A shorter version will appear in the 28th Hawaii International Conference on System Sciences, Jan 95

Extending an Object-Oriented Programming Language to Support the Integration of Heterogeneous Database Systems

Evaggelia Pitoura*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
Fax : (317) 494-0739
Phone: (317) 494-7871
Email: pitoura@cs.purdue.edu

Abstract

Many advanced applications need to access and integrate resources and services provided by distributed, autonomous, and heterogeneous systems. Building an integrated system to provide uniform access and cooperation among these underlying heterogeneous systems is both an increasing necessity and a formidable challenge. An important component of such a complex integrated system is a language that would serve 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 present 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 integrated system provides a set of basic classes and a set of 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 (i.e., the new class, subclass, and superclass constructors) to support the definition of virtual classes. A virtual class is a template that describes already-created instances. Finally, since the integrated system includes database systems, there is a need for atomicity and concurrency control. The language provides constructs to allow users to control the structure and execution of their programs. Furthermore, atomicity and concurrency information is included in the form of consistency assertions as part of the interface of each basic method.

*supported in part by a Purdue Research Foundation Grant

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 [MHG+92, Sol92]. 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 [SL90]. 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 language *multilanguage*. In this paper, we present the necessary extensions 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. 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 remainder of this paper is structured as follows. In the next section, we describe the object-oriented approach to integration and introduce the basic functionality of the language. In Section 3, we show how the class hierarchy of the system can be used to support sets and associative queries, which are necessary for efficient database management. In Section 4, we define the view mechanism of the multilanguage, which is created through extensions of the standard class constructors. The interaction of the multilanguange with the transaction manager is presented in Section 5. In this section, we introduce consistency annotation as a means of including atomicity and concurrency information within class definitions. Special constructs for controlling the execution flow are also presented here. Finally, we discuss related work in Section 6 and summarize our conclusions in Section 7.

2 The Multilanguage

The proposed methodology is grounded upon an object-based 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 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

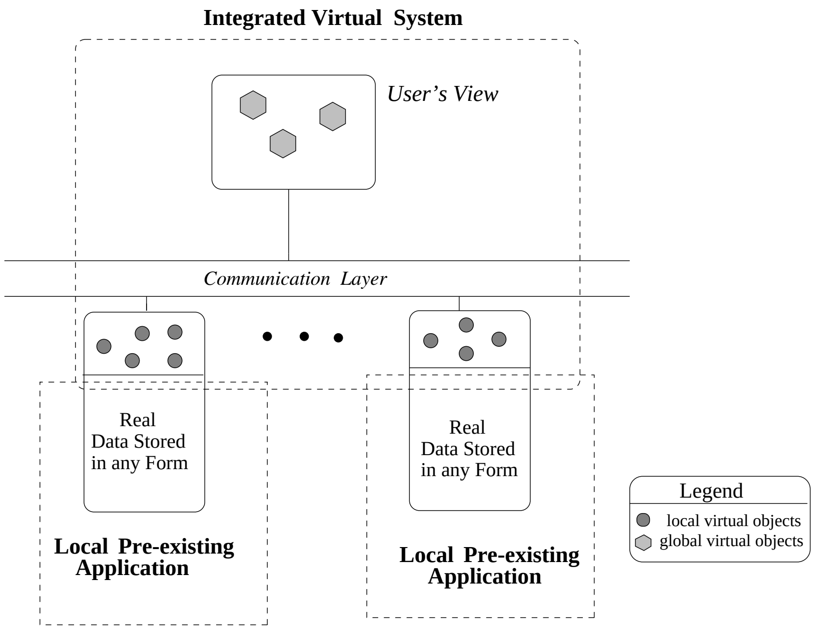

Figure 1: Distributed Object Architecture

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 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 [LM91]. 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.

