Journal of Systems Integration, 7, 99–126 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Providing Database Interoperability through Object-Oriented Language Constructs

EVAGGELIA PITOURA pitoura@cs.uoi.gr
Department of Computer Science, University of Ioannina, GR 45110 Ioannina, Greece

Abstract. Seamless access to resources and services provided by distributed, autonomous, and heterogeneous systems is central to many advanced applications. Building an integrated system to provide such uniform access and cooperation among underlying heterogeneous systems is both an increasing necessity and a formidable challenge. An important component of such a complex integrated system is a unified language that serves both as a data definition and as a data manipulation tool. Special requirements are posed in the instance of an integrated system which includes database systems among its components. In this paper, we introduce the necessary constructs that an object-oriented programming language should provide for being adopted as the language of the integrated system in such a setting. We adopt a modular, object-based approach to integration. Each component system that joins the integration provides a set of basic classes and pre-defined basic methods. We show how the class hierarchy of the system can be used to provide a uniform way of mapping database resources to basic classes and associative queries to basic methods. A view mechanism is introduced that supports the integration of the basic classes and provides a means of expressing relationships among them and resolving any potential conflicts. The view mechanism is implemented by extending the standard class constructors of an object-oriented language to support the definition of virtual classes. The language provides workflow constructs for defining the structure of programs and for attaining synchronization among concurrently executing programs. Furthermore, atomicity and concurrency control information is included in the form of consistency assertions as part of the interface of each basic method.

Keywords: database integration, multidatabase, object-oriented languages, assertions, workflow, views

1. Introduction

Today, worldwide high speed networks connect numerous information systems which provide access to a wide variety of data resources. Although no provision for a possible future integration was made during the development of these systems, there is an increasing need for technology to support the cooperation of the provided services and resources for handling complex applications. The requirements for building an integrated system that combines this available information can be met at two levels [23]. The lower-level ability of systems to communicate and exchange information is referred to as *interconnectivity*. At a higher level, systems would not only be able to communicate but additionally be capable of interacting and jointly executing tasks. This requirement is referred to as *interoperability*.

Building an integrated system is an intricate task complicated by the heterogeneity of the underlying systems. *Heterogeneity* manifests itself through differences at the operating, hardware, or communication level of the participating systems [37]. In the special case where database systems are being integrated, discrepancies among data models and query languages and variability in system-level support for concurrency, commitment, and recovery constitute additional sources of heterogeneity. Finally, frequent disparities occur in the semantic interpretation of stored information, caused by social or organizational differences among the users or developers of the systems. The process of building heterogeneous systems is further complicated by the fact that some of the component systems are *autonomous* and have complete control over the execution of all operations that access their resources.

Central to the integrated system is a language through which the integrated data is defined and manipulated. We call this unified language *multilanguage*. In this paper, we define necessary constructs which permit an object-oriented programming language to serve as a multilanguage in the special case where the integrated system includes database systems as components. We adopt an object-based approach in which each component system that joins the integrated system provides a set of basic classes and a set of pre-defined basic methods. Rather than defining yet another language, we propose minimal extensions to the existing basic constructs of an object-oriented language without modifying their semantics.

The main contributions of this paper are as follows. First, a view mechanism is introduced that is implemented by extending, without altering their usual semantics, the standard class constructors (i.e., the new class, subclass, and superclass constructors) of an object-oriented language to include virtual classes. A virtual class is a template that describes already-created instances. We show how this mechanism supports the integration of the basic classes and provides a means of expressing relationships among them and resolving potential conflicts. Second, we show how the class hierarchy of the system can be utilized to provide a uniform way of mapping database resources. Finally, we define workflow constructs to express the structure and order of execution of the multilanguage programs and synchronization constructs to control the interaction among programs. In addition, atomicity and concurrency control information is expressed in the form of consistency assertions for each basic method. We develop semantic-based correctness criteria and mechanisms for enforcing them in a decentralized manner.

The remainder of this paper is structured as follows. In the next section, we introduce the object-oriented approach to integration and the basic functionality of the multilanguage. In Section 3, we define constructs for supporting database integration and functionality through extensions of the standard class constructors. In Section 4, we show how these constructs along with the class hierarchy of the system can be used to support sets and associative queries and database integration. In Section 5, we introduce consistency annotations as a means of including atomicity and concurrency information within class definitions and special constructs for controlling the execution flow. In Section 6, we demonstrate the use of these constructs for concurrency control. Finally, we discuss related work in Section 7 and summarize our conclusions in Section 8.

2. The System Model

The proposed methodology is grounded upon an object-based approach to integration.

2.1. An Object-Oriented Approach to Integration

With object technology, the requirements of interconnectivity and interoperability are naturally met. In this scenario, the resources of the various local systems that participate in

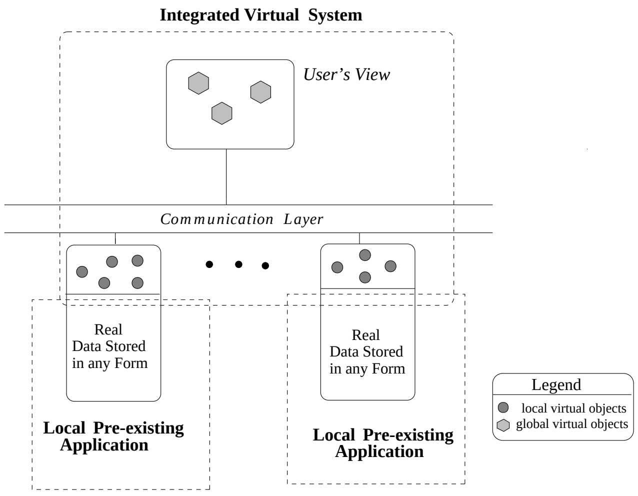
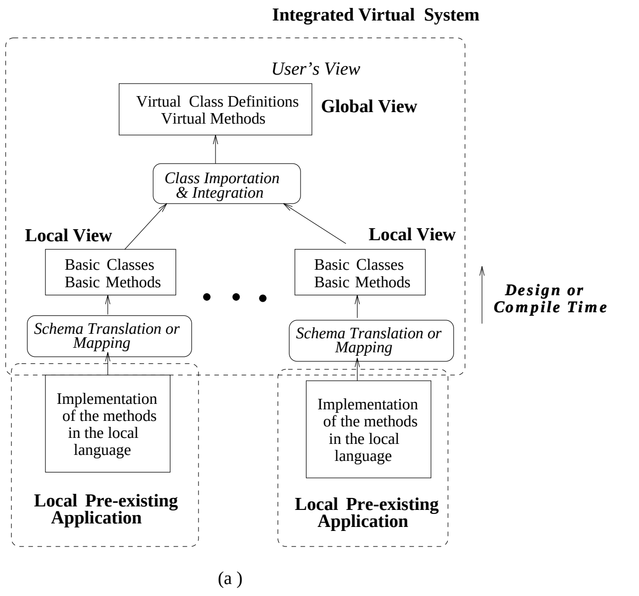

Integrated Virtual System
User's View
Communication Layer
Real Data Stored in any Form
Local Pre-existing Application
Real Data Stored in any Form
Local Pre-existing Application
Legend
local virtual objects
global virtual objects

Figure 1. Distributed object architecture.

the heterogeneous system are modeled as objects, while their services are modeled as the methods provided by these objects. The methods constitute the interface of the objects. We refer to this architecture as *Distributed Object Architecture* (Figure 1). We call the provided objects *virtual* to emphasize the fact that they may not be actually stored in toto but instead may be computed by combining existing resources (Figure 1).

Objects of similar behavior and structure are grouped together in classes. Each system that participates in the integration provides a set of classes, called *basic classes* and a set of methods that constitute their interface and are called *basic methods*. These basic methods offer the only means of accessing the virtual local objects. Each component system provides an implementation of the basic methods which is private to the local system and completely hidden from users of the integrated system. The basic classes and methods constitute the local view of each component system. During integration, new classes, called *virtual classes*, are defined by combining the provided basic classes using the view mechanism of the multilanguage. The methods of the virtual classes are combinations of the basic methods (Figure 2(a)). During operation, the user or client interacts with the system by using the multilanguage to send messages to the objects of the virtual classes. The system is responsible for translating these messages in terms of the basic methods, for directing the methods to the appropriate local system for execution, and then for combining the results to present the user with the output of the requested computation expressed in the multilanguage (Figure 2(b)).

