Object Orientation in Multidatabase Systems

Evaggelia Pitoura Omran Bukhres Ahmed Elmagarmid

Department of Computer Science
Purdue University
West Lafayette, IN 47907-1398
{pitoura, bukhres, ake}@cs.purdue.edu

Report CSD-TR-93-084 (Revised: April 1995)
To appear in *ACM Computing Surveys*, June 1995

Abstract

A multidatabase system (MDBS) is a confederation of pre-existing distributed, heterogeneous, and autonomous database systems. There has been a recent proliferation of research suggesting the application of object-oriented techniques to facilitate the complex task of designing and implementing MDBSs. Although this approach seems promising, the lack of a general framework impedes any further development. The goal of this paper is to provide a concrete analysis and categorization of the various ways in which object orientation has affected the task of designing and implementing MDBSs.

We identify three dimensions in which the object-oriented paradigm has influenced this task: the general system architecture, the schema architecture, and the heterogeneous transaction management. Then we provide a classification and a comprehensive analysis of the issues related to each of the above dimensions. To demonstrate the applicability of this analysis, we conclude with a comparative review of existing multidatabase systems.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases–*data translation, program translation*; D.1.5 [**Programming Techniques**]: Object-oriented programming; H.2.1 [**Database Management**]: Logic Design–*data models, schema and subschema*; H.2.3 [**Database Management**]: Languages–*query languages*; H.2.4 [**Database Management**]: Systems–*distributed systems, query processing, transaction processing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems– *distributed applications, distributed databases.*

General Terms: design, languages, management, standardization

Additional Keywords and Phrases: multidatabases, federated databases, distributed objects, integration, views.

Contents

1 Introduction

The computing environemnt in most contemporary organizations contains *distributed*, *heterogeneous*, and *autonomous* hardware and software systems. There is an increasing need for a technology to support cooperation of the services this software and hardware provides. The requirement for building systems that combine heterogeneous resources and services can be met at the low level of *interconnectivity* or at the higher level of *interoperability* [MHG+92, Sol92]. Interconnectivity simply supports system communication, while interoperability additionally allows systems to cooperate in the joint execution of tasks.

In this paper, we focus on the special case in which the goal is to use and combine information and services provided by database systems. A *multidatabase system* (MDBS) [EP990] is a confederation of pre-existing, autonomous and possibly heterogeneous database systems. The pre-existing database systems that participate in the confederation are called *local* or *component* database systems. The high-level architecture of a multidatabase system is depicted in Figure 1. The "global" layer provides access to the underlying local systems. The complexity of this layer varies from allowing direct access to each local system to providing sophisticated seamless access with control flow facilities.

Creating a MDBS is complicated by the heterogeneity and autonomy of its component systems. Heterogeneity manifests itself through differences at the operating, database, hardware, or communication level of the component systems [SL90]. Here, we concentrate on the types of heterogeneities caused by differences at the database system level, including discrepancies among data models and query languages and variability in system level support for concurrency, commitment and recovery. Building multidatabase systems is further complicated by the fact that the component databases are autonomous, have been independently designed, and operate under local control.

Recently many researchers have suggested using object-oriented techniques to facilitate building multidatabase systems. Object-oriented techniques, which originated in the area of programming languages, have been widely applied in all areas of computer science including software design, database technology, artificial intelligence and distributed systems. Although using them in building multidatabases seems promising, the lack of a common methodology impedes any further development.

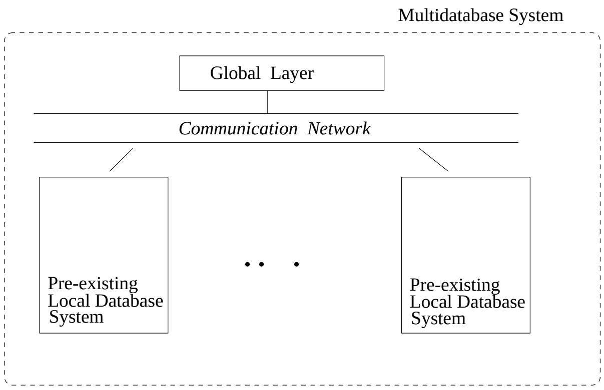

Figure 1: The high-level architectue of a multidatabase system.

This survey analyzes the various ways in which object-oriented techniques have influenced the design and operation of multidatabases. Our goal is to classify the approaches proposed and provide a comprehensive analysis of the issues involved. Although this survey is self-contained, a familiarity with basic database concepts (e.g., database textbooks such as [OV91]) and with the basic principles of object orientation (e.g., the survey paper [Weg87]) will facilitate understanding the issues involved.

1.1 Research Directions in Object-Oriented Multidatabase Systems

In this section we classify the ways in which object technology has influenced the design and implementation of multidatabase systems. First, the application of object-oriented concepts in system architectures provided a natural model for heterogeneous, autonomous, and distributed systems. According to this architectural model, called *Distributed Object Architecture*, the resources of the various systems are modeled as objects, while the services provided are modeled as the methods of these objects. Methods constitute the interface of the objects. In the special case in which the systems are database systems, the resources are the information stored in the database and the facilities provided are efficient methods of retrieving and updating this information.

Second, object technology has been used in multidatabase systems at a finer level of granularity. The information stored in a database is structured according to a data model. When a component database participates in a multidatabase system, its data model is mapped to a data model that is the same for all participating systems, the *Common (or Canonical) Data Model (CDM)*. Several researchers have recently advocated the use of an object-oriented data model as the CDM. The objects of the database model are of a finer granularity than the distributed objects: at one extreme, an entire component database may be modeled as a single distributed complex object [LM91].

In a multidatabase system, multiple users simultaneously access various component systems. Heterogeneous transaction management deals with maintaining the consistency of each component system individually and of the multidatabase system as a whole. Object technology has also influenced a number of aspects of heterogeneous transaction management. It offers an efficient method of modeling and implementation, facilitates the use of semantic information, and has independently introduced the notion of local transaction management.

Summarizing, we can identify the following three dimensions in which the object-oriented paradigm has influenced the design and implementation of multidatabase systems:

- system architectures have been influenced by the introduction of **distributed object-based architectures**
- schema architectures have been influenced by the use of an **object-oriented common data model** and
- transaction management has been influenced by the application of techniques from **object-oriented transaction management**

The above dimensions are orthogonal in the sense that systems may support object-orientation on one dimension but not necessarily on others. For example, a database system having a relational common data model can participate in a distributed object architecture by being considered a (large) distributed object. Analogously, database systems with object-oriented common data models can participate in non-object-based system architectures. Moreover, systems that do not support objects can use object-oriented techniques in developing their transaction management schemes. In the following sections, the relationships among the above dimensions will be further clarified.

Although all the above combinations are viable, a fully object-oriented multidatabase should support the same object model at all dimensions to avoid confusions, incompatibilities, or errors and repetitions in implementation. However, different requirements are placed on each one of these dimensions, resulting in data models that emphasize different features. Thus, different object-oriented data models have been introduced for the architecture, schema, and transaction management level. At the level of system architectures, models tend to be programming-based and focus on such issues as efficient ways of implementing remote procedure calls and naming schemas. At the level of schema architectures, models tend to be database-oriented, support persistency and database functionality and have extended view-definition facilities. Finally, at the transaction management level, models usually support active objects appropriate for modeling transactions and their interaction. In addition, at the transaction management level, different approaches utilize different features of the object model in the pursuit of efficiency. In this paper, we first provide a reference programming-based model and then highlight variations of this model appropriate for each of the above dimensions.

1.2 A Reference Programming-Based Object Model

Object orientation [Weg87] is an abstraction mechanism in which the world is modeled as a collection of independent objects that communicate with each other by exchanging messages. An *object* is characterized by its state and behavior and has a unique identifier assigned to it upon its creation. The state of an object is defined as the set of values of *instance variables*. The value of an instance variable is also an object. The behavior of an object is modeled by the set of operations or *methods* that are applicable to it. Methods are invoked by sending *messages* to the appropriate object. The state of an object can be accessed only through messages; thus, the implementation of an object is hidden from other objects.

Each object is an instance of a *class*. A class is a template (cookie-cutter) from which objects may be created. All objects of a class have the same kind of instance variables, share common operations, and therefore demonstrate uniform behavior. Classes are also objects. The instance variables of a class are called class variables and the methods of a class are called class methods. Class variables represent properties common to all instances of the class. A typical class method is *new*, which creates an instance of the class.

Classes are organized in a class hierarchy. When a class B is defined as a *subclass* of a class A, class B *inherits* all the methods and variables of A. A is called a *superclass* of B. Class B may include additional methods and variables. Furthermore, class B may redefine (overwrite) any method inherited from A to suit its own needs. Inheritance from a single superclass is called *single inheritance*; inheritance from multiple superclasses is called *multiple inheritance*. Some systems also consider classes to be instances of classes called *metaclasses*. Metaclasses define the structure and behavior of their instance classes. The metaclass concept is a very powerful one, since it provides systems with the ability to redefine or refine their class mechanism.

The relations typically supported by the object-oriented model are: the *classification* or *instance-of* relation between an object and the class (typically one) of which it is an instance, the *generalization/specialization* or *is-a* relation between a class and its superclasses, and the *aggregation* relation between an object and its instance variables. We discuss briefly below some design alternatives of the basic model.

• **Delegation versus Inheritance.** Inheritance is a mechanism for incremental sharing and definition in class hierarchies. 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.

• **Method Resolution.** Since a class may provide a different implementation for an inherited method, methods are overloaded in object-oriented systems. The selection of the appropriate method is called *method resolution*. With single inheritance (where the class hierarchy is a tree), when a message is sent to an object of a class A the most specific method is used; that is the method defined in the nearest ancestor class of A. This resolution method is also applied in the case of multiple inheritance, although the problem there is complicated by the fact that the same method may be defined in more than one of A's superclasses. In such an instance, there is no default resolution method for specifying which of the multiply defined methods A should inherit. Some systems support *multimethods*, which are methods that involve as arguments more than one object and in which the classes of all the arguments are considered in selecting the appropriate method during resolution [Day89, HZ90].