2.1 The Basic Functionality Provided by the Language

Our rationale for the provided functionality of the multilanguage is as follows:

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

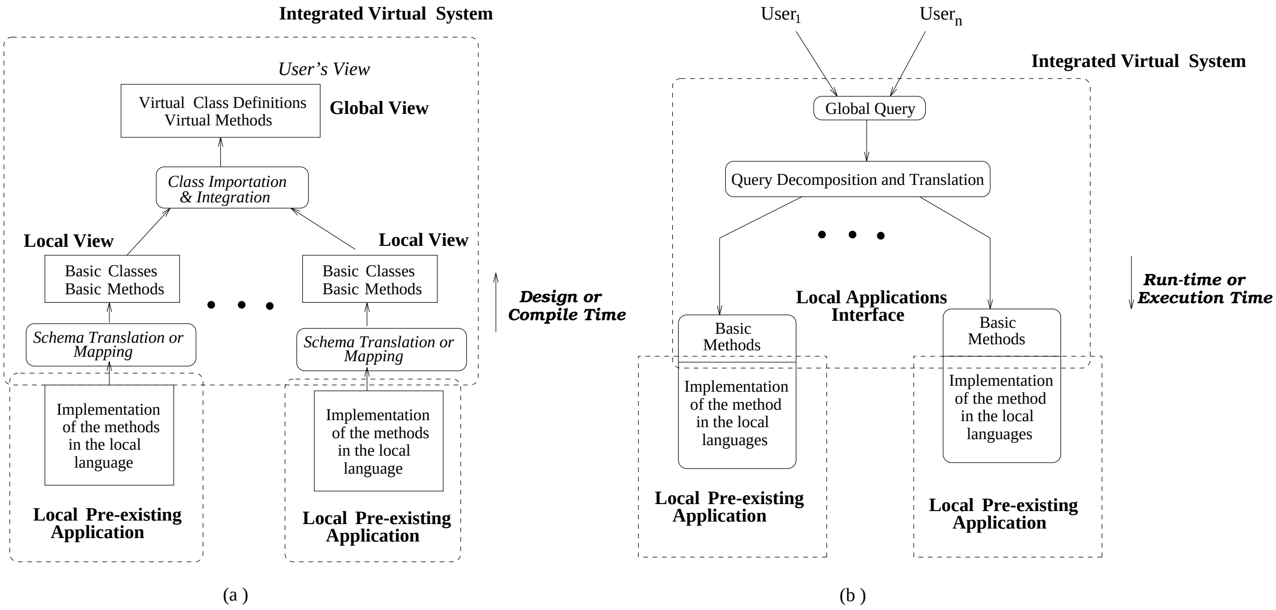

Figure 2: The use of the multilanguage (a) at design or compilation time, to define views and (b) at execution time, to express queries

- The language should be programming language-based rather than query- or database-based, to allow the integration of nondatabase systems. However, since the integrated system includes database systems, efficient data access is very important, and non-procedural 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 will be explained in detail in Section 3. 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.

- 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. View-definition facilities have been used in database languages to provide a different view of stored data to database users. We shall adopt this concept from database systems and extend the class constructors to define what we call *virtual* classes. Data defined during the execution of a multilanguage program are also persistent unless otherwise indicated. The view mechanism is detailed in Section 4.

- 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 Section 5.

- Since, our design is based on the principle of minimality, we seek to define as few new constructs as possible.

The following sections discuss the incorporation of these characteristics in a high-level object-oriented programming language.

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* [US87], 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.

