chosen to use a metaobject protocol in order to design our system for the following reasons:

A metaobject protocol allows to create new entities for an existing reflective language (i.e. a language that can describe himself by the mean of a MOP). As we have to create views and proxies that are similar but distinct from classes, it is interesting to use a MOP approach to allow simple description and ease of use of such entities. More generally, a MOP allows to add new syntax to a language. We can use such properties for including, for instance, the keywords that are necessities for transactional message calls.

Since a MOP is a set of metaclasses that are also objects, it can also be specialized. In our system this property is essential since it allows users to specialize each behavior of the system in order for the system to match the needs of the application.

### 3 The JavaViews system

In this section, we present an implementation of our system in the Java [13] programming language. We have chosen this language because it is nowadays one of the most widely used and that it is well adapted to access data sources over Internet.

#### 3.1 OpenJava

Since the Java reflective API is not sufficient to allow the complete implementation of our system, we had to find an extension of this language that permits to use a more powerful MOP. Java is in some way a compiled language. MOP are often only found for interpreted ones since they need a lot of dynamism. However some work has been done on including a MOP in compiled programming languages. These works have concluded that the most suitable approach is to realized compile-time MOP. The most achieved systems including such MOP are OpenC++ [8] and OpenJava [33] that extend respectively C++ [30] and Java. Such systems are very powerful preprocessors that can be completely specialized following a MOP semantic. In other terms these preprocessors are written in the base language of each system (C++ and Java) and are very well designed in order for the programmer to follow a MOP semantic that allows to easily refine the preprocessor.

Our work is based on the OpenJava system. Therefore, implementing our system has led, in the one hand, in order to specialize the preprocessor, to the definition of new Java classes, that are extending OpenJava ones and that we call metaclasses, and in the other hand, to the definition of a Java library which some parts are developed using our system, and that must be linked with the user application.

Using OpenJava and therefore our system is very simple. For each class that needs some special behavior, let us say some special preprocessing, the programmer precise which metaclass will lead the preprocessing. Each class is then associated with a metaclass, or by extension each class is instance of the metaclass that leads its preprocessing. Note that using a preprocessing tool allows to
generate classical Java byte code. As a result, applications developed using our system are “Java compatible” and the well-known “develop once – run anywhere” property of Java is preserved.

In the following, we quickly describe the main metaclasses of our system.

3.2 Queries

In our system, queries are composed of one Java boolean expression, the restriction part, and an expression, the projection part.

In order to allow the programmers to fine tune up their application, only some kind of classes are able to answer queries. These classes are either instance of the QueryAbleClass metaclass or are virtual classes or proxies classes. For such classes a static select method is defined. This method takes the restriction part and the projection part in arguments and returns the result of the query. The query processing is realized in memory by an ad-hoc interpreter and/or by an external DBMS for proxies classes. The figure 3 presents an example of querying a class named Person instance of QueryAbleClass

```java
metaobject Person instanceof QueryAbleClass;
public class Person {
    String name;
    ...
    public static void main(String[] args) {
        List result = select("firstname");
        "name.equals("foo")");
    }
}
```

**figure 3 : JavaViews query sample**

3.3 Transactions

In our system, transactions are controlling the execution of the application and each object contains its own transactional behavior.

From a classical object oriented point of view, each object shares the same behavior as the other instances of its class. Since transactional behavior can be considered as every other behavior, the transactional behavior of each object is, in our system, defined in its class. However, we also allow the re-definition of this behavior at run-time.

Three kind of metaclasses are then needed for our transactional system:

The first kind of metaclasses allows the programmer to precisely fix which methods of the application can lead to the beginning of a transaction. Such methods must currently be included in the declaration of a class instance of TransactionAbleClass and must not be modified by the untransactional keyword.

```java
metaobject Account instanceof TransactionableClass
defaultBehavior {AccountAtomicLocking}
class Account {
    double balance;
}
metaobject AccountAtomicLocking instanceof AtomicLockingBehaviorClass
class AccountAtomicLocking
forClass {Account} {}
metaobject Transfer instanceof TransactionableClass
class Transfer {
    public void transfer {Account c1, Account c2, double amount} {
        c1.putBalance(c1.getBalance()-amount);
        c2.putBalance(c2.getBalance()+amount);
        if (c1.getBalance()<0)
            throw new TransactionRuntimeException("*");
    }
}
public untransactional static void main(String[] args) {
    try {
        Transfer t = new Transfer();
        Account c1 = new Account();
        c1.putBalance(5000);
        Account c2 = new Account();
        c2.putBalance(5000);
        t.transfer(c1, c2, 500); // not transactional
        t.transactional_transfer(c1, c2, 500);
        //transaction committed
        t.transactional_transfer(c1, c2, 500);
        //transaction will be aborted
    } catch (TransactionAbortException e) { }
}
```

**figure 4 : JavaViews transaction sample**

The second kind of metaclasses (for instance TransactionalClass) allows the programmer to describe classes which instances, that we will call transactional objects, have a default transactional behavior. This behavior is define by a new syntax that associates each class with an instance of one of the metaclasses of the third kind. At run-time, each transactional object can be re-associated with another instance of such a metaclass.

The third kind of metaclasses for instance AtomicClass) allows the programmer to describe the transactional behavior of the instances of a precise class. Actually, we provide metaclasses that represents atomic behavior and/or locking concurrency control with deadlock detection. Let us note that the programmers can define virtually any other metaclasses in order, for instance, to allow optimistic concurrency control.