• **Subtyping versus Subclassing.** *Subtyping* rules are rules that determine which objects are acceptable in a specific context. Every object of a subtype can be used in any context where an object of any of its supertypes could be used. Although some systems relate subtyping and subclassing, subtyping should be based on the behavior of objects in order to increase flexibility [Sny86]. If instances of type A meet the external specification of class B, A should then be a subtype of B, irrespective of whether A is a subclass of B. *Conformance* [BGM89, RTL$^+$91] is a mechanism for implicitly deriving subtyping relations based on behavioral specifications.

1.3 Organization of this Paper

The remainder of this paper is organized as follows. In Section 2, we discuss the first dimension, namely distributed object-based architectures. Since the focus of this paper is on the integration of database systems, the influence of the architecture on multidatabase systems is stressed. Thus, this section does not exhaust this very important research area (for a review on this topic, see for example [NWM93]). The following two sections discuss the other two directions in detail. In Section 3, we describe how the object-oriented model has been adapted to serve as the data model of the multidatabase and its role in facilitating the tasks of schema translation and integration. In Section 4, we discuss the impact of object-oriented transaction management in multidatabases. Finally, Section 5 is a comparative review of existing multidatabase systems that adopt object-oriented techniques in one or more of the above directions.

2 Object-Based Architectures for Distributed Heterogeneous Systems

A popular way [MHG+92, NWM93] of modeling a distributed heterogeneous system is as a distributed collection of interacting objects that represent the distributed system resources. Each component system defines an interface of services and provides an implementation for these services. A client interacts with the heterogeneous system by issuing requests expressed in a common language. Distributed object managers are responsible for translating the client's requests in terms of the available services, for directing these requests to the appropriate systems, and for providing the response expressed in the same common language.

The use of objects to model distributed components accommodates both the heterogeneity and the autonomy requirements. The modeling of distributed resources as objects supports heterogeneity because the messages sent to a distributed component depend only on its interface and not on the internal implementation of the method or the component. This approach also respects autonomy because the components may operate independently and transparently, provided that their interfaces remain unchanged. The ultimate goal of distributed object-based architectures is the construction of a heterogeneous system in which all system resources may be treated as a commonly accessible collection of objects that can be recombined in arbitrary ways to provide new information accessing capabilities.

2.1 MDBSs in Object-Based Architectures

Object-based architectures offer means of integrating applications across technology domains, including the domains of GUIs, file systems, database systems, and programming languages. MDBSs focus on issues within the database system domain. As a consequence, an object model for a heterogeneous system that includes components that are database systems should be a specialization of the object model of the general object-based system architecture that will support database functionality such as persistence, querying, transactions, and concurrent sharing.

In the special case where an object-oriented component database system participates in a heterogeneous system with an object-based architecture, there are two types of objects, the local objects supported by the component database and the distributed objects of the heterogeneous system. The component object-oriented database system supports millions of fine-grained objects. Providing clients of the heterogeneous system with direct access to these objects may involve significant overhead or may violate the autonomy or security of the database. Instead, the heterogeneous system may provide access to a containing object, which in the extreme case may be the whole database. Then, the containing object can handle the requests. In this case, the local, fine-grained objects are hidden from the client of the heterogeneous system. In other words, the distributed objects of the heterogeneous system may not correspond directly to the local objects of the component database system.

Finally, research in architectures for distributed systems has concentrated on interconnectivity issues and has not yet addressed interoperability aspects. Thus, most research on integrating information resources is expressed in terms of database integration of the schemas of the component databases.

2.2 Standardization Efforts in Object-Based Architectures

The impact of object-orientation in the architecture of heterogeneous distributed systems is also evident in the fact that most standardization efforts in this direction are based on the object model. In the following, we describe the prevailing such approach, namely the OMG, and report briefly on some others. In the long run, future compliance of a database system with the standards will ease the task of building multidatabase systems. In the short run, new multidatabase systems should take into consideration such standards in defining their interfaces. Standardization efforts towards defining a standard object model are also being made in the database community. We discuss two of these efforts, SQL3 and ODMG, in Section 3.1.1 since they are pertinent to database modeling. These efforts define the standard services that should be provided by each component database system and in this regard are extensions of the object models defined for distributed object-based systems.

Object Management Group. The Object Management Group (OMG) [Sol92] is developing a suite of standards addressing the integration of distributed applications through object technology. The architecture proposed by OMG, the Object Management Architecture (OMA), is depicted in Figure 2. In the OMA model, every piece of software is represented as an object. Objects communicate with other objects via the *Object Request Broker* (ORB), which is the key communication element. ORB provides the mechanisms by which objects transparently make requests and receive responses. Figure 2(a) shows a request being sent by a client to an object implementation: the *client* is the entity that wishes to perform an operation on an object and the *object implementation* is the code and data that actually implement the object.

The OMG categorizes objects into three broad categories: Application Objects, Object Services and Common Facilities (Figure 2(b)). Application Facilities is a placeholder for objects that belong to the specific applications that are being integrated. The Common Facilities comprise general facilities useful in many applications that will be made available through OMA-compliant interfaces. The Object Services provide the main functions for implementing basic object functionality using the ORB, e.g., the logical modeling and the physical storage of objects. The proposed standard for the ORB, *CORBA* (Common Object Request Broker Architecture) [Gro91] supports a general Interface Definition Language (IDL) that may be mapped to any implementation language. ORB provides for location transparency and permits the integration of applications via *wrappers* to implement CORBA-compliant behavior.

All objects are expressed in a common Object Model [Gro92]. In this model, subtyping is based not on subclassing (Section 1.2) but rather on the behavior of objects. Inheritance is defined between interfaces. An interface is a description of a set of possible operations that a client may request of an object. An interface type is the type that is satisfied by any object that complies with a particular interface. An interface can be derived from another interface, which is then called a base interface of the derived interface. A derived interface inherits all elements (variables and methods) of the base interface and may redefine them or define new elements.

Other Efforts. In addition to the OMG standardization efforts, ISO and CCITT are also working on a joint standardization effort known as Open Distributed Processing (ODP) [Tay92]. ODP's goal is the development of a reference model to integrate a wide range of future ODP standards for distributed systems. The support of object-orientation in commercial systems and in stan-

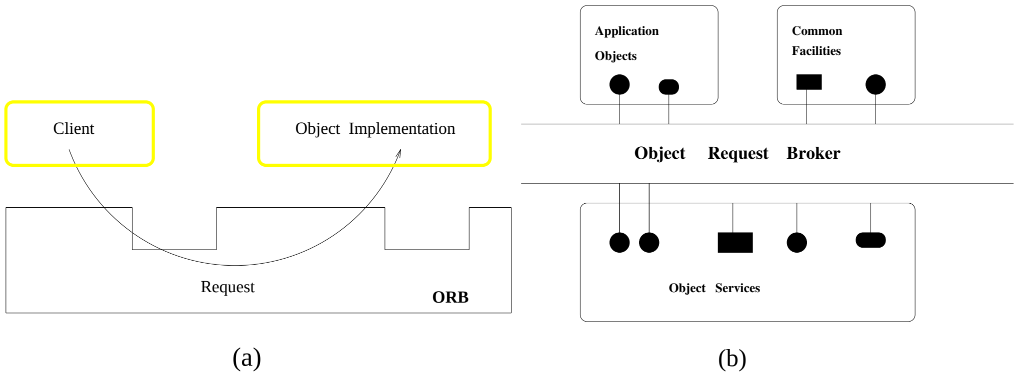

Figure 2: (a) A request being sent through the Object Request Broker, (b) Object Manager Architecture

dardization efforts for heterogeneous processing is examined in [NWM93]. Support in commercial systems for heterogeneous processing includes OSF's DCE (a set of tools and services to support distributed applications) and BBN's Cronus system (a system that provides operating system and communication services). Finally, X3H7, a new ANSI/X3 Technical Committee, has the mission of harmonizing the object-oriented aspects of standards developed by other committees [Ken93].

3 MDBSs with an Object-Oriented Common Data Model

The traditional three-level architecture [TK78] used to describe the schema architecture of a centralized database system has been expanded [DEL+82] to describe the architecture of a MDBS. In this 5-level architecture (see Figure 3, adapted from [SL90]), the conceptual schema of each component database is called the *local schema*. The local schema is expressed in the native data model of the component database; thus, the local schemas of different component databases may be expressed in different data models. To facilitate access to the system, most approaches translate the local schemas into a common data model, called the **canonical** or **common data model** (CDM). The schema derived by this translation is called the *component schema*. Each database participates in the federation by exporting a part of its component schema, called the *export schema*. A *federated* or *global* schema is created by the integration of multiple export schemas. Finally, for customization or access control reasons, an *external* schema is created to meet the needs of a specific group of users or applications.