2.2 An 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.

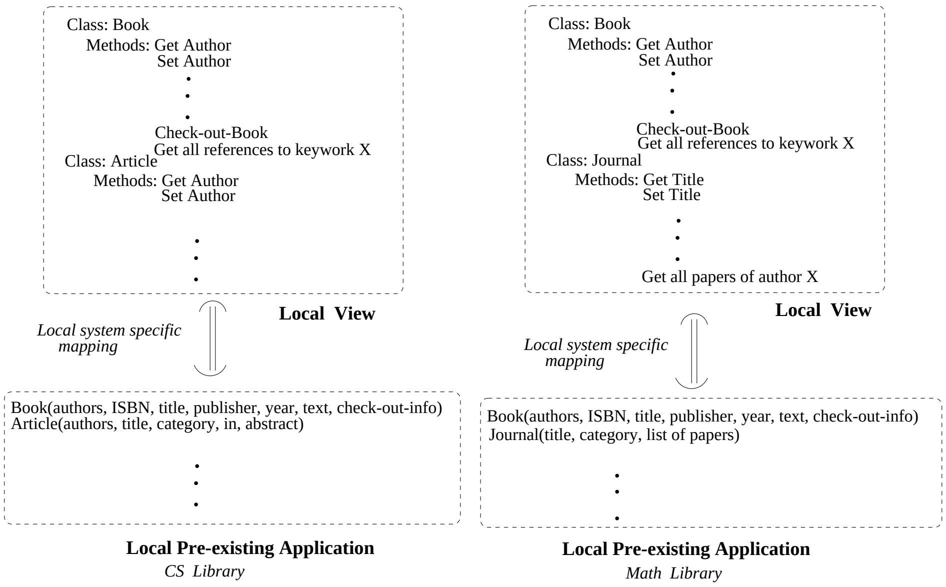

Figure 3: An example. Each local application provides a mapping of its data to a local view that consists of a set of classes. The methods of these classes (called basic methods) offer the only means of accessing the data of the pre-existing applications.

3 Handling Database Resources

Using as a multilanguage a language that is an extension of a programming language allows us to manipulate both database and non-database resources uniformly thus alleviating problems of impedance mismatch. On the other hand, the language should be customized to accommodate the manipulation of large amounts of data and to support associative queries on sets of such data.

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 set of objects. For that purpose, we define the notion of a *class extent*.

Definition 1 (extent) *The extent of a class includes all objects that belong to the class.*

In other words, the extent of a class defines how a class is populated.

We have informally defined the extent of a class as 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:

Definition 2 *An object* "belongs-to a class" *if it is an instance of that class or of any of its subclasses.*

The *belong-to relation* is similar to the *members-of* relation [Ber91, PM90]. Under Definition 2, 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 2 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. This is explained in the following section.

3.2 Class Hierarchy

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

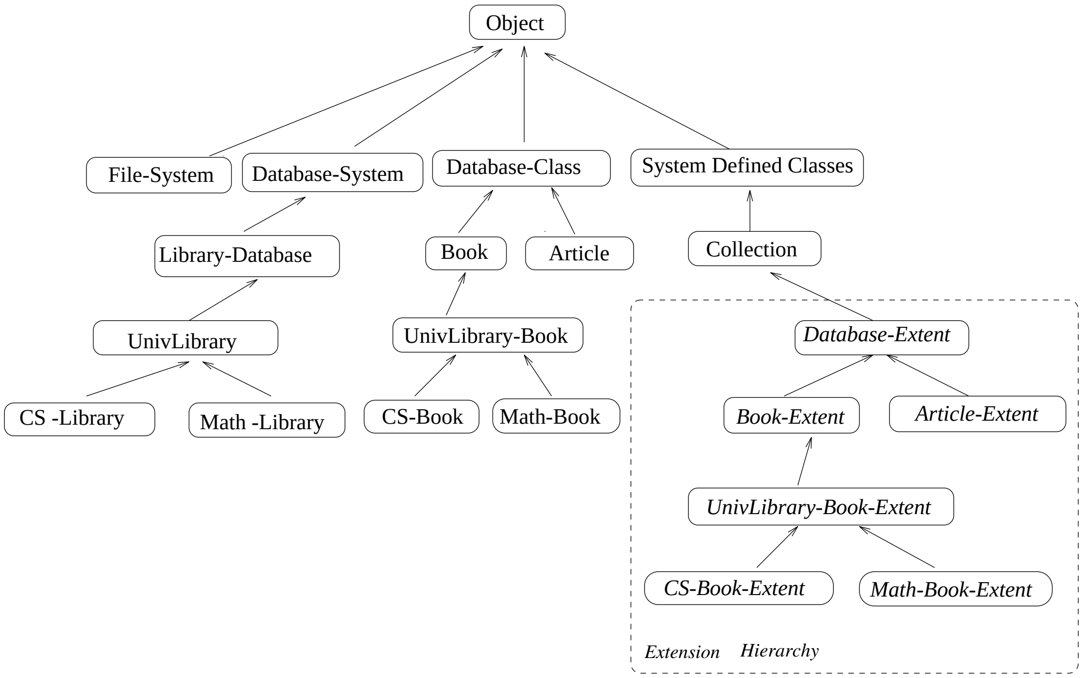

Figure 4: Part of the class hierarchy

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.