When the component systems are database systems, the resources are the information stored in the database, while the provided basic methods are efficient mechanisms for retrieving and updating this information. In this case, object technology is also applied at a finer level of granularity. The information stored in a database is structured according to a data model. When a database participates in the integrated system, its data model is mapped


Integrated Virtual System
User's View
Virtual Class Definitions
Virtual Methods
Global View
Class Importation & Integration
Local View
Local View
Basic Classes
Basic Methods
Basic Classes
Basic Methods
Design or Compile Time
Schema Translation or Mapping
Schema Translation or Mapping
Implementation of the methods in the local language
Implementation of the methods in the local language
Local Pre-existing Application
Local Pre-existing Application
(a)

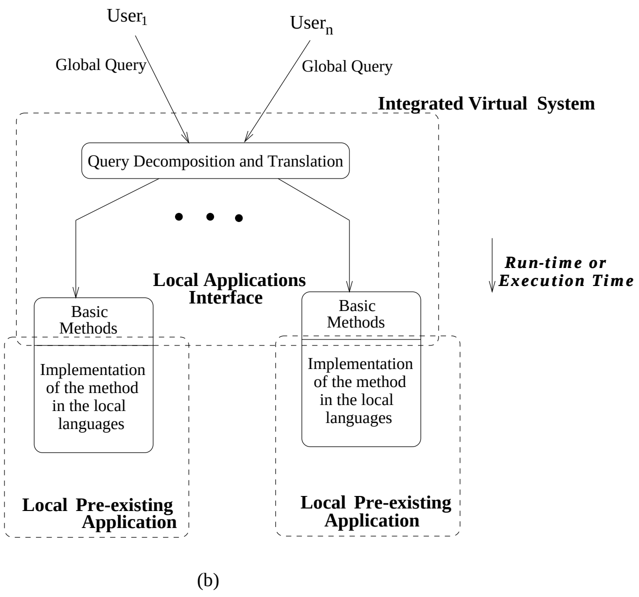

User$_1$
User$_n$
Global Query
Global Query
Integrated Virtual System
Query Decomposition and Translation
Local Applications Interface
Run-time or Execution Time
Basic Methods
Basic Methods
Implementation of the method in the local languages
Implementation of the method in the local languages
Local Pre-existing Application
Local Pre-existing Application
(b)

Figure 2. The use of the multilanguage (a) at design or compilation time, to define views and (b) at execution time, to express queries.

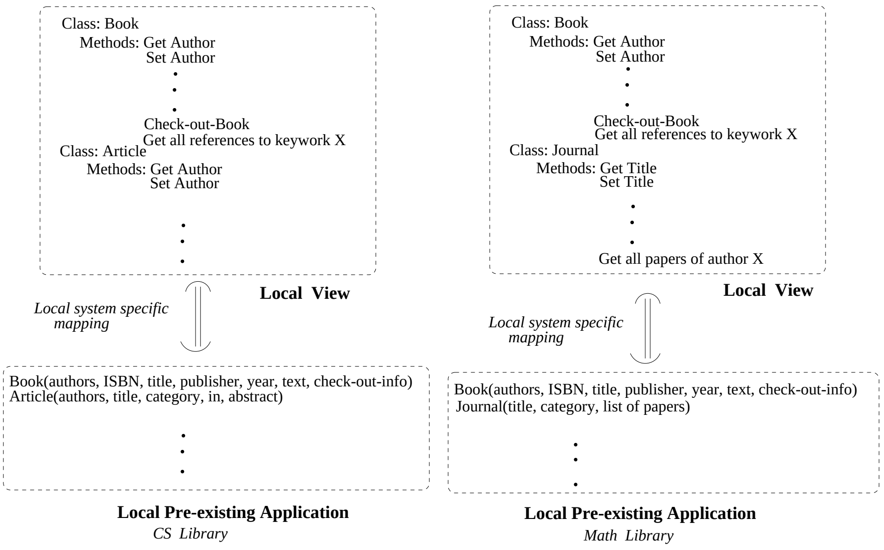

Class: Book
Methods: Get Author
Set Author
Check-out-Book
Get all references to keywork X
Class: Article
Methods: Get Author
Set Author
Local View
Local system specific mapping
Book(authors, ISBN, title, publisher, year, text, check-out-info)
Article(authors, title, category, in, abstract)
Local Pre-existing Application
CS Library
Class: Book
Methods: Get Author
Set Author
Check-out-Book
Get all references to keywork X
Class: Journal
Methods: Get Title
Set Title
Get all papers of author X
Local View
Local system specific mapping
Book(authors, ISBN, title, publisher, year, text, check-out-info)
Journal(title, category, list of papers)
Local Pre-existing Application
Math Library

Figure 3. An example. Each local application provides a mapping of its data to a local view that consists of a set of classes. The methods of these classes (called basic methods) offer the only means of accessing the data of the pre-existing applications.

to the same for all participating systems data model, called the *Common (or Canonical) Data Model (CDM)*. This model can be an object-oriented data model. The objects of the database model are of a finer granularity than the distributed objects. At one extreme, the entire database may be modeled as a single distributed complex object. Although object granularity may vary at different levels of the heterogeneous system architecture, the same basic object abstractions can be applied.

In the remainder of this paper, we shall identify these basic abstractions that a multilanguage must provide.

The running example. Assume two local databases, one corresponding to the library of the Computer Science Department and another corresponding to the library of the Department of Mathematics. Figure 3 shows part of the local views of these two systems. It should be noted that in this paper we are not concerned with the mapping between the local views and the data in the local pre-existing applications, but we assume that this mapping is provided by the local systems and is completely hidden from the integrated system.

2.2. Overview of the Language

Our rationale for the functionality that the multilanguage should provide is as follows:

1. The language should be consonant with the Distributed Object Architecture.

2. The language should be programming language-based rather than query- or database-based, to allow the integration of nondatabase systems. Programming-based approaches are also favored by the fact that a multilanguage is a high-level language superimposed on pre-existing systems. The actual storage and retrieval of data is supported by the underlying systems. So the main concern of the multilanguage is not storage but the provision of flexible means of reusing existing mechanisms.

3. Object-oriented programming languages are used for manipulating transient data whose lifetime coincides with the execution of a program. A multilanguage should be capable of combining and manipulating persistent pre-existing data. To this end, in Section 4 we extend the class constructors to define *virtual* classes. Data defined during the execution of a multilanguage program are also persistent unless otherwise indicated.

4. The language should provide for combining information from the component databases. View-definition facilities have been used in database languages to provide a different view of stored data to database users. We adopt this concept from database systems. The view mechanism is detailed in Section 5.

5. Although the language is programming-based, since the integrated system includes database systems, efficient data access is very important, and nonprocedural operations (such as search and select) should be efficiently supported. This can be accomplished without sacrificing the power of the programming language approach, by considering database systems as a special class in the class hierarchy of the system. This technique is explained in Section 5.

6. Multilanguage programs express complex tasks that access and update various databases. The language should provide constructs for specifying the control flow of each task and the interaction among concurrently executing tasks. Furthermore, we propose that each basic method should include informations about its consistency requirements and atomicity properties. This leads to a close connection between transaction management and the multilanguage. This issue is explored in Sections 5 and 6.

7. Our design is based on the principle of minimality, we seek to define as few new constructs as possible.

Many object-oriented programming languages distinguish between the state of an object, which is modeled by its instance variables, and its behavior which is modeled by its methods. We shall not make this distinction between the state and the behavior of an object but use a single construct, called a *method*, to model both. An instance variable is modeled by a pair of *set* and *get methods* [38], where set assigns a value to the variable and get returns its value. This approach leads to a model with fewer constructs and thus minimizes the number of possible conflicts during the integration of different local views. More importantly, it offers increased flexibility to the integrator by permitting the state of an object to be redefined in the global schema. For example, assume an object of a class named $employee$. An $employees$'s $salary$ may be represented in dollars in one component database and in German marks in another. We can define an appropriate method in the global view that performs the necessary transformations based on the daily rate of exchange between these monetary units. In contrast, if $salary$ is represented as an instance variable, there is no straightforward solution to the above conflict.

3. Constructs to Support Database Functionality and Integration

Using as a multilanguage a language that is an extension of a programming language allows us to manipulate both database and nondatabase resources uniformly thus alleviating problems of impedance mismatch. On the other hand, the language should be customized to accommodate the manipulation of sets of data and to support database integration.