The figure 4 presents an example of transferring money between two accounts. Each account is modeled by an instance of the Account class and managed following an atomic property and with a locking concurrency control.
3.4 Persistence

The persistency property is added to our system through the concept of proxies classes. A proxy class is a class which instances are persistent (i.e. saved in a database). They can be queried by following our semantic and can be associated as well as their instances with a transactional behavior like any other classes. Note that this latter property implies that the persistent objects have the transactional behavior specified for them in the application and not the transactional behavior of the underlying DBMS. We actually provide three metaclasses that allow respectively to access Poet databases, Versant databases and ObjectDriver [22] databases. This latter system is a virtual object DBMS on top of relational DBMS. It allows actually to access any ODBC compliant database, so do our system. Moreover, with few extensions our system should allow to access any ODMG compliant DBMS.

The figure 5 presents the same example as the figure 3 excepted that instances of the class Person are retrieved here from a Versant database named demo.

```java
metaobject Person instanceof VersantPersistentClass;
public class Person
database {demo}
{String name;
String firstName;
}

public static void main(String[] args) {
  Dlist result = select("firstName;",
    "name.equals("foo");");
  ...
}
}
```

**figure 5 : JavaViews persistent sample**

3.5 Views

In our system, two concepts are necessary to implement views: virtual classes and mapping.

A virtual class is a Java class instance of our View-Class metaclass. There is no restriction on the description of fields or methods of a virtual class. Virtual classes can be queried by following our semantic and can be associated as well as their instances with a transactional behavior like any other classes. Virtual instances (i.e. instances of virtual classes) are automatically updated when their base instances are updated. The inverse update is also allowed depending on the corresponding mapping. A mapping is a modified Java class instance of the MappingClass metaclass. A mapping is associated with a particular virtual class at run-time. It describes how the fields of the virtual class are mapped to one or many fields of one or more base classes that are either instance of BaseMappingClass, virtual classes or proxies. The mapping declaration allows constraints, one-to-one and one-to-many mapping descriptions. For this latter case, special expressions can be used by the programmer to describe how to update virtual instances.

```java
metaobject Person instanceof ObjectDriverPersistentClass;
public class Person
database {demo}
{
  long ss;
double salary;
double bonus;
}

metaobject PersonName instanceof BaseMappingClass;
public class PersonName {
  double ss;
  String name;
}

metaobject PersonView instanceof VirtualClass;
public class PersonView {
  String name;
double salary;
}

public static void main(String[] args) {
  ...
  addMapping(new PersonMapping(), "map");
  Dlist result = select("salary;",
    "name.equals("foo");");
  RemoveMapping("map");
  ...
}
}

metaobject PersonMapping instanceof MappingClass;
public class PersonMapping
structured_by(PersonView)
based_on(pl, Person; p2, PersonName)
{field name = p2.name;
  field salary = f(pl.salary, pl.bonus);
}

double f(double a, double b) {return a+b;}

constraint complex () {
  p1.ss==p2.ss;
}
```

**figure 6 : JavaViews view sample**

The figure 6 presents our view system. In this example, we describe one proxy class named Person. The instances of this class come from the demo database of the ObjectDriver system. We then describe the PersonName class that can be a base class for a mapping since it is instance of the BaseMappingClass. Then we describe a view, modeled by the PersonView virtual class. At last we describe a mapping that will be associated at run time with the virtual class and that is based on the Person and PersonName classes. This mapping describes
that each virtual instance of PersonView is based on an instance of Person and on an instance of PersonName that must have, by the mean of a constraint, the same ss number. Each virtual instance of the view then associates the total salary with the name of a person.

4 Conclusion

In this paper we have described a MDBS extension for the Java programming language that allows to integrate existing and autonomous heterogeneous data sources by following the needs of each application. This extension is based on a mixed coupled approach which implies that any concept of our system can be applied on persistent, volatile or distributed data. Our system solve both the data sources heterogeneity and the data access heterogeneity. The extension is composed of a query processor, a view system, a transaction manager and allows to access concurrently Poet databases, Versant databases and any ODBC relational databases with the help of the ObjectDriver DBMS.

Our system has been designed in order to make the development of applications, according to multiple heterogeneous data sources, very easy. Moreover, each concept included in our system can be customized for each object of the application. More precisely, we propose new architectures for transaction processing and view definition that, in the one hand, match requirements of object oriented programming concept and, in the other hand, allow programmers to finely tune their applications. Currently, we haven’t yet seen a system that can globally be compared to our work. However, a lot of work has been done on incorporating the transaction concept into a general-purpose object oriented programming languages (see for instance Argus [23], Arjuna [24], PJava [11]). Nevertheless, these systems haven’t very well solved the data access heterogeneity since they do not allow all the forms of customizations that we propose (transactional model customization but also dynamic customization for each object or each execution, …) and often impose that only persistent objects are managed by the transaction manager. Besides, they are often based on a classical transactional API and do not propose an implicit transactional property as our transactional message call.

On the side of views, other systems integrating this concept in a general purpose object oriented programming languages has not yet been designed. Views structures are always left in the DBMS or MDBS (see for instance [27], [4], [1], [21]). Moreover, such view systems often do not allow instant bottom-up updates or even up-bottom updates. Finally, these systems do not propose the ability of dynamically combining many mappings for one view.

To sum up, we think that our system, by including and adapting in a general purpose object oriented programming language each interesting aspects of the MDBS architecture, answers pragmatically to the more and more expressed needs of real programmers that want to access simultaneously heterogeneous data sources.

5 References