All 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 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 subsets of this 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.

<Class> *Database-Class*
<Subclass of> *Object*
<Methods>
Get(attribute)
Set(attribute)
⋮
<End>

<Class> *Database-Extent*
<Subclass of> *Collection*
<Methods>
Insert(object)
Select(object with attribute-Y)
⋮
<End>

<Class> *CS-Book*
<Subclass of> *UnivLibrary-Book*
<Methods>
Print-abstract
Get-all-references-to-X
Check-out-book
⋮
<End>

Figure 5: Class definitions of some classes

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.

4 View Definition

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 are not stored, but are recomputed for each query that refers to them. Views play an important role in the creation of an integrated system, since they can be used to define a global view that includes all information stored in the component systems.

If no relations pertained among the basic classes provided by each component system, 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 gives a taxonomy of the possible conflicts when object-oriented models are used.

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 Synonyms: the same concept is described by different names	Article for a journal article and for a newspaper article 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 although the same concept is represented by the same construct, the classes have different methods or the methods have different parameters or return values	Author is a class in one and a method in the other 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

Table 1: Taxonomy of the possible conflicts between two local views

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* [BLN86]. 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 2 lays out the possible interschema relations.

A global view provides the user with a personalized view in which most conflicts are resolved and interschema relationships are expressed.

Definition 3 *An* object-oriented view *is defined as a set of virtual classes that are populated by local or* imaginary *objects. Imaginary objects do not directly correspond to local objects but to parts* *of one or more such local objects. Once a virtual class is created, it should be treated like any other class.*

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

Table 2: Interschema Relations

The set of virtual classes defines a schema and the objects that populate them define a (virtual) database described by the schema [HZ90]. The classes used in the definition of a virtual class are called *base classes*. Extending the concept of a virtual database, a virtual integrated system is defined as a collection of virtual classes populated by virtual (or proxy) objects. A virtual class is a template describing already created-instances. A view defines the scope of the multilanguage program.

While the majority of object-oriented languages provide a subclass constructor and support inheritance from the superclass to the subclass, they 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 [Ped89, SN88]). To resolve this asymmetry, we first define the notion of a superclass constructor before proceeding to define the view mechanism of the multilanguage.

Definition 4 *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.*

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. Our view mechanism has the following goals:

- 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.

- The view mechanism should take into account the fact that each database class has a dual class, namely its extent.

- The view mechanism should facilitate the resolution of conflicts and the expression of inter-schema relations.

4.1 Class Constructors for Defining Virtual Classes

In an object-oriented language, 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 new 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, which in our approach are both represented as methods; and

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

4.1.1 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 view can incorporate 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 view. Part of the imported data can be hidden by using explicit hide commands.

Example 1 *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 can be defined by queries on existing classes. The new classes have no implicit relations with existing classes. Their extension is defined as the result of the query, and their methods are defined explicitly and can use methods defined in the base classes.

```
<class> A <import> B1 <from> database-name1
                          where Predicate1 on B1's methods
                   B2 <from> database-name2
                          where Predicate2 on B2's methods



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

4.1.2 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
```

4.1.3 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.

4.2 Issues

In the following section, we discuss several subtle issues concerning the proposed view mechanism.

4.2.1 Identity 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.

4.2.2 Resolution Problems

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.