3.1. Class Extension

A class is a template for creating objects with a specific behavior and structure and is not directly related to the real objects whose structure and behavior it models. However, in a database system, we need a language construct to model sets of objects. For that purpose, we define the notion of a *class extent*. The extent of a class defines how a class is populated. Informally, the class extent is the set of objects that *belong to* the class. A natural way to define the *"belong-to a class"* relation is as the set of all objects that are instances of that class. However, this approach proves to be restrictive. For example, assume a simple library database where the books in each department's library are modeled as a class. Two such classes could be *CSLibrary-Book* and *MathLibrary-Book*. All these classes are subclasses of the class *UnivLibrary-Book*, which has no instances. To find a book, a user must name all existing libraries, although it is intuitively more logical, to designate the extent of *UnivLibrary-Book* as the target of his query. This leads us to the following definition of the belong-to relation and of the extent of a class:

Definition. [belong-to] An object *"belongs-to a class"* if it is an instance of that class or of any of its subclasses.

Definition. [extent] The extent of a class includes all objects that belong to the class.

The *belong-to relation* is similar to the *members-of* relation [9], [29]. Under the above definitions, the extent of the class *UnivLibrary-Book* is the union of the extent of all its subclasses. One can therefore express the above request as a query with the extent of *UnivLibrary-Book* as its target. Definition 1 is valid, since an instance of a subclass has at least the behavior of its superclass. The implication of the above definition is to impose a hierarchy of the extents that parallels the hierarchy of their classes. If class A is a subclass of class B, then the extent of class A is a subset of the extent of class B. We should stress that the class hierarchy is of a semantic nature, whereas the extent hierarchy is an inclusion relationship between sets of objects. Furthermore, although the definition of a class remains the same, the extent of a class changes with time as new instances are created or deleted. The definition of a class extent is implemented by associating each class that expresses a database item with a collection class as explained in Section 4.1.

3.2. Virtual Classes

Most object-oriented languages, though rich in facilities for structuring new objects, lack some necessary mechanisms for grouping already-existing objects. Classes are defined as templates for creating new objects and no mechanism for grouping existing objects is supported. In the case of database integration, grouping existing objects is essential. To this end, we define virtual classes. A virtual class is a template that describes already-existing objects. Once a virtual class is created is treated as any other class. The classes used in the definition of a virtual class are called *base classes*.

Rather than defining new constructs, we extend the existing ones. We wish to maintain the role of classes, as a mechanism for grouping objects with common structure and behavior. Similarly, subclassing and superclassing should maintain their roles as mechanisms for incremental definition and sharing. Those principles distinguish our work from related approaches to the definition of views for object-oriented databases. Typically, classes are defined as templates by simply specifying their structure and behavior. The instances of the class are explicitly created after the definition of the class. To create a virtual class, however, we must define:

1. the initial members of the class (class extension). For this purpose, we may need to use a query language to select objects belonging to different classes or even to different component systems;

2. the structure and behavior of the members of the class, which in our approach are both represented as methods; and

3. the position of the new virtual class in the class hierarchy.

The only means of creating new virtual classes is by applying one of the class constructors. Classes that result through subclassing or superclassing take the corresponding position in the class hierarchy. However, no specific position in the hierarchy is assumed for classes created by the new class constructor.

As a final note, since most object-oriented languages do not provide a superclass construct nor do they support the corresponding concept of inheritance from the subclass to the superclass (generalization or upwards inheritance [30], [36]), we first define the notion of a superclass constructor before proceeding to define virtual classes.

Definition. [superclass constructor] A class A, defined as a superclass of one or more already defined classes B_i, inherits (upwards inheritance) all the common methods of the B_is. A's extent class is a collection of the members of all B_is.

3.2.1. The <New Virtual Class> Constructor

We shall make a distinction between two different cases of new virtual classes, new virtual classes that directly correspond to one basic class in a local view (importation) and new virtual classes that are created by combining instances belonging to different basic classes (derivation).

Importation. The new class incorporates data from other databases via import statements.

```
     <class> A <import> B <from> database-name
```

Once a class is imported, its definition and instances become visible to the user of the new class. Part of the imported data can be hidden by using explicit hide commands.

Example: The following definition:

```
    <class> CS-Book <import> Book <from> CS-Library
                     <hide> set-ISBN
```

creates an imported class *CS-Book* which corresponds to the class *Book* of the *CS-Library* but does not permit the user to modify the *ISBN* of a book. ■

Derivation. New classes are defined by queries on existing ones. The new classes have no implicit relations with existing classes and no specific position in the class hierarchy as regards the base classes. Their extension is defined as the result of a query that selects instances of the base classes. Finally, their methods are defined explicitly and can use methods defined in the base classes.

```
 <class> A <derive> B1 <from> database-name1
                           <where> Predicate1 using B1’s methods
                   B2 <from> database-name2
                           <where> Predicate2 using B2’s methods


                   Bj <from> database-namej
                           <where> Predicatej using Bj’s methods
           <methods>
                 ( ... method definition ... )
```

3.2.2. The <Virtual Subclass> Constructor

A virtual class which is defined as a *subclass* inherits all the methods of its superclasses. Methods can be redefined in the virtual subclass, and new methods may be defined. In the case of multiple inheritance, *imaginary objects* combine the behavior of two or more local objects, based on the value of some common method.

```
     <class> A <virtual subclass> B, C
       <on> Predicate
```

3.2.3. The <Virtual Superclass> Constructor

A virtual class defined as a *superclass* inherits the common behavior and structure of its subclasses (*upwards inheritance*). Methods may be redefined in the superclass. The instances of the superclass are imaginary objects that consist of the common part of the structure and behavior of the objects of the subclasses.

```
    <class> A <virtual superclass> B, C
```

In the case of object-oriented languages where subclassing is used for subtyping, some restrictions must be enforced on the type of arguments and on the type of the return values of all methods defined by inheritance in the virtual class. These restrictions should permit every object of a subclass to be used in any context where an object of any of its superclasses could be used.

4. Integration through Views

Views are an important part of any database language which provide a means of defining a virtual database on top of one or more existing databases. Views in the global system, as opposed to warehouses [42], are not stored, but are recomputed for each method that refers to them. Views play an important role in the creation of the integrated system, since they are used to define a global view that includes all information stored in the component systems.

Definition. [object-oriented view] An object-oriented view is a set of virtual classes.

We will first show how the class hierarchy of the system can be used to provide a uniform way of mapping database resources to basic classes and associative queries to basic methods. Then, we will detail the view mechanism and how it provides a means of expressing relationships among basic classes and resolving any potential conflicts.

4.1. The Use of the Class Hierarchy to Map Database Systems

Figure 4 presents part of the class hierarchy of the global view. All classes are subclasses of a system-defined class called *Object*. The system provides a predefined class called *Database-System*. All database systems that participate in the integration are modeled as descendants of this class. This class provides a set of predefined methods for manipulating component database systems such as data structures for storing directory information for these databases.

The classes of the global system are virtual classes created by applying appropriate class constructors to the basic classes. All imported basic classes of a component database are indirect or immediate subclasses of the class *Database-Class*. For each element class in the tree rooted by *Database-Class*, the system produces a corresponding hierarchy of classes that represent the extent of the element class. The classes that represent the extent

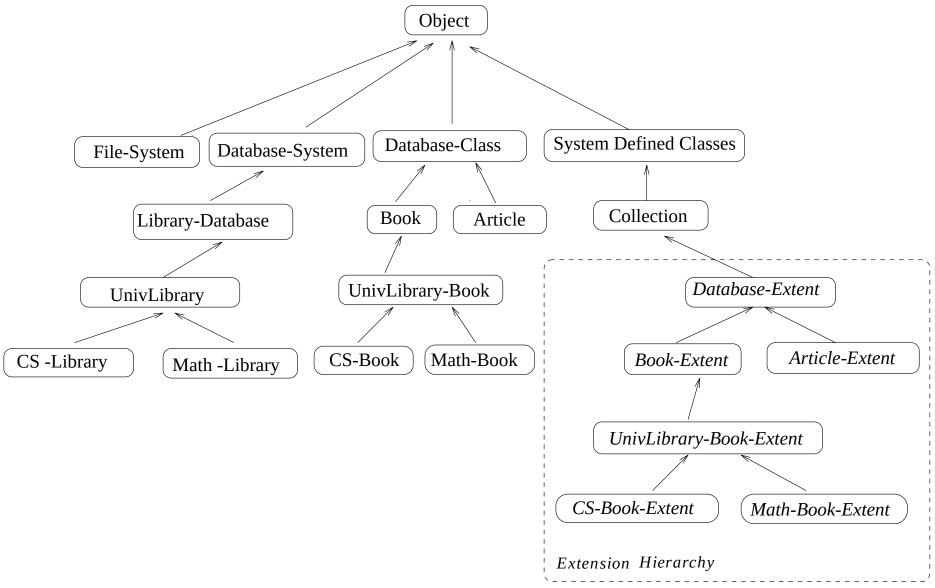


Figure 4. Part of the class hierarchy.

of a class are descendants of the system-defined class called *Collection*. A *Collection* is an unordered list of objects.