Different types of multidatabase systems are created by different levels of integration of the export schemas of the component databases [BHP92, SL90]. The **non-federated approach** [LMR90] assumes no integration of the export schemas. An important component of a non-federated multidatabase system is the multidatabase language, that allows uniform access to all component databases. In contradistinction to the multidatabase approach, the **federated approach** [SL90] assumes the integration of the export schemas to create a global schema. Federated database

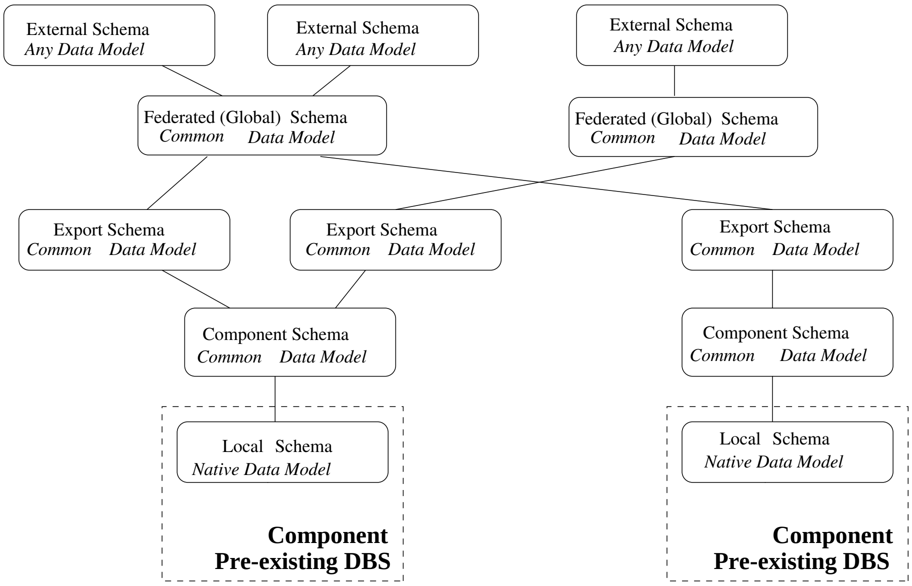

Figure 3: The 5-level schema architecture.

systems (FDBSs) can be further categorized based on the distribution of integration. *Centralized* FDBS [Ber91] support a single federated schema systemwise. This federated schema is built by the selective integration of the export schemas of the component sites. In the case of *decentralized* FDBSs [CT91, LM91] each component site builds its own federated schema by integrating its local schema with the export schemas of some other component sites. Decentralized FDBSs are further characterized by the degree of consistency that they maintain among the different federated schemas.

The translation of local schemas into the CDM is essential to both the federated and non-federated approaches. The CDM should be both rich enough [SCGS91, BLN86] to capture the semantics expressed or implicit in the local schemas and simple enough to facilitate the creation of the federated schema in the federated approach and the multidatabase queries in the non-federated approach. Most systems [BHP92, TTC+90, SBCE93] use a relational data model as the CDM. Recently, many systems [BEM92] have been introduced that use an object-oriented data model as their CDM. In this section, we attempt to evaluate the rationale behind this approach and discuss its effectiveness.

Introduction to Integration

Schema translation alleviates the problems that occur due to the use of different data models. If there were no relations among the concepts represented in each component schema, then the federated schema would simply be the union of the component schemas. Unfortunately, the same concepts may be represented in different databases and furthermore, due to heterogeneity, these concepts may be represented differently.

Type of Conflicts. The following is a general classification of the possible conflicts between component schemas. This classification is independent of the type of the CDM used.

1. **Identity Conflicts** occur when the same concept is represented by different objects in different component databases.

2. **Schema Conflicts** occur when the component schemas that represent the same concept are not identical.

   (a) **Naming conflicts** occur when the same name is used for different concepts (homonyms) or when the same concept is described by different names (synonyms).

   (b) **Structural Conflicts** occur when
   (i) the same concept is represented by different constructs of the data model, or
   (ii) although the same concept is modeled by the same constructs, the constructs used have either different structure (missing or different relations/dependencies) or different behavior (different or missing operations).

3. **Semantic Conflicts** occur when the same concept is interpreted differently in different component databases. This category includes scale or rate differences.

4. **Data Conflicts** occur when the data values of the same concept are different at different component databases.

Similar taxonomies are presented in [BLN86, DH84, KS91, KCGS93]. In [BLN86], two types of conflicts are described: naming and structural conflicts. These largely correspond to the naming and structural conflicts as defined above except from the key structural conflict which, in our taxonomy, is considered a special case of the identity problem. In [DH84], a taxonomy is proposed of conflicts that might occur when all component schemas are expressed in an extended functional model with three basic constructs, functions, objects and types. This taxonomy differentiates between schema and data conflicts. Our definition of schema conflicts is an extension of this formulation with the exception of scale differences, that we classify as semantic rather than as schematic differences. The classification presented in [KS91] is similar to [DH84], but is tailored for the case of relational schemas. [KCGS93] provides a comprehensive taxonomy of conflicts which arise when the common data model is a relational-object model, called SQL/M.

Table 1(a) summarizes the different types of conflicts along with some examples in the case of an object-oriented CDM.

Interschema Relations. In order to perform integration, it is crucial to identify not only the set of common concepts but also the set of different concepts in different schemas 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 federated schema. Interschema relations which arise when the common data model is a relational model have been extensively studied (e.g., [LNE89]) and are most commonly expressed in terms of the inclusion relationships among the domains of the related entities. To the best of our

Type		Definition	Example
Identity conflict		Same concept represented by different objects in different local databases	Copies of same book stored in both CSLibary and MathLibrary with different local identifiers
Schema conflict	Naming	*Homonyms:* Same name used for different concepts	In MathLibrary "media" refers to magazines and newspapers; in CSLibrary to videotapes
		Synonyms: Same concept described by different names	In MathLibrary "references" is used; in CSLibrary "bibliography" is used.
	Structural	Same concept represented by different constructs of the model, (i.e., by a method in one database and a class in the other) or	In CSLibrary number of of citations is an instance variable of the book; in MathLibrary it is a method recomputed upon invocation
		Same concept represented by same construct, but classes have different methods, or methods have different parameters or return values	Book has an instance variable "keywords" in one library but not in the other
Semantic conflict		Same concept interpreted differently in different databases	Conference is a refereed conference in one not in the other
Data conflict		Data values of the same entity are different in different component databases	Same book appears to have different authors

(a)

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

(b)

Table 1: (a) Taxonomy of the possible conflicts. (b) Interschema relations. We consider two local database schemas, one that describes the library of the *Computer Science* Department (CSLibrary) and the other the library of the Department of *Mathematics* (MathLibrary).

knowledge, there has been no comprehensive study of the different interschema relations that can exist when an object-oriented common data model is used. The problem is complicated by the fact that object-oriented models express semantic relations that are difficult to capture by such simple relations as the set-inclusion relation.

Table 1(b) lists some types of interschema relations that correspond directly to the relations supported by the reference object-oriented model along with some examples.

Organization of the remainder of this section

The task of translation and integration are strongly influenced by the data model used to represent the component schemas. In the remainder of this section, we discuss how object-oriented data models facilitate both tasks. The basic object-oriented model as introduced in section 1.2 lacks some concepts necessary to a common data model. In Section 3.1, we discuss how it can be augmented to serve as a common data model. In Section 3.2, under the general title multidatabase languages, we present some issues related to the languages used. Issues related to schema translation where the target of the translation is an object-oriented model are discussed in Section 3.3. In Section 3.4, we focus on issues germane to the creation of the global (federated) schema. Section 3.5 concludes this section with an overview of some of the advantages of using an object-oriented common data model.

3.1 Object-Oriented Data Models used as CDMs

The object-oriented model as defined in Section 1.2 lacks some concepts pertinent to multidatabase systems. Various research approaches have resulted in different extensions of the basic data model. We first describe efforts in ODMG and ANSI SQL3 in terms of defining a standard object-oriented data model. Then, we describe extensions of the model that facilitate integration and translation. Since there is no standard object-oriented data model, in this section we discuss the most prevailing of the proposed extensions.

3.1.1 Standardization Efforts in Object-Oriented Database Models

SQL3 is a new database language standard developed by both ANSI X3H2 and ISO DBL committees targeted for completion in 1997 [Kul94, Kul93]. SQL3 is upwards compatible with SQL-92, the current ANSI/ISO database language standard [Gal92]. The major extension is the addition of an extensible object-oriented type system based on Abstract Data Types (ADTs). However, SQL3 still maintains the restriction that tables are the only persistent structures [Kul93]. ADT definitions include a set of attributes and routines. Using the terminology of the reference model, ADTs correspond to classes, attributes to instance variables and routines to methods. Routines can either be implemented using SQL3 procedural extensions or using code written in external languages. ADTs are related by subtype relationships, where an ADT can be a subtype of multiple supertypes. Resolution of overloaded routines is based on all arguments in a routine invocation.

The Object Database Management Group (ODMG) is a consortium of object-oriented vendors that have developed a standard interface for their products called ODMG-93 [Cat93]. The ODMG members are also members of OMG task forces, and OMG has adopted the ODMG-93 interface as part of the Persistence Service, which is one of OMG's Object Services. Unlike SQL3, ODMG choose not to extend SQL but rather to extend existing programming languages to support persistent data and database functionality. ODMG combines SQL syntax with the OMG object model extensions to allow declarative queries.

3.1.2 Extensions for Object-Oriented Common Data Models

In this section, we discuss a number of proposed extensions of the reference object model for providing database interoperability.

Types and Classes. A class, as defined in the reference model, is a template for creating objects with a specific behavior and structure. A class is not directly related to the real objects whose structure and behavior it models. In a database system we need a language construct to model a set of objects. In this section we discuss how this construct should be defined and related to the notion of a class, so that integration and translation are facilitated.

To express sets of objects and queries on these objects, a new concept, called the *extent* [BCG+87, Ber91] of a class, is defined as the set of all objects that *belong to* the class. The extent of a class defines how a class is populated. To differentiate between the extent of a class and the class itself, many researchers [GCO90] term these aspects the *intensional part* and the *extensional part* of a class, respectively.

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. 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; for instance 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, though the intuitive way to accomplish that, is 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: an object *"belongs-to a class"* if it is an instance of that class or of any of its subclasses. This is also called the *member-of* relation [Ber91, PM90]. Under this definition, the extent of the class UnivLibrary_Book is the union of the extent of all its subclasses and one can express the above request as a query with the extent of UnivLibrary_Book as its target. This is a valid definition 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 a class A is a subclass of a class B then the extent of a 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 hierarchy between sets of objects. We should also mention that, although the definition of a class remains the same, the extent of a class changes with time as new instances are created or deleted.