4.2.3 Inheritance vs Delegation

In this paper, we propose the use of the concept of class hierarchies as the means for incremental sharing and definition. An alternative mechanism, independent of the concept of class, is *delegation*. Delegation [Ste87, Lie86, Weg87] is a mechanism that allows objects to delegate responsibility for performing an operation to one or more designated ancestors. A key feature is that when an object delegates an operation to one of its ancestors, the operation is performed in the environment (scope) of the ancestor. In lieu of delegation, our proposed method adopts inheritance, since this is the concept supported by most object-oriented programming languages.

4.2.4 Conflicts and Interschema Relations

Table 3 shows how the view mechanism is used to resolve conflicts and to express interschema relations. 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 resoved by defining appropriate reconciliation functions. 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 way.

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 3: Using the view mechanism (a) to resolve conflicts and (b) to express interschema relationships

5 Support for Concurrency

Programs written in the multilanguage access persistent data stored in various pre-existing database systems. We call these programs *tasks*. Tasks are combinations of the basic methods provided by the local systems. Transaction management is performed at two levels: at a local level by the pre-existing transaction managers of the local databases, and at a global level by a global transaction manager. According to our assumption, the global transaction manager interacts with the local transaction managers by submitting basic methods. This differs from the traditional approach where the global transaction manager submits transactions consisting of simple read and write operations, rather than methods.

5.1 Consistency Assertions

In a 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 5 *Methods are defined as Hoare triples. Each method $op_i(a, b)$ on an object o, where o is an instance of a class C, a is an input parameter, and b is an output parameter, is annotated as follows: $\{P_i\}$ $op_i(a, b)$ $\{Q_i\}$, 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).

Consistency annotations offer a natural and convenient formalism for incorporating the consistency requirements of a persistent local object into its interface. It also provides the basis for semantic-based concurrency and atomicity control, as will be shown in section 5.3.

5.2 Defining the Structure of a Task

A user interacts with the multidatabase by submitting programs written in the multilanguage. These programs are combinations of the basic methods. In traditional database management schemas, each user's interaction with the database was implemented as an ACID transaction. The execution of each program was isolated from the execution of all other programs, and was 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 both unrealistic and undesirable for many applications.

Recently, various models have been proposed for modeling the execution of a program as a set of ACID transactions related by different dependencies and having different properties in terms of visibility, atomicity, and durability. The properties of this task have been defined functionally [Elm92], or by using first-order logic [CR91].