In Figure 4, for example, each instance of the class *Book-Extent* is a collection of objects of class *Book*. Predicate-based queries are modeled as messages to the appropriate class extent and return as result subsets of this extent. The result of such queries are defined as subclasses of the associated class extent. For instance, a query to select all books written by an author X should be modeled as a method of the objects of the class *Book-Extent*. The system provides predefined methods for accessing members of both the *Database-Class* and its extension.

- Predefined basic methods of the *Database-Extent* include methods for searching all the elements of the collection, methods for selecting an element from the collection, insertion, deletion, and other set-based operations (such as Union and Difference).

- Predefined basic methods of the *Database-Class* include operations to get and set an attribute and create an instance.

Other operations are defined using these basic operations. The advantage of this mechanism is that it provides a default mechanism and library support for basic database functionality. In Figure 5, some of the basic methods of the classes in the hierarchy are depicted.

```
<Class>  Database-Class                   <Class>  Database-Extent
    <Subclass of>  Object                        <Subclass of>   Collection
     <Methods>                                   <Methods>
             Get(attribute)                              Insert(object)
             Set(attribute)                              Select(object with attribute-Y)
                  •                                           •
                  •                                           •

                  •                                           •
<End>                                    <End>


 <Class>  CS-Book
      <Import> Book   <from>  CS-Library
      <Methods>
             Print-abstract
             Get-all-references-to-X
             Check-out-book

                  •
                  •

                  •
 <End>
```

Figure 5. Class definitions of some classes.

4.2. Using Views to Resolve Conflicts and Express Interschema Relations

If no relations pertained among the imported basic classes then the global view would simply be the union of these classes. Unfortunately, the same concepts may be represented in different databases and furthermore, due to heterogeneity, these concepts may be represented in variant forms. Schema translation alleviates the problems that arise from the use of different data models at each component database. Even when all component schemas are represented by the same data model, however, conflicts may still occur.

Table 1(a) provides a taxonomy of the possible conflicts when object-oriented models are used. First, the same entity may be represented by different objects. Then, several other conflicts are possible along the various system levels. At the schema or representation level, different names or structural constructs may be used to represent the same entity. At the semantic level, the same entities may have different underlying interpretations. Finally, at the data or storage level, the same entities may have been assigned different values.

Furthermore, for integration to be successful, it is crucial to identify not only the set of common concepts but also the set of different concepts in different systems that are mutually related by some semantic properties. These properties are called *interschema properties* [7]. They are semantic relationships which hold between a set of objects in one schema and a different set of objects in another schema. For reasons of completeness, these relations should be represented in the global view. Table 1(b) outlines types of interschema relation that can be easily expressed using object-oriented mechanisms. Other relations are considered arbitrary and treated on a per case basis.

Type		Definition	Example
Identity Conflict		The same concept is represented by different objects in different local databases	The same book is stored in both the CS and the Math Library
Schema Conflict	Naming	Homonyms: the same name is used for different concepts	Article for a journal article and for a newspaper article
		Synonyms: the same concept is described by different names	References and bibliography
	Structural	The same concept is represented by different constructs of the model, e.g., by a method in one and by a class in the other, or	Author is a class in one and a method in the other
		although the same concept is represented by the same construct, the classes have different methods or the methods have different parameters or return values	Book has an attribute keyword in one class but not in the other
Semantic Conflict		The same concept is interpreted differently in different databases	Conference is a refereed conference in one but not in the other
Data Conflict		The data values of the same entity are different in different component databases	The same book appears to have different authors

(a)

Type	Example
Aggregation	A book in one library has as parts articles stored in the other
Specialization	An article of a specific mathematical journal is a special case of a journal
Generalization	Books in the Math and the CS library are all books
Arbitrary	Some books and articles of interest to a particular user

(b)

Table 1. (a) Taxonomy of the possible conflicts between two local views (b) Interschema relations.

Type		Resolution
Identity Conflict		Use a user-defined same method
Schema Conflict	Naming	Use renaming operators
	Structural	Ad-hoc combinations of the virtual class constructors
Semantic Conflict		Define an appropriate method in the virtual class that combines any conflicting values or semantics
Data Conflict		

(a)

Type	Operation
Aggregation	New method
Specialization	Subclassing
Generalization	Superclassing
Arbitrary	New class

(b)

Table 2. Using the view mechanism (a) to resolve conflicts and (b) to express interschema relationships.

Tables 2(a) and 2(b) show how the view mechanism can be used to resolve conflicts and to express interschema relations. These tables are meant to be indicative of the general methodology rather than to provide a detailed coverage of the topic.

We define a special method, called *same*, that defines the conditions under which two instances are considered equal. The method *same* is defined at the class *Database-Class* and can be overloaded at its descendants. Renaming synonyms or homonyms suffices to resolve naming conflicts. Structural conflicts, however, are harder to cope with. In general, a combination of class constructors can be applied to the conflicting virtual classes, to put them in compatible forms. Semantic and data conflicts can be resolved by defining appropriate reconciliation methods. For instance, if a book appears to have two different authors in two local databases, a properly defined method should report the problem and return both authors as an answer.

To express aggregation between classes, a pair of *get* and *set* methods can be defined in a virtual class A. These methods will return and modify the objects of A's component class. Specialization and generalization are modeled by subclassing and superclassing respectively. Finally, to express arbitrary relations between classes, we can use the new class constructor to combine them in various ways.

4.3. Identity and Resolution Problems

Object-oriented systems associate a unique identifier with each object upon its creation. Accordingly, upon creation of an imaginary object, an identifier must be associated with it. Care must be taken to ensure that an imaginary object is assigned the same identifier at each invocation. Moreover, the identity of the imaginary object should be modified if the local objects from which it is constructed are updated. We adopt the solution of defining the identifier of an imaginary object as a function of the identifiers of those local objects.

In an object-oriented system, a method defined in a class may be redefined in its subclasses, resulting in method overloading. The default resolution method adopted by the object-oriented systems states that, when a method is applied to an object the most specific method from those applicable to the object is selected. Thus, when a method m is applied to an object O, first the class whose instance O is, is searched to find the definition of m. If m is not found, then the immediate superclass is searched, and the process continues in this fashion. The introduction of virtual classes complicates the resolution problem, when a virtual class A is defined as a superclass of existing classes. In that case, the default resolution method always selects the most specific method, i.e., one defined in one of A's subclasses, even when the user wants a more general method defined in A to be selected. To cope with this problem, we allow the user the possibility to override the default resolution mechanism by explicitly specifying which of the applicable methods should be used.

5. Constructs to Support Concurrency and Consistency

The multilanguage provides constructs for supporting concurrency in the execution of each program and among programs. This is achieved by nonblocking message calls. There is also a need for synchronization among concurrently executing programs which is attained through defining dependencies between execution states and by appropriate message exchanges. Finally, concurrent executions should not violate database consistency. Consistency is expressed using assertions and is enforced by ensuring semantic serializability.