Many researchers go beyond that and fully differentiate the structure of objects from the real objects having that structure [SST92, SS90, GPN91, GPNS92]. In this case, *types* are defined as templates and *classes* as sets of typed objects. Inheritance of structure and behavior is supported in the subtype hierarchy, whereas the subclass hierarchy, if such exists, is based on set-inclusion relations. A class may have an associated type that defines the structure and behavior of its members. An object may belong to more than one class and to more than one type (or, more precisely, to more than one type extent). Furthermore, a class may contain objects belonging to different types but related by some common property.

It is very difficult to evaluate what is the best choice for a canonical model. Each of the proposed models is accompanied by a related methodology that resolves some types of conflicts and expresses some interschema relations. In general, the distinction between sets and types adds flexibility to the model. Integration may then be supported at two different levels, at a *type (structural)* level and at a *class (set-based)* level. At a structural level, global types abstract commonalities in the structure and behavior of the component types. At a set-based level, objects (or parts of objects) belonging to more than one component class are brought together in some global class. On the other hand, this distinction complicates the maintenance of relations among classes, among types, and between classes and their associated types.

Finally, we should mention that all the above are not necessarily different models, but can be implemented as extensions of the basic object model using the metaclass mechanism. For example, classes representing set of objects, may be considered a special kind of class (e.g., collective classes). For example, ORION [BCG+87] offers an elegant implementation of the concept of class extent.

Schema Evolution Operations. Many systems [BCG+87, LM91, CT91] support *schema evolution* operations, that is operations for dynamically defining and modifying the database schema, e.g., the class definitions and the inheritance structure. These operations play an important role in restructuring the schema resulting from the merging of component schemas.

Semantic Extensions. Many object-oriented models used as CDM are extended to support additional relations which can capture the semantics of the local schemas and of their interrelationships. These extensions can be implemented using the metaclass mechanism of the basic model. The relations added are either specializations of pre-existing relations or correspond to relations explicitly supported by other kinds of data models (e.g., relational). One typical example of the latter case is the *part-of* relation. The basic object-oriented data model is sufficient to represent a collection *(aggregation)* of related objects by allowing an object to have other objects as its instance variables. However, it fails to represent the notion of dependency between objects, since an object does not own the value of its instance variables but simply keeps references to them. Many database models add the notion of dependency by defining a *composite object* [PM90, BCG+87, GCO90] as an object with a hierarchy of exclusive component objects. These component objects are dependent objects, in that their existence depends upon the existence of the composite object that they are part-of.

State and Behavior. Most object-oriented data models used as CDMs do not distinguish between the state and the behavior of an object but use the same construct, usually called *function*, to model both instance variables and methods. An instance variable is modeled by a pair of *set* and *get functions* [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 structural conflicts. 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, take an object of a class named *employee*. Let us say that an *employees*'s *salary* is represented in dollars in one component database, in drachmas in another, and in marks in the global database. Then, if *salary* is represented as a function, we can define an appropriate function in the global schema 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 way to solve the above conflict. Alternatively, schema evolution operators may be applied to the component database schemas prior to their integration, to restructure them appropriately.

Upwards Inheritance. The reference model suffers from an asymmetry. While a subclass constructor is provided and inheritance from a superclass is defined, there is no superclass constructor. Suggested extensions provide such a construct and also define inheritance from a subclass to a superclass, called generalization or upwards inheritance [Ped89, SN88]. Resolution problems related to upwards inheritance are discussed later in this section (Section 3.4.2).

3.2 Multidatabase Languages

There are two fundamental approaches to the design of object-oriented database languages [Kim90]. The first extends a query language (usually SQL) to support the manipulation of object-oriented databases and then embeds the extended query language in the application language. We call this type of languages *query-based.* The second approach extends an object-oriented programming language to support database operations. In this case, the application and query languages are the same and no impedance problem exists [Pit95]. We call this type of languages *programming-based.*

For the purposes of this paper, we further characterize query languages as (1) *language-oriented* when they allow operations (messages) to be sent to single objects or as, (2) *set-oriented* when they permit queries to sets (or collections) of objects other than class extents.

In a multidatabase system, a Data Definition Language (DDL) is used to define the global schema while a Data Manipulation Language (DML) is used to manipulate data. Most object-oriented systems use the same language for both purposes. The language is extended [KLK91] (a) to support queries (or methods) that access data stored in different component databases and (b) to allow the definition of the global schema by integrating the component schemas. The definition of the global schema is usually accomplished by using the view definition facilities of the language. Those facilities are described in Section 3.4.1. When a global schema is not provided, uniform access to the component schemas is accomplished only through the language.

Furthermore, object-oriented languages defined for multidatabases have additional constructs to support the extensions of the data model described in the previous section. These may include declarations for defining types and classes. Some languages [ADD+91] also provide constructs for defining the mapping between local and component schemas.

Finally, some multidatabase systems allow the user to specify the flow of control of his interactions with the database system at a finer level of detail. This specification is expressed using an *extended transaction model* (see Section 4). Some systems extend their DML or DDL with constructs for defining and using extended transaction models [CBE93]. Others offer a special language for defining transaction models [WSHC92].

3.3 Schema Translation

Schema translation is performed when a schema (schema A) represented in one data model is mapped to an equivalent schema (schema B) represented in a different data model. This task generates the mappings that correlate the schema constructs in one schema (schema B) to the schema constructs in another schema (schema A). The task of command transformation [SL90] entails using these mappings to translate commands involving the schema constructs of one schema (schema B) into commands involving the schema constructs of the other schema (schema A). In the multidatabase context, schema translation occurs (see Figure 3):

- when translating from the local model to the common data model, and
- when translating from the federated (global) model to the external model.

When the target schema B is expressed in an object-oriented data model, roughly speaking, relations are mapped to classes and tuples to objects. The inclusion relationship between two relations in schema A may be used to determine the semantic (e.g., subclassing) relationships between the corresponding classes in schema B [CS91].

In addition, during translation, semantic information is collected and represented in the common data model. This process is called *semantic enrichment* [CS91] or *semantic refinement* [MNE88].

Some multidatabase languages (such as HOSL [ADD+91]) provide constructs that support procedural mappings of schemas expressed in other models to their object-oriented model.

[BNPS89] introduces a new approach to schema translation; called the *Operational Mapping Approach.* Instead of defining the correspondence between the data elements of the schemata A and B *(Structural Mapping Approach)*, the correspondence is defined between operations of the different schemata. A number of basic operations of the schema B (called *abstract operations*) are defined in terms of a number of primitive operations of the schema A. All other operations of B are implemented using these abstract operations, possibly automatically by the integration system. The primitive operations provided by A must be an appropriate *minimal set* so that the corresponding abstract operations provide the necessary functionality. The use of an object-oriented CDM facilitates schema translation by operational mapping. The operational mapping approach is based on the same principle as the object-based architectures, that is, each component system provides a specific interface consisting of a set of primitive operations.

3.4 Schema Integration

Schema integration is defined as the activity of integrating the schemas of existing or proposed databases into a global, unified schema [BLN86]. In the case of FDBSs, schema integration occurs in two contexts (see Figure 3):

- when integrating the export schemas of (usually existing) component systems into a single federated schema; and
- during database design, as view integration of the multiple user views of a proposed federated database (federated schema) into a single conceptual description of this database (external schema).

In many applications, there is a need to integrate non-traditional component databases that do not support schemas. It is necessary to *generalize* the concept of schema integration to include the integration of such systems. Object-oriented data models can be very useful, since they permit the definition of the conceptual schemas of non-database systems in terms of the operations they support, thus completely hiding the structure of their data.

[BLN86] identifies four main steps in the process of integration: preintegration, comparison of schemas, conforming of schemas, and merging and reconstructing. Translation is considered as part of the preintegration step. In general, a data model to facilitate all steps of the integration task should be semantically rich; it should provide mechanisms for expressing not only the semantics expressed at the local databases but also additional semantics relevant to schema integration (*schema enrichment)*. Furthermore, it should ideally be capable of expressing the semantics of any new local database that might be added to the system in possible future expansions. From this perspective, object-oriented models are especially appropriate.

During the comparison step, the component schemas are compared to detect conflicts in their representation and to identify the interschema relations. The comparison of the schema objects is primarily based on their semantics, not on their syntax. The CDM should be semantically rich to facilitate comparison and should also support abstraction mechanisms to permit comparisons to be made at a higher level of abstraction. The objective of the conformation step is to bring the component schemas into compatibility for later integration. Comparison and conforming activities are usually performed in layers. These layers correspond to the different semantic constructs supported by the model. The fewer the basic constructs supported by the model the fewer the conflicts and the easier the conformation activity. For this reason, object-oriented models which support a single construct (function) for both instance variables and methods are preferable. When only functions are supported, comparison and conformation are performed first for classes (structural conformation) and then for functions (behavioral conformation [Ber91]).

The search for identifying relations or possible conflicts may be guided by the class hierarchy. Instead of comparing all classes in a random manner, classes may be compared following the class hierarchy in a top-down fashion [GSCS93].

As in the translation phase, relations between different classes must be identified. The difference is that now these classes may belong to different databases. Subclassing relations may be specified based on inclusion relations between the extents of the corresponding classes [MNE88, SLCN88]. Assertions may be used to express these relations. The assertions should be checked for consistency and completeness [MNE88, SLCN88]. The identification of relations between classes can also be made by comparing the definitions of classes [SSG+91] rather than their actual extensions.

Most systems use view definition facilities for defining the global schema, during the last step of integration. The creation of the global view is usually performed in two phases. In the first phase, the classes of the component schema are imported or connected, that is, they are mapped to corresponding global classes. In the second phase, classes are combined based on their interschema relations. View definition facilities are described in the following section.