In this paper, we outline a method for supporting these transaction models by structural constructs of the multidatabase language. We identify mechanisms for specifying the flow of control at two different levels. At the intra-task level, the user should be able to specify the structure of

```
<Class>  Book
    <Subclass of>  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

each single interaction, execute part of it in parallel, or choose alternative paths. At the inter-task level, a user should be able to control the interaction of his own task with other tasks undergoing concurrent execution.

One important issue that needs to be addressed is whether the properties of the model define the user's notion of database correctness. For example, while adherence to the ACID properties of each program execution is sufficient to maintain the database integrity constraints, the potential of arbitrary interaction between two different tasks weakens the assumptions we can make regarding the maintenance of these constraints. A complete treatment of this issue is beyond the scope of this paper. We can assume that a transaction manager will be responsible for ensuring the correctness of the execution of each task. This can be accomplished by respecting the flow of control that the user explicitly specifies and by enforcing at the same time any additional control necessary to ensure the maintenance of the integrity constraints.

5.2.1 Inter-task Dependencies

Traditional concurrency control does not allow any form of interaction between tasks initiated by different users, since serializability enforces the isolation of each task. This model is inappropriate for environments where there is a need for cooperation between users, such as in software development or CAD applications. In such cases, a multidatabase must provide constructs for controlling the interaction between tasks.

Such control can be provided through either direct or indirect means. While the former is based on a message mechanism, the latter relies on the observation of partial changes in object states which are caused by other tasks. Direct communication should support both asynchronous and synchronous messages.

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

This method provides an explicit control of the visibility of a task. 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 is, in a sense, a form of shared memory communication between tasks. This may be accomplished by explicitly defining breakpoints within the execution of a task where other tasks are allowed to interleave.

It is the responsibility of the transaction manager to monitor the validity of a specific inter-task interaction. For example, if only atomic tasks are allowed, any use of the above mechanisms should be deemed illegal.

5.2.2 Intra-task Dependencies

Dependencies between the execution of methods inside the same task can be specified in terms of the submission and completion (abort or commit) points of each method. In reality, these are the only points within the lifetime of a task that can be specified by a user, since the exact execution time of a method is under the control of local systems.

We allow the user a specific degree of parallelism inside each task by providing an explicit *do-in-parallel* construct. This construct specifies that a number of methods can be submitted concurrently. The user should also be able to select an alternative method, if the original method aborts. Language constructs can also be used to express real-time dependencies based on requirements about the submission and completion time of a method.

These dependencies should be validated by the transaction manager since executions that violate database consistency may arise for example as a result of the parallel execution of conflicting operations.

5.3 Ensuring Database Consistency and Task Atomicity

We assume that the local transaction manager at each site ensures that each basic method is executed as an ACID transaction. Then, the role of the global transaction manager is to maintain the structure and correctness of the multilanguage tasks. The exact definition and maintenance of correctness is not a function of the multilanguage but rather of the transaction manager. However, we want to emphasize the impact of the multilanguage constructs on transaction management.

The consistency assertion can be used to define when two operations conflict. Conflicting methods can be 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 commutability 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. The list of definitions is intended to be illustrative rather than exclusive.

Definition 6 *Two basic methods* $\{P_i\}$ op_i $\{Q_i\}$, *and* $\{P_j\}$ op_j $\{Q_j\}$ *commute if:*

- $P_i \wedge P_j \Rightarrow wp.op_j.P_i$ *(non-interference) [OG76];*
- $P_j \wedge wp.op_i.P_j \Rightarrow P_i \wedge wp.op_j.P_i$ *(left commutativity) [AAS93, HH91];*
- *In any state where both* P_i *and* P_j *hold (both* op_i *and* op_j *can be executed),* $P_i \Rightarrow wp.op_i.P_j$, *and* $P_j \Rightarrow wp.op_j.P_i$ *(forward commutativity) or, if* $P_i \Rightarrow wp.op_i.P_j$, *then* $P_j \Rightarrow wp.op_j.P_i$ *(backward commutativity) [Wei88].*

Another important concern of the global transaction manager 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 method. To resolve this problem, most systems either attempt to compensate (semantically undo) a committed method or to retry an aborted method. Again, the role of the multilanguage is to provide information to the transaction manager. This information can include the type of each method, such as whether it is retriable or compensatable, and, in the latter case the compensation method to be used.

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

```
<Class> Book
    <Subclass of> 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

6 Related Work

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.

6.1 General Frameworks for Building Extensible and Interoperable Systems:

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 [NWM93, Gro91], 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 [WBT92], 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 [WWC92] 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 [Wie92], 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.

6.2 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 [BCG+87]. 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 [DH84, Mot87, AAD+93, Wie86, KDN91, CL92, SLT91, AB91, HZ90, Ber92] 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 [AB91]. In [AB91], 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 [Elm92]). The approach of making transaction specification constructs part of the multilanguage is also demonstrated in IPL [CBE93]. 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 [Mey93] and in the area of database transaction systems in [ELLR90, WR92]. 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.

7 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 [SL90, PBE94]. 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

[AAD+93] Rafi Ahmed, Joseph Albert, Weimin Du, William Kent, Witold Litwin, and Ming-Chien Shan. An Overview of Pegasus. In *Proceedings of the RIDE-IMS*, pages 273–277, April 1993.

[AAS93] Divyakant Agrawal, Amr El Abbadi, and Ambuj K. Singh. Consistency and Orderability: Semantics-based Correctness Criteria for Databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.

[AB91] Serge Abiteboul and Anthony Bonner. Objects and Views. In *Proceedings of the ACM SIGMOD*, pages 238–247, 1991.

[BCG+87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, and Nat Ballou. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(4):3–26, January 1987.

[Ber91] Elisa Bertino. Integration of Heterogeneous Data Repositories by Using Object-Oriented Views. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, April 1991.