5.1. Consistency Assertions

In each local database system, data are related by a number of integrity constraints that express real-world restrictions on the allowable values they can take. This is formalized by incorporating the consistency requirements of each class into its interface. For each basic method, an assertion, called a *precondition*, expresses the input requirements, and a second assertion, called a *postcondition*, expresses the properties ensured by any call to the method. Beyond the precondition and postcondition of each of its individual basic methods, a class is also characterized by its *invariant*, which characterizes the consistency of the class. The class invariant applies to all its methods, being in effect added to both the pre- and post-conditions. Formally,

Definition. Methods are defined as Hoare triples. Each method $\mathrm{op}_i(a, b)$ on an object o, where o is an instance of a class C, a is an input parameter, and b is an output parameter,

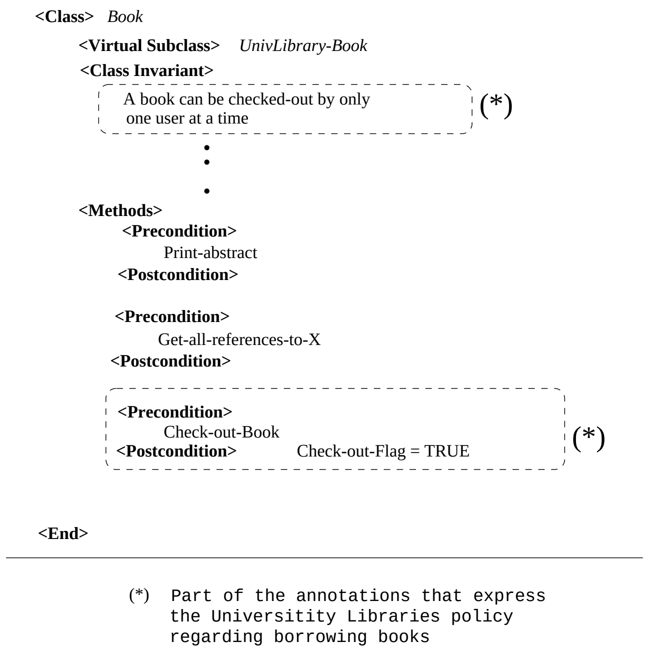

<Class> Book
<Virtual Subclass> UnivLibrary-Book
<Class Invariant>
A book can be checked-out by only one user at a time
(*)
<Methods>
<Precondition>
Print-abstract
<Postcondition>
<Precondition>
Get-all-references-to-X
<Postcondition>
<Precondition>
Check-out-Book
<Postcondition> Check-out-Flag = TRUE
(*)
<End>
(*) Part of the annotations that express the Universitity Libraries policy regarding borrowing books

Figure 6. Extending the definition of a class with consistency annotations.

is annotated as follows: $\{P_i\}$ $\mathrm{op}_i(a, b)$ $\{Q_i\}$, where $\{P_i\} \equiv Precondition_i(o) \wedge INV_C$ and $\{Q_i\} \equiv Postcondition_i(o) \wedge INV_C$. $Precondition_i(o)$ and $Postcondition_i(o)$ are predicates on o's local state and on a and b and INV_C expresses the integrity constraints of the class.

Figure 6 shows such an example. Our *only assumption* is that the basic operations are implemented atomically at each local site and in a manner that preserves the consistency of the local site that is the INV_C for all classes C in that site. Any call to a basic method may assume that the invariant is initially satisfied (i.e., it may expect to find the object in a consistent state), but it must maintain the invariant (i.e., it must leave the object in a consistent state).

During view definition, the consistency assertion of the basic classes must be respected in two ways. First, the invariant of a subclass must be stronger than the invariant of its superclass. Second, when a method is redefined, its precondition can only be weakened and its postcondition can only be strengthen.

Consistency annotations offer a natural and convenient formalism for incorporating the consistency requirements of a persistent local object into its interface. Furthermore, assertions provide the basis for semantic-based concurrency and atomicity control, as will be shown in Section 6.1.

5.2. Task Concurrency and Synchronization

A user interacts with the multidatabase by invoking programs called tasks. Programs are special objects of the multidatabase that are instances of the system-defined class $Program\_Class$. Tasks are activated by an initialize message and are executed in the context of their own local data. Specifically, tasks have instance variables (modeled as explained in Section 2.2) and a special method called *Compute* that encodes the intended computation. There are at least three types of instance variables: (a) instance variables that model local data, (b) instance variables that express the state of each method of a task and (c) a history variable that records the execution of messages, in particular method invocations and dependencies, and is used for recovery. The methods that access the instance variables of a task are called *primitive*. *Compute* is the top-most method of the task object and uses only basic and primitive methods.

Upon activation each task creates a Task Transaction Manager (TTM) to coordinate its execution. TTMs, like programs, are objects, instances of a special class of the multidatabase called TTM_Class. Concurrency in the execution of a task is achieved through asynchronous message calling. In addition, tasks submitted by different users may be executed concurrently.

5.2.1. Intratask Concurrency and Synchronization

Concurrency inside a task is achieved through asynchronous message invocations. Synchronization mechanisms are necessary to control the interaction among methods. Intratask synchronization is based on defining a set of dependencies on the ordering of the execution of methods [34]. Such dependencies are called structural and are expressed in terms of controllable states of the methods of the $Compute$ of the task. In terms of basic methods, the states that can be controlled is the completion (commit or abort state) and the submission (begin state) of the method. The actual execution time of a basic method is under the control of the corresponding local database. In addition, some database systems provide a prepare-to-commit state that indicates that a basic method has completed execution and its results are about to become permanent. Finally, we distinguish two types of commitment depending on whether the result of the execution is semantic failure or semantic success.

Definition. [structural dependency] A structural dependency SD is a triple (C, $\mathcal{M}$, $\mathcal{S}$), where C is a specification, $\mathcal{M}$ is a set of methods, and $\mathcal{S}$ is a set of controllable states of the methods $M \in \mathcal{M}$. For a basic method M, a controllable state in $\mathcal{S}$ may be commit (semantic failure or semantic success), abort, prepare-to-commit, or submit, and for a primitive method it can also be execute.

We distinguish three types of structural dependency based on the form of C: existence, order, and real-time dependencies in accordance with the primitives defined in [22].

Definition. [existence] In an existence structural dependency ($C, \mathcal{M}, \mathcal{S}$), C has the following form: if M_i enters state s_i, then M_j must enter state s_j for M_i, $M_j \in \mathcal{M}$ and s_i, $s_j \in \mathcal{S}$.

Special cases of existence structural dependencies include critical, contingency, and compensation methods. *Critical methods* are methods that, when aborted (or fail semantically) cause the entire task to abort (or fail semantically). *Contingency methods* are methods that are executed as alternatives when a task fails semantically. *Compensation methods* are methods that are executed to semantically undo the effect of a committed method when some other method aborts.

Definition. [order] In an order structural dependency $(C, \mathcal{M}, \mathcal{S})$, C has one of the following forms: M_i can enter state s_i only after M_j has entered state s_j, or M_i cannot enter state s_i after M_j has entered state s_j for M_i, $M_j \in \mathcal{M}$ and $s_i, s_j \in \mathcal{S}$.

Ordering structural dependencies can be used to express data flow dependencies, for instance that M_i reads data produced by M_j.

Definition. [real-time] In a real-time structural dependency $(C, \mathcal{M}, \mathcal{S})$, C specifies a requirement for the real time submission or completion of the methods in $\mathcal{M}$.

The synchronization mechanism of the language is based on (a) allowing direct access to the state of each method, and (b) employing a special method called *form_dependency*. A state is an instance variable of the associated task and can be modified by the Task Transaction Managers of the sender or the receiver of the method. The *form_dependency* method is sent to the Task Transaction Manager and takes as arguments the specification of the dependency and the names of the associated methods and states.

5.2.2. Intertask Communication and Synchronization

Concurrency at the intertask level is achieved by allowing more than one task enactment. Traditional concurrency control does not allow any form of interaction between tasks initiated by different users, since serializability enforces the isolation of each task. In particular, each user's interaction with the database is implemented as an ACID transaction. The execution of each program is isolated from the execution of all other programs, and is also formulated as atomic, consistent, and durable. In a multidatabase system, however, these interactions model long-lived, complex activities that scan more than one autonomous system. Ensuring the ACID properties is thus unrealistic. The ACID model is also inappropriate for environments where there is a need for cooperation between users, such as in workflow tools and in software development, computer-aided design and other CSCW applications.

There is a need to control the interaction between concurrently executed tasks and also preserve the consistency of local databases. The multidatabase language provides communication and synchronization constructs for expressing and controlling such intertask interactions. This intertask synchronization control can be either through a message passing mechanism or through shared memory. In the latter case, interaction relies on the observation of partial changes in object states which are caused by other tasks.

Message passing provides an explicit control of the visibility of a task. Both asynchronous and synchronous messages from a task to another task should be supported.

```
<send> Object <to> Task-name
```

```
<receive> Object <from> Task-name
```

In contradistinction, a task can define points in its execution where other tasks are allowed to observe its partial results as changes in the database state. This may be accomplished by explicitly defining break [17] or permit [11] points within the execution of a task where other tasks are allowed to interleave.

Definition. [breakpoint] A breakpoint B of a task T is a triple $(B_s, B_e, \{(T_i, M_j)\})$ where $\{(T_i, M_j)\}$ is a set of pairs of methods and tasks, and B_s, B_e are pairs of methods and controllable states in T which allow members of $\{(T_i, M_j)\}$ to be executed between states B_s and B_e of T.

Another form of intertask communication is *delegation*, where a task delegates responsibility of the execution of a method to another task.