3.4.1 Object-Oriented Views

A view is a way 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. The definition of a view is dependent upon the data model and the facilities of the language used to specify the view. In a relational model, a view is defined as a set of relations, each populated by a query over the relations of the existing databases and sometimes over already-defined view relations. There are as many different approaches to defining an object-oriented view as there are object-oriented data models. In general, an *object-oriented view* is defined as a set of virtual classes that are populated by existing objects or by *imaginary* objects constructed from existing objects. The set of virtual classes defines a schema and the objects that populate them define a (virtual) database described by the schema [HZ90]. Once a virtual class is created, it should be treated like any other class. The classes used in the definition of a virtual class are called *base classes*.

The reference object-oriented model, though rich in facilities for structuring new objects, lacks 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. The most common view facilities added to object-oriented systems are:

(i) facilities for *importing* classes from the local databases into the view. Virtual classes created in this manner correspond directly to existing classes, and

(ii) facilities for defining new classes, called *derived classes*, that do not directly correspond to an existing class.

Importation

A view can incorporate data from other databases via import statements. Once a class is imported, its definition and its instances become visible to the user of the view. Part of the imported data can be hidden either by explicit hide commands [AB91] or by specifying in the import command only the visible functions [DH84]. Import mechanisms differ in whether the importation of a class results in an implicit importation of all its subclasses.

Other types of importation statements import in a single statement classes or entire hierarchies belonging to more than one component database. In essence, these statements combine the importation phase with the class derivation phase. The virtual class may be defined either as the supertype of the top superclasses of each component database [SST92] or by combining those top classes based on their interschema relation [MNE88]. During importation of this sort, basic types, such as integers and strings, can be implicitly unified [SST92].

Other approaches distinguish between the importation of behavior (functions) and the importation of objects [FHM92]. By doing so, local functions may be executed on imported objects and imported functions may be executed on local objects.

Derived Classes

The definition of a virtual class includes the specification of the following three components: (i) the initial members of the class (class extension); (ii) the structure and behavior, that is the functions of the virtual class (class intention); and (iii) the relation between the new class and the other virtual classes.

As we have already pointed out, some systems provide both classes and types. In such systems, a virtual class may have no intensional part. Furthermore, the relations between the derived class and the base classes in such systems, are purely set-oriented (for example relations such as union, difference, etc). Finally, in such systems, the relation between the associated types of the derived class and its base classes must be also specified.

Different methodologies provide different language constructs for specifying the above three components of a virtual class. Most of these constructs define one component directly and leave the other two to be derived by the system. There are three general methodologies:

1. The language provides a variety of class constructors which correspond to the relations between classes supported by the model. These constructors are applied to existing base classes to create new derived classes that have the corresponding relation with the base classes [DH84, Mot87, KDN91, MNE88, SGN93]. This methodology in effect defines explicitly the third component of a virtual class and then implies the other two, namely its population and intention. The most common such constructors are the *generalization* or *superclass* constructor and the *specialization* or *subclass* constructor.

   A derived class which is defined as a *subclass* inherits all the functions of its superclasses. In the subclass, functions may be redefined and new functions may be defined. There is no standard definition of the extension of the subclass. It is generally defined as a subset of the extensions of the superclasses. In [DH84] and [Mot87] (subclassing is called join in this framework), the extension is defined as the intersection of the extensions of the superclasses.

   A derived class defined as a *superclass* inherits the common functions of its subclasses (upwards inheritance). Functions in a superclass may be redefined. The extension of the superclass is defined as the union of the extensions of its subclasses.

2. Derived classes are defined by specifying their population. The population of the derived class is defined as the result of a query on existing base classes. This is the most commonly

used approach [Ber91, MHG+92, KCGS93, HZ90, AB91, SST92, CL92, Ber92, KKS92]. The class intention and the position in the hierarchy may [AB91, SST92] or may not [CL92] be implied automatically by the system. This methodology includes mechanisms for defining classes populated by imaginary objects. Functions defined in a derived class can typically use the functions defined in the base classes.

A complete methodology for inferring both the position of the derived class and its intention, is presented in [AB91]. A class whose population is defined by a selection predicate on some function of the base classes is considered their subclass. A class whose population includes the population of the base classes is considered their superclass. [AB91] also introduces the notion of *behavioral* and *parameterized* subclassing. Behavioral subclassing allows a superclass to include all classes that have a specific property (function). Parameterized subclassing allows the partition of a superclass into subclasses based upon the value of one of its functions. A mechanism similar to parametric subclassing, called *type schemas*, is presented in [CL92].

One important research issue [MHG+92] concerning classes defined by that way is the definition of a query algebra with a minimum number of operators for creating arbitrary derived classes and imaginary objects. This algebra should also support efficient query optimization.

3. The structure (intention) of the derived class is explicitly defined.

Combinations of the above methodologies are also possible, especially in the form of queries that involve class constructors.

When subclassing is used for subtyping purposes (see Section 1.2), some restrictions must be placed on the type of arguments and on the type of the return values of all functions defined in the virtual class by inheritance. These restrictions should be such that every object of a subclass could be used in any context where an object of any of its superclasses could be used. We call these restrictions *subtyping restrictions*. [DH84] defines *superfunctions* as functions that satisfy appropriate subtyping restrictions. In this framework, a virtual class that is defined as superclass can include only superfunctions of the functions defined at its subclasses.

3.4.2 Issues in Integration

In the following, we discuss several subtle issues concerning object-oriented integration.

Resolution Problems and Behavioral Sharing

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. 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. The straightforward solution is to allow the user to explicitly specify which of the applicable functions should be used [AB91]. [SN88] introduces the concept of object *coloring*; the color of an object specifies the class from which the search for a function should begin. If the function is not defined in this class, it is searched for in the appropriate subclasses. This method also identifies the different semantic relations that may hold between the subclasses being generalized and their attributes. These relations are utilized to produce different default treatments of function resolution.

The above discussion refers to behavioral sharing along the inheritance hierarchy and specifies how the correct function is chosen either implicitly by the default resolution mechanism or explicitly by the user. An alternative method to behavioral sharing is introduced in [HZ90]. In this framework, the functions of the virtual class may invoke functions from the base classes, but these functions will be applied not to the new objects but to the objects of the appropriate base class. This corresponds to using the delegation rather than the inheritance method for sharing behavior (see Section 1.2).

Assigning Identifiers to Imaginary Objects

Object-oriented systems associate a unique identifier with each object upon its creation. Accordingly, upon the creation of an imaginary object, an identifier must be associated with it. An imaginary object must be assigned the same identifier at each invocation. Moreover, if an imaginary object is defined as a composite object (see Section 3.1.2) then its identity should be modified when the real objects, that constitute it, are updated. The most common solution [AB91, HZ90, KKS92] is to define the identifier of the imaginary object as a function of the identifiers of the real objects upon which the imaginary object depends.

Resolving Conflicts and Expressing Interschema Relationships

The following outlines the most common ways of resolving conflicts based upon the taxonomy presented in the introduction of this section:

- *Identity conflicts* in object-oriented data models are in general very difficult to resolve since the identity of an object is not based upon the value of some of its attributes but is characterized by an identifier (*oid*) assigned to the object upon creation. Most systems [ADD+91, HJK+92] allow the user to specify which objects are equivalent and should share the same *oid*. [SST92] uses the metaclass mechanism to define a function, called the *same-function*, which is applicable to all objects and specifies object equivalences. The user may appropriately define the *same-function* so that equivalent objects are treated as the same object.

- *Naming conflicts* are handled in all systems by defining renaming operators.

- *Structural conflicts* correspond to a restructuring of the class hierarchy or to modifications of the aggregate relations. There is no standard method of resolving structural conflicts; examples are presented in [ADD+91, KCGS93, DH84]. [KFM+95] proposes a method of resolving structural conflicts by applying graph operations, called *augmenting* transformations. These transformations are performed on the graphs that represents the conflicting component schemata so that these graphs become isomorphic.

- *Semantic and data conflicts* are resolved by defining an appropriate function in the virtual class [DH84, ADD+91, KCGS93].

There is no systematic way of expressing interschema relations in the global schema. The most common relations expressed are those that correspond to semantic relations directly supported by the object-oriented model. The most common such relations are the subclass, superclass and aggregation relations [KDN91, AB91].

3.5 Advantages of Adopting an Object-Oriented CDM

We enumerate below some of the distinctive characteristics of the object-oriented data model that render it suitable to serve as the CDM. The usefulness of these characteristics has been demonstrated throughout this section, and the following listing serves as a final recapitulation.

1. The object-oriented data model is semantically rich, in that it provides a variety of type and abstraction mechanisms. It supports a number of relations between its basic constructs which are not expressed in traditional models.

2. The object-oriented data model permits the behavior of objects to be captured through the concept of methods. Methods are very powerful because they enable arbitrary combinations of information stored in local databases. For example, if books with similar topics exist in different databases, a method can be defined in the global schema that eliminates duplicates, sorts different editions, translates titles to a common natural language (e.g., English), etc.

3. The object-oriented model makes it possible to integrate non-traditional databases through behavioral mapping.

4. Since the actual storage and retrieval of data is supported by the underlying local systems, there is no important performance degradation of the overhead of supporting objects in the conceptual CDM.

5. Finally, the metaclass mechanism adds flexibility to the model, since it allows arbitrary refinements of the model itself, e.g., additions of new relationships.