[Ber92] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology - EDBT '92*, pages 136–151. Springer Verlag, 1992.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. Comparison of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[CBE93] Jiansan Chen, Omran Bukhres, and Ahmed K. Elmagarmid. IPL: A Multidatabase Transaction Specification Language. In *Proceedings of the 1993 International Conference on Distributed Computing*, 1993.

[CL92] Jan Chomicki and Witold Litwin. Declarative Definition of Object-Oriented Multidatabase Mappings. In *Proceedings of the International Workshop on Distributed Object Management*, pages 307–325, Edmonton, Canada, August 1992.

[CR91] Panos K. Chrysantis and Krithi Ramamritham. A Formalism for Extended Transaction Models. In *Proceedings of the International Conference on Very Large Databases*, pages 103–112, September 1991.

[DH84] U. Dayal and H. Hwang. View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, 10(6):628–645, 1984.

[ELLR90] 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*, pages 507–518, August 1990.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[Gro91] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 91.12.1, December 1991.

[HH91] Thanasis Hadjilacos and Vassos Hadjilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43:2–24, 1991.

[HZ90] Sandra Heiler and Stanley Zdonik. Object Views: Extending the Vision. In *Proceedings of the 6th International Conference on Data Engineering*, pages 86–93, 1990.

[KDN91] M. Kaul, K. Drosten, and E.J. Neuhold. Viewsystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. In *IEEE International Conference on Data Engineering*, pages 2–10, 1991.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of OOPSLA '86*, pages 214–223, September 1986.

[LM91] Qing Li and Dennis McLeod. An Object-Oriented Approach to Federated Databases. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 64–70, April 1991.

[Mey93] Bertrand Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[MHG+92] Frank Manola, Sandra Heiler, Dimitrios Georgakopoulos, Mark Hornick, and Michael Brodie. Distributed Object Management. *International Journal of Intelligent and Cooperative Information Systems*, 1(1), June 1992.

[Mot87] Amihai Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, July 1987.

[NWM93] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, 26(6):57–67, June 1993.

[OG76] Susan Owiski and David Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6:319–340, 1976.

[PBE94] Evaggelia Pitoura, Omran Bukhres, and Ahmed Elmagarmid. A Survey of Object-Oriented Multidatabase Systems. In Ahmed Elmagarmid and Omran Bukhres, editors, *Object-Oriented Multidatabases*. Prentice Hall, 1994. to appear.

[Ped89] C. H. Pedersen. Extending Ordinary Inheritance Schemes to Include Generalization. In *Proceedings of OOPSLA '89*, pages 407–417, October 1989.

[PM90] M. P. Papazoglou and L. Marinos. An Object-Oriented Approach to Distributed Data Management. *Journal of Systems and Software*, 11(2):95–109, February 1990.

[SL90] Amit Sheth and James Larson. Federated Database Systems. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SLT91] Marc H. Scholl, Cristian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases*, pages 188–207, Munich, Germany, December 1991.

[SN88] Michael Schrefl and Erich J. Neuhold. Object Class Definition by Generalization using Upward Inheritance. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 4–13, 1988.

[Sol92] Richard Mark Soley. Using Object Technology to Integrate Distributed Applications. In Jr Charles J. Petrie, editor, *Enterprice Integration Modeling, Proceedings of the First International Conference*, pages 446–454. The MIT Press, 1992.

[Ste87] Lynn Andrea Stein. Delegation Is Inheritance. In *Proceedings of OOPSLA '87*, pages 138–146, October 1987.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*, pages 227–242, October 1987.

[WBT92] David L. Wells, Jose A. Blakeley, and Graig W. Thompsosn. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–82, October 1992.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pages 168–182, October 1987.

[Wei88] William E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[Wie86] Gio Wiederhold. Views, Objects, and Databases. *IEEE Computer*, pages 37–44, December 1986.

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[WR92] Helmut Wachter and Adreas Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 239–264. Morgan Kaufmann, 1992.

[WWC92] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Towards Megaprogramming. *Communications of the ACM*, 35(11):89–99, November 1992.