Definition. [delegation] A delegation of a task T is a pair (M_i, T_j) that denotes that the method M_i invoked by T will be executed as part of task T_j.

To denote breakpoints and delegation we use two special methods called $breakpoint$ and $delegation$. A $breakpoint$ method is sent to the Task Transaction Managers of the tasks whose methods' call are allowed to interleave. The $delegation$ method is sent to the Task Transaction Manager of the task to which the execution of the method is delegated.

6. Semantic-Based Concurrency Control

We outline how assertions and workflow specifications are used for concurrency control.

6.1. Assertion-Based Definitions of Commutativity

The consistency assertions are used to define when two operations conflict. Conflicting methods are defined as methods that do not commute; that is, as methods whose relative order of execution affects the result of a task. There are different definition of commutativity based on the type of global concurrency control and recovery mechanism used. In the following, we illustrate how some proposed definitions of commutativity can be expressed using the consistency assertions. $Wp.op_j.P_i$ stands for the weakest precondition for P_i to hold after the execution of op_j. Specifically, we consider noninterference [28], left-commutativity [2], [19], and forward and backward commutativity [40]. The list of definitions is intended to be illustrative rather than exclusive.

Definition. Let two basic methods $\{P_i\}\ op_i\ \{Q_i\}$, and $\{P_j\}\ op_j\ \{Q_j\}$.

- $P_i \wedge P_j \Rightarrow wp.op_j.P_i$ (noninterference)
  The intuition is that, P_i is not interfered with, if it is not invalidated by the execution of op_j, and thus op_j can be interleaved at this point.

- *op_j left-commutes with op_i* if $(P_i \wedge wp.op_i.P_j \Rightarrow P_j \wedge wp.op_j.P_i)$.
  The intuition is that in any execution where op_j immediately follows op_i, op_i and op_j can be swapped.

- op_i and op_j *commute forward* if $(P_i \wedge P_j \Rightarrow wp.op_j.P_i \wedge wp.op_i.P_j)$.
  The motivation is that in a state where both op_i and op_j can be executed we can "push op_i forward over op_j" and vice versa.

- op_i and op_j *commute backward* if $P_i \wedge wp.op_i.P_j$ eq $P_j \wedge wp.op_j.P_i$.
  The motivation is that in any state where $op_i op_j$ represents a valid order of execution we can "push op_j backwards over op_i".

Figure 7 shows how information about the atomicity and commutativity of each basic method can be included in its definition.

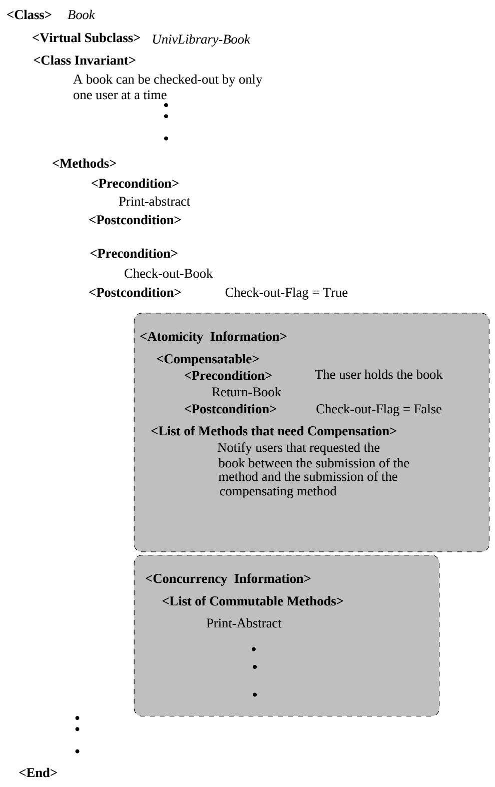
An assertion-based definition of commutativity supports extensibility. When a new basic method is added to the interface of a local database system, the only concurrency information that the system is required to provide is the assertions of the methods. Its commutativity with the other methods can then be computed.

6.2. The Transaction Model

A commutativity relation is defined for each pair of methods. In a closed-nested transaction model, such as that in [19], conflicts among primitive or local methods result in conflicts among the composite methods from which they are invoked. In open-nested transactions [26], there is no such implication. Although, we assume for the clarity of presentation, an open-nested transaction model, most of the following definitions and protocols translate easily to the closed-nested situation by for instance using such techniques as hierarchical timestamps [19].

The computation of a task is extended to include the methods that are delegated to it. In particular, the execution of a task is modeled as a sequence of its breakpoints and of state transitions of the methods that it invokes or are delegated to it. Thus, a *task execution* is a pair $(\Sigma, <)$ where Σ is a set of events and $<$ is a partial order. An event in a task execution is either a breakpoint B of the $Compute$ of the task or a pair (M, S) where M is a nondelegated method invoked by the $Compute$ method of the task or a method delegated to it and S is a state of M. The partial order $<$ is such that for all non commutable methods M_i and M_j either $(M_i, E) < (M_j, E)$ or $(M_j, E) < (M_i, E)$, where E stands for the execution state of the method. A task execution is *well-structured* in terms of a set D of structural dependencies if the order $<$ of its events does not violate any of the dependencies in D.

A schedule is an interleaved execution of methods and breakpoints of a set of tasks. Formally, a *schedule* of a set $\{T_1, T_2, \ldots, T_n\}$ of task executions $T_i = (\Sigma_i, <_i)$ is a pair $(\Sigma, <_h)$ where $\Sigma = \bigcup \Sigma_i$ and $<_h$ is a partial order such that: (1) if, for any s_k and $s_l \in \Sigma_i$, $s_k <_i s_l$ then $s_k <_h s_l$, and (2) for all non commutable methods M_i and M_j of two different task executions either $(M_i, E) <_h (M_j, E)$ or $(M_j, E) <_h (M_i, E)$.



```
<Class>  Book
    <Virtual Subclass>  UnivLibrary-Book
    <Class Invariant>
        A book can be checked-out by only
        one user at a time
            •
            •
            •
      <Methods>
          <Precondition>
              Print-abstract
          <Postcondition>

          <Precondition>
              Check-out-Book
          <Postcondition>        Check-out-Flag = True
            <Atomicity  Information>
              <Compensatable>
                  <Precondition>       The user holds the book
                      Return-Book
                  <Postcondition>      Check-out-Flag = False
              <List of Methods that need Compensation>
                      Notify users that requested the
                      book between the submission of the
                      method and the submission of the
                      compensating method

            <Concurrency  Information>
              <List of Commutable Methods>
                      Print-Abstract
                          •
                          •
                          •
        •
        •
        •
  <End>
```

Figure 7. Extending the definition of a class with atomicity and commutativity information.

The projection of a schedule S on the local data of task T_i is the schedule that results if we exclude from S all but the primitive methods on data of T_i and the breakpoints, if any, that immediately precede each of them. Similarly, the projection of a schedule S on the local data of a local database DB_i is the schedule that results if we exclude from S all but the basic methods on data of DB_i and the breakpoints, if any, that immediately precede each of them.

A *step* of a task execution T is a subset of T that includes exactly the events between two consecutive breakpoints in T. We use a modified model of step-wise serializability [17] that allows for efficient, i.e., polynomial, serializability testing [3]. We say that a method M_2 directly *depends on* a method M_1 if $M_1 <_h M_2$. The *depends on* relation is the transitive closure of the directly depends on relation. A schedule is *relatively serial* if for all T_i and T_j, if a method M of T_i is interleaved with a a step of T_j then M does not depend on any operation of the step and no method of the step depends on M. A relatively serial schedule is *correct* if, for any method M of a task T that interleaves a step of T_j starting with breakpoint $(B_s, B_e, \{(T_i, M_j)\})$, it holds (T, M) $\in \{(T_i, M_j)\}$.

Then, a schedule is correct if (1) all its tasks are well-structured, and (2) it is conflict equivalent to a correct relatively serial schedule. It is easy to prove that,

THEOREM 1 *If each task and local database projection of a schedule S is conflict equivalent to a correct relatively serial schedule, then, if there is an order $<_o$ consistent with the serialization orders assumed by each projection, then S is conflict equivalent to a correct relatively serial schedule with order $<_o$.*

6.3. Transaction Management

Transaction management is performed at three levels (see Figure 8): (1) at a local level by the pre-existing transaction managers of the local databases, (LTMs), (2) at a task level by task transaction managers (TTM), and (3) at a global level by distributed transaction managers located on top of the LTMs at each database system (DGTMs). Each TTM is created upon the activation of a task to control its execution. Each DGTM receives methods from the various TTMs, schedules them to control concurrency and intertask synchronization and in turn submits them at the corresponding LTMs. LTMs at each site ensure that each basic method is executed as an ACID transaction.