4 Object-Orientation and Transaction Management

This section discusses the impact of object-orientation on transaction management in multidatabase systems. In Section 4.1, the current trends in transaction management are reviewed to provide a perspective on multidatabases and object-oriented transaction management methods. Section 4.2 relates object-oriented approaches to the reviewed literature. Section 4.3 introduces the specific challenges of transaction management in multidatabases, many of which motivated the new trends. In Section 4.4, the object-oriented approach is applied to transaction management in multidatabases. In this section, we will use the term *method* rather than *function* since this is the term most commonly used in the transaction management literature. Also, the terms *class* and *type* will be used interchangeably, since the subtle differences between the two terms, although central to database modeling, do not affect transaction management techniques.

4.1 Trends in Transaction Management

A database consists of a set of named data items, while a database state is an assignment of values to these data items [Pap86]. Not all possible combinations of values represent a legal database state. For example, a state that represents a negative balance in a bank database or an overbooked flight in an airline database is not a legal state. These real-world restrictions are called *integrity constraints* of the database. A database state that satisfies the integrity constraints is a consistent state.

A *transaction* is an execution of a program that consists of a sequence of operations that access and manipulate database items. In the traditional model, transactions consist of simple read and write operations and have a single begin and commit point. Individual transactions maintain database consistency; that is, if they are applied to a consistent state, they result in a consistent state. Transactions are executed concurrently and their operations execute in an interleaved fashion, potentially creating an inconsistent database state. Furthermore, transactions may fail while executing. The objective of transaction management is to ensure that the concurrent execution of transactions leaves the database in a consistent state, even in the event of failures.

Classical transaction management deals with executions rather than with specifications of programs (as in concurrent program proofs). We call the execution of several transactions a *history*. A history is correct if it leaves the database in a consistent state. The approach taken to prove the correctness of a history is based on the observation that a serial history (a history that corresponds to a serial execution of transactions) is correct - by induction on the number of transactions involved. That leads us to the following correctness criterion: a history is correct if it is *serializable*; that is, if it is equivalent to a serial history. Failures are accounted for by including in the definition of serializability only committed transactions, which are transactions that have successfully completed their operations. Thus, the definition of correctness may be restated as follows: a history is correct if its committed projection is serializable. In practice, a more restrictive notion of serializability, called *conflict-serializability* is used because there is an efficient graph-based algorithm for testing it. Two operations *conflict* if the result of their execution depends upon the order in which they are processed. Two histories are conflict-equivalent if they consist of the same transactions and they order conflicting operations of committed transactions in the same way. [Pap86, BHG87] offer an excellent treatment of the above issues.

Current research has called the above assumptions regarding transaction correctness and database consistency into question and new approaches to the problem of maintaining consistent databases are under development. We identify the following directions in the development of new transaction mechanisms [RC92, BOH$^+$92, BK91, Elm92]:

- **New Database Consistency Requirements**
  The requirement that database correctness be preserved, which can be alternatively characterized as the preservation of integrity constraints, has been relaxed in various ways [RC92]. For example, while consistency has traditionally been defined with respect to all items in the database, new approaches require consistency only for parts of the database.

- **New Transaction Models**
  In the traditional model, transactions were sequences of simple read and write operations with a single begin and commit point. New transaction models introduce transactions with an *extended structure* that is, transactions that consist of a number of subtransactions related by model-specific relations. Nested transactions [Wei91, BBG89] are an example of this type. Additionally, new models provide *complex operations* on complex data items instead of read and write operations on single-valued items. Furthermore, to model the execution of complex tasks at various heterogenous systems *workflows models* have been proposed. Workflows extend transaction models by adding even more elaborate structure to transactions.

The above directions are by no means orthogonal.

As a result of the above advances, new correctness criteria for database consistency and transaction correctness are under development to provide alternatives to serializability and to the atomicity, consistency, isolation and durability (ACID) properties of transactions.

4.2 Object-Oriented Transaction Management

Conventional database concurrency control can be used in object-oriented database systems (ORION [Kim90], GemStone [B+89]). In this case, each object is treated as a data item and the methods as a set of read and write operations on this data item. Locking algorithms can use objects as the granularity of the lock, and hierarchical-based locking algorithms can take advantage of the class hierarchy. However, many researchers have utilized the particular characteristics of object-oriented systems to introduce new approaches to transaction management. This section discusses these approaches.

4.2.1 Modular Concurrency Control

According to this approach, each object is responsible for the correct execution of the transactions applied to it [HH91, Wei89, SS84]. More than one transaction is allowed to execute in an object simultaneously and transactions may be executed in more than one object. The correct execution of the transactions applied to an object is referred to as *intra-object synchronization* [HH91]. To ensure global database consistency, in addition to intra-object synchronization, there is a need to control the correct execution of transactions not only locally at each object but also globally over all involved objects; this is referred to as *inter-object synchronization* [HH91]. The advantage of separating the intra- from the inter- object synchronization is that each object can individually select the most suitable definition of correctness and algorithm for its preservation. The concurrency properties of each object are defined according to the semantics of its type and operations. These properties can be specified in different ways, for example in terms of acceptable histories by using state machines [Wei89] or in terms of dependencies between the methods of the type [SS84].

Most models employ serializability as their correctness criterion. Each object ensures the serializability of the transactions submitted to it. Then, roughly speaking, global serializability (serializability of all transactions executed at all objects) is ensured if there is a serialization order compatible with the serialization order at each object.

[Wei89] identifies a property P, called the *local atomicity property*, such that if every object in the system satisfies P, then every history is globally serializable. In that case, there is no need for inter-object synchronization. Any local atomicity property must be such that, in satisfying the property, the objects agree on at least one (global) serialization order for the committed transactions. [Wei89] identifies three *optimal* local properties such that no strictly weaker local constraint on objects suffices to ensure global atomicity for transactions. These three properties are dynamic atomicity (a generalization of two phase locking), static atomicity (a generalization of timestamp ordering) and hybrid atomicity (a combination of the other two). In cases of dynamic and hybrid atomicity, global control may still be needed to resolve deadlocks. Finally, combining objects with different atomicity properties does not guarantee global serializability. [SS84] identifies groups of types called *cooperative types* consisting of types whose objects produce compatible serial histories. No inter-object synchronization is needed for objects belonging to cooperative types.

4.2.2 Semantic Serializability and Type-specific Operations

In an object-oriented system, information is represented by typed objects that can be accessed only through a number of predefined type-specific methods. Systems typically exploit this property to enhance concurrency by [SZ89]:

1. *modifying the definition of history equivalence.*
   A history is equivalent to another history when a difference between their results cannot be detected by data operations. Thus, two histories are semantically indistinguishable, even though they may leave objects in slightly different states, as long as these differences cannot be detected by operations.

2. *permitting operations to be characterized in finer detail than simply as reads and writes.*
   Most approaches also consider arguments and results as part of operations. This result in greater concurrency because it allows for more precise definitions of conflicts. The most common definitions of conflicts in the object-oriented database context are: (1) *Commutativity-Based:* Under this definition, two operations conflict if they do not *commute*. In [BR88] commutativity for complex objects is defined based on the structure of objects. Two operations on an object O commute, if they do not affect the same components objects (e.g., instance variables) of O. [Wei88] defines commutativity as *forward commutativity* and *backward commutativity*. The definition is given in terms of state automata that describe the acceptable sequence of operations by each object depending on its type. Let S be a sequence of operations and s a state, then let $S(s)$ stand for the state reached after applying S on s. Two sequences of operations S and T *commute forward* if, for every state s of the object in which T and S are both acceptable, $T(S(s)) = S(T(s))$ and $S(T(s))$ is an acceptable state; and *commute backward* if $T(S(s)) = S(T(s))$. The definition of commutativity chosen for the conflict relation determines the recovery algorithm used. An intention list algorithm is used with a conflict relation based on forward commutativity, while an undo log algorithm is used with backward commutativity. (2) *Dependency-Based:* Under this definition, two operations conflict if one depends on the other. A binary relation R on operations is a *dependency relation* [HW91], if for all sequences of operations T and S, and all operations p, such that S and p are acceptable after T, and no operation q in S depends on p (e.g., $(p, q) \notin R$), then it should be acceptable to do S after p. An example of a dependency relation is the *invalidated-by* relation [HW91], where operation p invalidates operation q if there are sequences of operations T and S such that $T \circ p \circ S$ and $T \circ S \circ q$ are acceptable but $T \circ p \circ q$ is not ($\circ$ is the concatenation operation).

   Most algorithms used for ensuring concurrency are locking schemas where non-conflicting operations are assigned locks with compatible modes.

The above description does not clearly specify which are the primitive operations of a transaction. Traditionally, the primitive operations are read and write operations, but object-oriented transaction management techniques differ on how they define them. A primitive operation must be implemented *atomically* by the system (for example using mechanisms as semaphores or monitors). Some researchers [Wei89, SZ89] define methods as the primitive operations implying that methods are performed atomically by the system. This is not a realistic assumption, especially when methods invoke methods in other objects, which is necessary when complex objects (objects that have other objects as instances) are supported. In addition it provides less concurrency than the concurrency provided by approaches [HH91] that define methods as transactions of low-level primitive operations.

4.2.3 Nested Transactions

There is an implicit nesting in object-oriented transactions imposed by the way methods are invoked. Methods may invoke other methods leading to a nested transaction structure. The objective is to allow methods to exhibit internal parallelism by exploiting their semantics. A variation of nesting is introduced in [Ska91, SZ89] as a conceptual framework for modeling design applications. The nesting is not the result of the invocation order of methods but it is based on the application requirements. A design task is modeled as a nested structure of transactions, where the root is an atomic transaction, and the nodes are either atomic transactions or *Transaction Groups* (TG). The members of a TG are called *cooperating transactions.* While atomic transactions are consistent, members of a cooperative transaction may or may not be individually consistent. Synchronization is modular and is controlled at two levels by TGs and by objects. Objects synchronize atomic transactions. TGs produce consistent histories of their members, where consistency is defined in terms of *semantic patterns.* A pattern is a sequence of operations that represent semantic actions and it preserves consistency within or among objects. A history of cooperating transactions is consistent if it satisfies the semantics patterns that apply to the group.