The above approach differs from traditional multidatabase transaction management techniques in many ways. First, the interface between the integrated system and the local systems is through basic methods. By doing so, we are in accordance with DOM standards, encapsulation is achieved, and semantic serializability can be utilized. Second, there is no central point of control, in the form of a global transaction manager, instead control is distributed among tasks and local schedulers. This way, bottlenecks that can seriously affect performance, especially in cases of widely distributed systems, are avoided.

Upon creation, each task receives a *timestamp*. The timestamp is defined to be a combination of the value of the clock and the user's id. The timestamp of a task corresponds to its global serialization order. To handle breakpoints, the commutativity relation between the methods that follow the breakpoint and the methods specified in the breakpoint is changed so that for the duration of the breakpoint, the methods commute even if they normally do not. Delegation is taken into consideration directly in the definition of a task execution by making a delegated method part of the execution of the task it was delegated to.

Each TTM has two basic responsibilities. First, it coordinates the execution of its task. It decomposes the $Compute$ of its associated task to basic and primitive methods, and takes care of submitting these methods to the appropriate sites. It also ensures that its task execution is well structured. To enforce the specified structural dependencies, the TTM can either use graph-based methods [11] or automata-based techniques [5]. Second, each TTM produces correct relatively serializable executions on its local data based on the timestamp order.

Each DGTM produces DB_i correct relatively serializable schedules consistent with the timestamp order. We now describe the submission of a primitive method from a DGTM to an LTM. To execute a composite method, each TTM can use techniques such as the semantic-based locks of [26]. Each DGTM possesses a variable called a *logical ticket* (LT) and a list of the timestamps of all basic methods that have been submitted to the

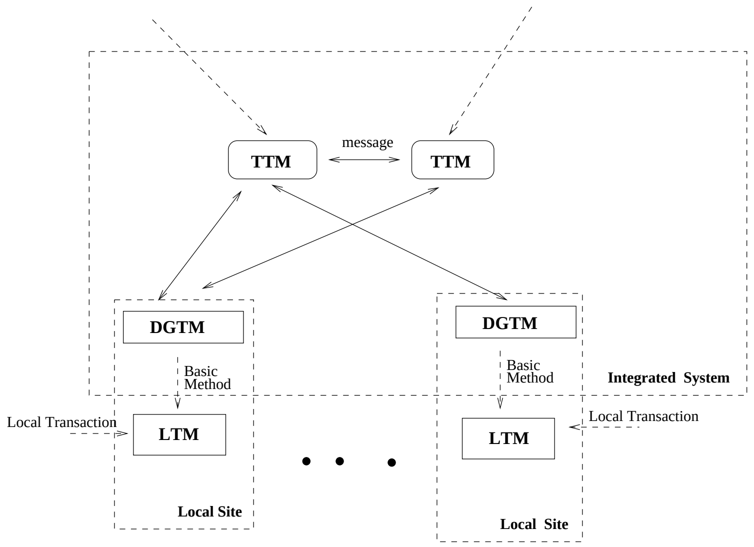


Figure 8. Task execution.

site. A method that does not commute with a submitted method is not allowed to execute concurrently with it; thus, if such a method arrives with timestamp smaller than LT, it is aborted. Two commutable methods can be executed concurrently without any further control. However, if we allow submission of operations directly to the LTM, indirect conflicts among commutable methods may arise through conflicts with these autonomous operations; these can be avoided by forcing direct conflicts among them. In this case, an additional data item per database site is needed. This data item is physically stored in that site and is called a *physical ticket* (PT). This is accomplished by having each DGTM execute the following code after a commutable method M of a task T is received. The algorithm is a slight variation of [8] for the case of a database interface of primitive methods.

```
get(LT)
if (LT > T’s timestamp)
   abort(M)
else
   submit(M) to the LTM
   in a critical region
     get(LT)
     if (LT > T’s timestamp)
         abort(M)
     else
         write(PT, T’s timestamp)
         send prepare-to-commit(M) to T
         if decision taken to commit M
```

```
            set (LT, T's timestamp)
            commit(M)
        else abort(M)
```

Another important concern is the maintenance of the atomicity property of a task, which ensures that either all or none of its methods commit. This is complicated because each local site makes independent decisions on whether to commit or abort an individual basic method. To resolve this problem, a task manager can either attempt to compensate (semantically undo) a committed method or to retry an aborted method. Information per basic method includes the type of each method, such as whether it is retriable or compensatable, and, in the latter case the compensation method to be used. Compensation methods are also annotated with assertions.

7. Related Work

Reference [33] is a preliminary brief presentation of the multilanguage, in this paper we provide a more complete description. Many of the issues discussed in this paper have been the topic of previous research. However, this paper covers all aspects of the multilanguage and shows how database functionality, views and transaction management can be combined effectively using a minimum number of well-defined language constructs. In the following we briefly review related research and compare it with the proposed approach.

7.1. General Frameworks

Recently there has been a proliferation of research regarding the integration of various computing resources to create network-wide integrated distributed systems. In this section we put our work in perspective with respect to proposed, general frameworks for this problem.

Distributed Object Management Systems (DOMS). DOMSs which are the topic of much current research [27], [18], propose an object-based approach to distributed computation to overcome the shortcomings of client-based computation. The main focus of this research is on specifying the common interface of each local system for supporting interconnectivity and extensibility. Instead, in this paper, while being in compliance with the general principle of DOMS, we focus on the special case of database systems, and discuss how this interface can be customized for local database systems and how integration of information and support for concurrency can be achieved using language constructs.

Open Object-Oriented Database Systems (Open OODB). In terms of proposals for open object-oriented database system architectures [41], our approach focuses on the role of the language in the general architecture, especially in terms of supporting the integration of existing systems.

Megaprogramming. Megaprogramming or programming in the large [45] is a proposal for a future technology which will support programming with large modules that capture the functionality of services provided by pre-existing systems. In that context, the multilanguage corresponds to the language used to define, integrate and program with these modules. Our multilanguage provides a concrete example of such a technology and thus can be seen as a step towards a better understanding of this issue.

Mediators. In [44], mediators, were introduced as an intermediate layer between the user applications and the systems that store the data resources. In this framework, the local and global views can be considered as the mediators and the multilanguage as the specification and interface language of those mediators.

7.2. Specific Issues

In this section, we compare research related to specific characteristics of the multilanguage.

Database Functionality. There is no consensus yet on a standard object-oriented data model. In this paper, instead of defining yet another data model, we show how the standard characteristics of object-orientation as manifested in most programming languages can be customized to model sets and associative queries. A similar notion of a class extension is implemented in the object-oriented database language ORION [6]. In this paper, we extend this notion for the case of heterogeneous database systems and show how it can be used as part of the class hierarchy of the global view.

View Definition. There are as many approaches to defining object-oriented views [14], [25], [4], [43], [21], [13], [35], [1], [20], [10] as different object-based data models. The distinctive characteristic of our approach is that we extend an object-oriented *programming* language, where the only language primitive for grouping sets of objects is the class. Based on that assumption, views in our approach are virtual classes whose extent is defined by queries. To define virtual classes, instead of defining new class constructors, we extend the existing ones (namely, the new class, subclass, and superclass constructors). In that sense, the work most related to ours is [1]. In [1], a virtual class is defined by specifying its extension, and then its position in the hierarchy is implied. In our approach, we explicitly define a virtual class as a subclass or a superclass. Also, in this paper we show the view mechanism of our multilanguage in the context of heterogeneous system and show how it facilitates integration and transaction management.

Concurrency. There is a flurry of research on defining extended transaction models (see for example [15]). The approach of making transaction specification constructs part of the multilanguage is also demonstrated in IPL [12]. In this paper, we show how these constructs interact with the object-oriented features of the language and suggest new constructs for specifying inter-task interactions, which are not supported in IPL. Annotations have been introduced in the area of concurrent object-oriented programming languages in the Eifel language [24] and in the area of database transaction systems in [16], [39]. In this paper, we extend this work to show how it can be part of the multilanguage and how it can be used to assist global transaction management.

8. Conclusions

There is an increasing need to provide users with uniform access to information stored in many pre-existing, autonomous, and possibly heterogeneous databases. This is accomplished by combining these systems in a high-level confederation called a multidatabase [37], [32], [31]. We adopt a modular, object-based approach to the creation of a multidatabase. Each local system participating in the federation provides an interface that consists of classes and methods expressed using the constructs of an object-oriented programming language, called the multilanguage. The mapping of local resources to classes and the implementation of the local methods is the responsibility of the component systems and can be completely hidden from the federation.

The multilanguage plays an integral part in the federation. It serves as both a data definition and data manipulation language. In this paper, we have described how a typical object-oriented programming language should be extended to provide the functionality of a multilanguage. Specifically:

- Associative queries and efficient set operations must be efficiently supported. This is accomplished by automatically defining a dual concept, called extension, for each class that represents a database entity. The extension of a class is a subclass of a system-defined collection class and provides an efficient implementation of sets.
- View facilities are necessary for the integration of the basic classes. We extend the existing class constructors of an object-oriented language to provide constructors for *virtual classes*, which are classes that group existing objects. Once a virtual class is created, it is treated as any other class.
- The multilanguage closely interacts with the transaction manager in two ways. First, it includes as part of the interface of each class the consistency requirements and atomicity properties of each of its provided methods in the form of *consistency assertions*. Second, it provides *flow of control* constructs to specify the execution order of methods inside a single program as well as among different programs.

References

1. S. Abiteboul and A. Bonner. "Objects and views," in *Proceedings of the ACM SIGMOD*, 1991, pp. 238–247.
2. D. Agrawal, A. E. Abbadi, and A. K. Singh. "Consistency and orderability: Semantics-based correctness criteria for databases." *ACM Transactions on Database Systems* 18(3), pp. 460–486, September 1993.
3. D. Agrawal, J. Bruno, A. Abbadi, and V. Krishnaswamy. "Relative serializability: An approach for relaxing the atomicity of transactions," in *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, 1994, pp. 139–149.
4. R. Ahmed, J. Albert, W. Du, W. Kent, W. Litwin, and M-C. Shan. "An Overview of Pegasus," in *Proceedings of the RIDE-IMS*, April 1993, pp. 273–277.
5. P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. "Specifying and enforcing intertask dependencies," in *Proceedings of the 9th International Conference on Very Large Database Systems*, 1993, pp. 134–144.

6. J. Banerjee, H-T. Chou, J. F. Garza, W. Kim, D. Woelk, and N. Ballou. "Data model issues for object-oriented applications." *ACM Transactions on Office Information Systems* 5(4), pp. 3–26, January 1987.
7. C. Batini, M. Lenzerini, and S. B. Navathe. "Comparison of methodologies for database schema integration." *ACM Computing Surveys* 18(4), pp. 323–364, 1986.
8. P. K. Batra, M. Rusinkiewics, and D. Georgakopoulos. "A decentralized deadlock-free concurrency control method for multidatabase transactions," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.
9. E. Bertino. "Integration of heterogeneous data repositories by using object-oriented views," in *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, April 1991, pp. 22–29.
10. E. Bertino. "A view mechanism for object-oriented databases." In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology—EDBT '92*, pp. 136–151. Springer Verlag, 1992.
11. A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. "ASSET: A system for supporting extended transactions," in *Proceedings of the 1994 SIGMOD Conference*, May 1994, pp. 44–54.
12. J. Chen, O. Bukhres, and A. K. Elmagarmid. "IPL: A multidatabase transaction specification language," in *Proceedings of the 1993 International Conference on Distributed Computing*, 1993.
13. J. Chomicki and W. Litwin. "Declarative definition of object-oriented multidatabase mappings," in *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada, August 1992, pp. 307–325.
14. U. Dayal and H. Hwang. "View definition and generalization for database integration in a multidatabase system." *IEEE Transactions on Software Engineering* 10(6), pp. 628–645, 1984.
15. A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.
16. A. K. Elmagarmid, Y. Leu, W Litwin, and M. Rusinkiewics. "A multidatabase transaction model for InterBase," in *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990, pp. 507–518.
17. A. A. Farrag and M. T. Ozsu. "Using semantic knowledge of transactions to increase concurrency." *ACM Transactions on Database Systems* 14(4), pp. 503–525, December 1989.
18. Object Management Group. "The common object request broker: Architecture and specification." OMG Document Number 91.12.1, December 1991.
19. T. Hadjilacos and V. Hadjilacos. "Transaction synchronization in object bases." *Journal of Computer and System Sciences* 43, pp. 2–24, 1991.
20. S. Heiler and S. Zdonik. "Object views: Extending the vision," in *Proceedings of the 6th International Conference on Data Engineering*, 1990, pp. 86–93.
21. M. Kaul, K. Drosten, and E.J. Neuhold. "Viewsystem: integrating heterogeneous information bases by object-oriented views." In *IEEE International Conference on Data Engineering*, 1991, pp. 2–10.
22. J. Klein. "Advanced rule driven transaction management," in *Proceedings of the IEEE COMPCON*, 1991.
23. F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. "Distributed object management." *International Journal of Intelligent and Cooperative Information Systems* 1(1), June 1992.
24. B. Meyer. "Systematic concurrent object-oriented programming." *Communications of the ACM* 36(9), pp. 56–80, September 1993.
25. A. Motro. "Superviews: Virtual integration of multiple databases." *IEEE Transactions on Software Engineering* 13(7), pp. 785–798, July 1987.
26. P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse. "Semantic concurrency control in object-oriented database systems," in *Proceedings of the 9th International Conference on Data Engineering*, 1993, pp. 233–242.
27. J. R. Nicol, C. T. Wilkes, and F. A. Manola. "Object orientation in heterogeneous distributed computing systems." *IEEE Computer* 26(6), pp. 57–67, June 1993.
28. S. Owiski and D. Gries. "An axiomatic proof technique for parallel programs." *Acta Informatica* 6, pp. 319–340, 1976.
29. M. P. Papazoglou and L. Marinos. "An object-oriented approach to distributed data management." *Journal of Systems and Software* 11(2), pp. 95–109, February 1990.
30. C. H. Pedersen. "Extending ordinary inheritance schemes to include generalization," in *Proceedings of OOPSLA '89*, October 1989, pp. 407–417.
31. E. Pitoura, O. Bukhres, and A. Elmagarmid. "Object-orientation in multidatabase systems." *ACM Computing Surveys* 27(2), pp. 141–195, June 1995.
32. E. Pitoura, O. Bukhres, and A. Elmagarmid. "Object-oriented multidatabase systems: An overview." In

A. Elmagarmid and O. Bukhres, editors, *Object-Oriented Multidatabases*. Prentice Hall, 1996.

33. Evaggelia Pitoura. "Extending an object-oriented programming language to support the integration of database systems." In *28th Annual Hawaii International Conference on System Sciences (HICSS-28)*, Maui, Hawaii, Jan. 1995, pp. 707–716.
34. M. Rusinkiewicz and A. Sheth. "Specification and execution of transactional workflows." In W. Kim, editor, *Modern Database Systems*, pp. 592–620. Addison Wesley, 1995.
35. M. H. Scholl, C. Laasch, and M. Tresch. "Updatable views in object-oriented databases," in *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, December 1991, pp. 188–207.
36. M. Schrefl and E. J. Neuhold. "Object class definition by generalization using upward inheritance," in *Proceedings of the IEEE International Conference on Data Engineering*, 1988, pp. 4–13.
37. A. Sheth and J. Larson. "Federated database systems." *ACM Computing Surveys* 22(3), pp. 183–236, September 1990.
38. D. Ungar and R. B. Smith. "Self: The power of simplicity," in *Proceedings of OOPSLA '87*, October 1987, pp. 227–242.
39. H. Wachter and A. Reuter. "The ConTract model." In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pp. 239–264. Morgan Kaufmann, 1992.
40. William E. Weihl. "Commutativity-based concurrency control for abstract data types." *IEEE Transactions on Computers* 37(12), pp. 1488–1505, 1988.
41. D. L. Wells, J. A. Blakeley, and G. W. Thompsosn. "Architecture of an open object-oriented database management system." *IEEE Computer* 25(10), pp. 74–82, October 1992.
42. J. Widom. "Research problems in data warehousing," in *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM '95)*, November 1995, pp. 25–30.
43. G. Wiederhold. "Views, objects, and databases." *IEEE Computer*, pp. 37–44, December 1986.
44. G. Wiederhold. "Mediators in the architecture of future information systems." *IEEE Computer* 25(3), pp. 38–49, March 1992.
45. G. Wiederhold, P. Wegner, and S. Ceri. "Towards megaprogramming." *Communications of the ACM* 35(11), pp. 89–99, November 1992.