4.3 Transaction Management in Multidatabases

Transaction management in a multidatabase system is performed at two levels, at a local level by the pre-existing transaction managers of the local databases (LTMs), and at a global level by a global transaction manager (GTM) superimposed on them. There are two types of transactions, local and global transactions. *Local transactions* access data managed by a single local database system and are submitted to the appropriate LTM outside the control of the GTM. *Global transactions* may access data in more than one local database system, and are submitted to the GTM, where they are parsed into a number of *global subtransactions* each of which access data in a single local database. These subtransaction are then submitted for execution to the appropriate LTM. Global subtransactions are viewed by an LTM as ordinary local transactions. The GTM retains no control over the execution of global subtransactions after their submission to the LTMs and can make no assumptions about the LTMs.

4.3.1 Multidatabase Consistency Criteria

In a multidatabase, a *local history* includes all local and global subtransactions executed at a local site, while a *global history* includes all local and global transactions. The most commonly used criterion for ensuring consistency is based on conflict-serializability. The goal is to ensure *global serializability*, that is serializability of global histories, under the assumption that all local histories are serializable. In general, to ensure global serializability it suffices to ensure that there is a serialization order for the global transactions that is consistent with all local serialization orders. Ensuring global serializability is difficult because the GTM has no knowledge or control over the serialization order of local histories. Local transaction can be the cause of indirect conflicts between global transactions that do not conflict directly.

One practical solution to the problem of maintaining global serializability was proposed in [GRS91] and is based on the concept of a *ticket*. A ticket is a special data item stored at each local database. Each global transaction before starting executing at any site, must read the ticket at this site, increment it, and write back the incremented value. These operations force direct conflicts between all subtransactions at the site and thus the ticket value read by a subtransaction indicates its serialization order at this site and can be used by the GTM to ensure global serializability. [ZE93] discusses the properties that *global* transactions must have such that their serialization order at each local site can be determined without using any knowledge about the local transaction management. These properties are realized by enforcing additional conflicts between global transactions.

Both of the above approaches require the enforcement of conflicting operations among global subtransactions at each local site. However, enforcing conflicts may result in poor performance if most global transactions would not naturally conflict. Relaxing global serializability is thus a significant issue for concurrency control in multidatabases. Alternative definition of consistency criteria that are less restrictive than global serializability have been introduced. Examples include two-level [BGMS92], quasi- [DE89] and view-based [ZP93] serializability.

Many researchers have studied different *properties of local histories* and how they affect global serializability [BGMS92]. One interesting such property of histories is *rigorousness* [BGS91]. Rigorous histories disallow conflicts between uncommitted transactions. The commitment order of the transactions in a rigorous history determines their relative serialization order, thus the GTM can ensure global serializability by ensuring that the order of commitment of transactions at all local databases is compatible.

4.3.2 Atomicity and Failures

Even a greater challenge than satisfying global serializability is maintaining the atomicity of a global transaction in the presence of failures, that is ensuring that all its subtransactions either commit all abort. It is proven that there is no algorithm that can ensure atomic global transactions in the absence of a prepare-to-commit state [MKSA92]. Two techniques are used for handling non-atomic commitment. The first technique attempts to ensure atomicity of a global transaction by trying to commit its aborted subtransactions. To do so, either an aborted subtransaction is resubmitted for execution as a new subtransaction of the original transaction (retry approach) or only the write operations of the aborted transaction are resubmitted (redo approach). The second technique attempts to undo the results of a committed subtransaction by executing a compensating transaction at the corresponding local site that undoes from a semantic point of view the effects of the subtransaction [BGMS92].

4.3.3 Extended Transaction Models

The traditional transaction concept is very difficult to maintain in a multidatabase system. As we have already pointed out, it is very hard to ensure global serializability. Moreover, due to local autonomy, a local database is entitled to delay or even reject a global subtransaction. Thus global transactions tend to be long-lived and error-prone and the traditional transaction model seems inadequate to model them efficiently. Moreover, global transactions often model complex tasks with arbitrary dependencies among their subcomponents. Many researchers have proposed extended transaction models and flow of control structures that take into account the above considerations

[ELLR90, GHK93].

4.4 Bringing the two Concepts Together

Although research in object-oriented transaction management and research in multidatabase transaction management are being performed in parallel, it seems that they both lead to what we call layered transaction management. Layered transaction management is introduced in Section 4.4.1, where the common concepts between object-oriented and multidatabase transaction management are discussed. In Section 4.4.2, we describe how ideas pertinent to object-oriented transaction management can be adapted for multidatabase transaction management. Since this is a new research direction, the following sections are only a first attempt to address the issues involved.

4.4.1 Layered Transaction Management

Traditional transaction management coped with the problem of maintaining consistency in a single database. Research in transaction management in multidatabase systems and research in transaction management in object-oriented database systems have independently introduced the notion of layered databases, where each object (local database) is a single database responsible for its own consistency, and where transactions span more than one single database. The structure of these systems has lead to layered transaction management; local or intra-object transaction management and global or inter-object transaction management.

The traditional treatment of layered transaction management is based on the sole assumption that each local database accepts correct histories, e.g., histories that maintain local consistency. Then, properties of histories or local transaction managers are identified such that global consistency is maintained. Research to identify properties of local histories sufficient for maintaining global consistency is being pursued in both the multidatabase and the object-oriented database communities [BGS91, Wei89].

4.4.2 Adapting Object-Oriented Techniques

In this section we discuss the effect of object-orientation on multidatabase transaction management. We identify two important impacts. The first is the use of object-oriented techniques to implement extended transaction models. The second is the use of semantic information. In the following, we elaborate on them.

One application of object-orientation in multidatabase systems is the use of object-oriented techniques to implement extended transaction models. Under this implementation, transactions are modeled as objects and their interactions as the methods of these objects [HHZ+92, BOH+92]. Flat transactions (transaction with a single begin and commit point) correspond to simple objects and extended transactions correspond to complex objects. Implementing transactions as objects requires neither an object-oriented common data model nor an object-based architecture.

In particular, most systems that support extended transactions model their transactions as active objects [BOH+92, MHG+92]. *Active objects* [BOH+92, DBM88] are defined as objects capable of responding to events by triggering the execution of actions when certain conditions are satisfied. An *event-condition-action* (ECA) rule specifies the events that are to be monitored, the conditions that must be fulfilled, and the actions that are to be executed. Transactions involving active objects are intrinsically nested since events may be detected while a transaction is being executed in that object and thus the corresponding rule is spawned as a nested transaction.

Research in object-oriented transaction management has advanced the use of semantic information in transaction management (see Section 4.2.2). Principles and techniques from this area may be used to produce more efficient transaction management in multidatabase systems that support an object-oriented common data model. Since each database object comes with a specific set of methods, semantic serializability can be employed as the correctness criterion. Moreover, type-specific operations may be utilized to provide more appropriate definitions of the conflict relation.

In the special case where the operational mapping approach is used in translation, then each object in a local system provides a number of predefined primitive operations. In that case, the interface of a local system is a set of methods [KFM+95] rather than read and write operations as in traditional multidatabase transaction management. These methods are assumed to always ensure local consistency constraints if executed in a (locally) serializable way.

As regards the maintenance of atomicity by employing the undo approach, multidatabase systems with an object-oriented common data model allow compensating actions to be defined at the method level. Since each class has only a specified set of methods, it is easy to define a compensation method for each one of them.

Finally, semantic information can be used during query processing to produce subtransactions with known interdependencies based on the way the target class is defined. For example, to support parallelization within a transaction, if a query has as its target a superclass whose extension is defined as the disjoint union of the extensions of two subclasses in component databases, the extensions of the two subclasses can be computed concurrently. This approach of combining query processing and transaction management though promising has not yet been fully explored.

5 Case Studies

We conclude this review with a comprehensive study of existing object-oriented multidatabase projects. The projects presented in this section serve as a means to demonstrate how the issues analyzed in the previous sections are being handled in practice. Their inclusion does not, by any means, imply that there are no other systems at least as important as those presented here. The previous analysis applies to other systems as well.

Pegasus provides an interesting, though yet not well-formalized, approach to integration and a practical treatment of query optimization. The *ViewSystem* provides a comprehensive and consistent approach to performing integration by using constructors. *OIS(CIS)* serve as representatives of the operational mapping approach. *EIS/XAIT* proposes an object algebra for performing integration by queries and also an application-specific transaction management model. *DOMS* is the the only system whose architecture corresponds directly to the object-based architecture and the proposed standards. In addition, DOMS supports "customized" transaction management. *UniSQL/M* provides a detailed classification of conflicts and a number of conflict resolution techniques. *Carnot* offers a knowledge-based approach to integration. Although *Thor* is not a multidatabase system, it is included in this section to show how sharing of information from non pre-existing systems raises new research issues. *FBASE* offers an example of the use of class hierarchies in integration, while *InterBase** provides a complete transaction specification language for its extended transaction model. Finally, FIB introduces the notion of *classification*, which is based on the structural properties of its proposed data model, and can lead to the automation of the integration process.

The presentation of the systems is structured as follows. The "*System Architecture*" section describes the structure of the system. The CDM is described in the section titled "*Common Data Model*". In the "*Translation and Integration*" section, we describe the processes of translation and integration following the analysis in Section 3. In the "*Query Processing*" section we describe issues relevant to the execution of global queries. Transaction management is studied in the "*Transaction Management*" section. Finally, the section entitled "*Important Features*" emphasizes the main characteristics of the system reviewed.

5.1 Pegasus

Pegasus [ADK+91, ADD+91, AAD+93] is a multidatabase system being developed by the Database Technology Department at Hewlett-Packard Laboratories. Pegasus provides access to native and external autonomous databases. A *native* database is created in Pegasus and both its schema and data are managed by Pegasus. *External databases* are accessible through Pegasus, but are not directly controlled by it.

System Architecture: The functional layers provided by Pegasus are shown in Figure 4 adapted from [ADD+91]:

- The *intelligent information access layer* provides such services as information mining, browsers, schema exploration and natural language interfaces.
- The *cooperative information management layer* deals with schema integration, global query processing, local query translation, and transaction management.
- The *local data access layer* manages schema and command translation, local system invocation, network communications, data conversion and routing.

Common Data Model: The model, based on the Iris object-oriented model, consists of three basic constructs: objects, types, and functions. Types is the Iris term for classes. Types are organized in a directed acyclic graph that supports generalization and specialization and provides multiple inheritance. Iris uses functions to model both instance variables and methods. Properties of objects, relationships among objects and computations on objects are expressed in terms of functions. Pegasus use the same language, called HOSQL (Heterogeneous Object SQL), both as a data definition and as a data manipulation language. HOSQL is a superset of OSQL, which is the language of Iris, and is query-oriented. HOSQL provides nonprocedural statements to manipulate multiple databases and also supports statements for creating types, functions and objects both in Pegasus and in the component databases. Furthermore, HOSQL provides for attachment and mapping of schema of local databases. Specification of types and functions can also be imported from underlying databases.

Translation and Integration: An external database is represented in Pegasus by its *imported schema* (which corresponds to the component schema of the extended architecture presented in Section 3). The translation from the native schema to the imported schema and the importation of the external schemas are performed in a single step, using the view mechanism of the HOSQL language. The importation facilities are provided in a modular fashion; for each external data model a separate module can be developed, sold and installed independently. A technique for importing automatically an external relational schema is described in [AAK+93].

The imported classes are called *producer types*. Their extension is defined by a special kind of query, called *producer expression*, over a base type called *producer set*. The producer expression defines the instances of the producer type based on some stable and identifying literal-valued property for each entity in an external database. *Oid*s are fabricated for instances of the producer types and have a suffix value taken from the producer set and a prefix unique to the producer type. The functions of the producer type are called *imported functions* and are mapped to properties or relations in external data sources. A characteristic of the function importation mechanism is that a program written in some general-purpose programming language and compiled outside Pegasus may be linked to an Iris function, called *foreign function* [CL88].

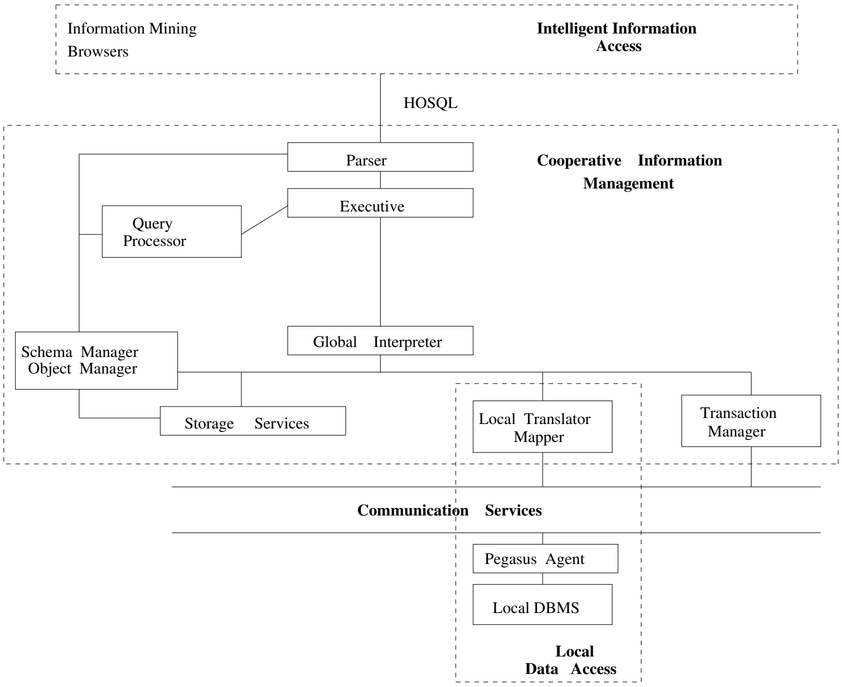

Figure 4: Pegasus System Architecture

Pegasus approach to integration distinguishes between the views of the data administrator and of the end user. This distinction is supported by defining two kinds of types: unifying types and underlying types. Administrators see both kinds whereas users see only underlying types. Producer types are *underlying types*. Each (underlying) type has a unifying type. The initial assumption is that every type is its own unifier but unifying types and their instances can also be defined by combining more than one underlying types using HOSQL statements. Pegasus supports *unifying inheritance*; every function defined for a type is also defined for its unifying type. Resolution problems are resolved explicitly by the administrator who defines a *reconciler* algorithm for each overloaded function. The reconciler algorithm specifies which of the applicable functions will be

used.
Pegasus handles the following types of conflicts:

- *Semantic Conflicts* (called *Domain Mismatch*) are handled by defining appropriate functions at the unifying type.

- *Schema Conflicts :*
  Naming Conflicts referring to function synonyms are solved by defining alias. Additionally, names of functions and types can be prefixed by their database names to prevent ambiguities.
  Structural Conflicts (called *Schema Mismatch*) can be handled by defining adequate imported functions. An example is presented [ADD+91] for handling the conflicts introduced when the same concept is represented by a function in one local schema and by a class in another. In this example, a function is created in the global schema that returns the value of the local function or the associated object of the local class.

- *Identity Conflicts* (called *Object Identification*) are resolved by allowing the user to specify equivalences among objects.

Query Processing: The user issues HOSQL queries against the unified schema of Pegasus. These queries are decomposed into a set of (possibly parametric) subqueries, each one of which refers to data residing in a single external database. In a *parametric subquery* some predicates include parameters whose values are received from other subqueries at evaluation time. Query optimization in Pegasus is either cost-based or heuristic-based, depending on the availability of statistical data. The process of cost-based query optimization is modeled by three characteristics: an execution space, a cost model, and a search strategy. For the case of heterogeneous environments, the traditional execution space is extended to include multi-site joins and the traditional search strategy. Because of the autonomy of the external databases, Pegasus may not be able to control physical parameters such as page I/O and CPU time. Thus, instead of using cost formulae based on physical parameters, it has developed a set of cost formulae based on logical parameters such as data cardinality and selectivity. A set of calibrating databases has been designed to estimate the values of the coefficients in the cost formulae. If the external databases do not provide adequate information for cost-based query optimization, a number of heuristics is used to generate the evaluation plan. One simple heuristics is to put functions that reference one another and belong to the same external database in the same subquery. This minimizes invocations of an external database. For query evaluation, each decomposed subquery, after being translated into the DML of its external database, is sent to its external DBMS for evaluation. The result is returned to the central query processor of Pegasus, and is used to drive other subqueries which make reference to it.

Important Features:

- treatment of conflicts [ADD+91]

- implementation of foreign functions [CL88]

- cost-based or heuristic-based query optimization, depending on the availability of statistical data [AAD+93]

5.2 ViewSystem

The KODIM (Knowledge Oriented Distributed Information Management) [KDN91] project at GMD-IPSI is mainly concerned with the (dynamic) integration of heterogeneous and autonomously administered information bases. An object-oriented environment, called ViewSystem, has been developed as a first prototype. The ViewSystem provides an object-oriented query language with extensive view facilities for defining virtual classes from base classes. The ViewSystem is implemented in an object-oriented environment, namely the Smalltalk environment, and in this way benefits from a large set of tools and reusable software.

Common Data Model: The CDM, called the VODAK data model [DKT88], consists of four basic constructs: instances (or objects), types, classes, and methods. Classes are not templates for creating objects but rather an abstraction for naming collections of objects and associating a number of methods with each collection. Types are templates for defining the structure and behavior of their instances and are organized in a subtype hierarchy (see Section 3.1.2). Classes are related by a number of *semantic relationships*, such as specialization, generalization, grouping and aggregation. These semantic relationships have a set-theoretic counterpart indicating how the objects of semantically related classes correspond to each other; namely: Specialization corresponds to Subsetting, Category Generalization to Disjoint Union, Role Generalization to Overlapping Union, Grouping to Power Set, and Aggregation to Cartesian Product. The query language, called DML, is programming based and set-oriented. Queries are directed against classes of interest and return the set of instances satisfying a qualification predicate. Queries may be nested to arbitrary depth, i.e., a query may occur at any place in the qualification part where a set-valued term is allowed.

Integration: To support semantic integration the VODAK model allows the definition of virtual classes called *intensional* classes. There are two kinds of intensional classes, *external* and *derived* classes. An external class is the VODAK representation of an information unit imported from an underlying database. Derived classes are constructed using a repertoire of *class constructors* from a number of base classes. Derived classes in some sense correspond to relational views. The only difference is that derived classes can have methods attached to them which are written in an object-oriented programming language. There is an one-to-one correspondence between class constructors and semantic relationships; when an operator is applied to a number of classes, it establishes a new (derived) class having the corresponding semantic relationship with the argument classes. The methods of the derived class are computed using the methods of the argument classes. The ViewSystem also offers a concept to support the modularization of views. Different views are organized in different modules. A *module* M consists of a number of classes, an input interface, which is a list of all imported methods of all classes imported to M, and an output interface which consists of all methods exported from M.

Query Processing: The ViewSystem identifies two different ways of performing query processing in the presence of derived classes. One way is to materialize all derived classes that are affected by the query. The other way is to get rid of derived classes by transforming the query into an equivalent set of subqueries which refer to external and base classes only. The ViewSystem takes a *hybrid approach*; it lets the kind of derived class determine whether materialization or query transformation is more appropriate. Aggregation operators, such as RoleGeneralization, Grouping and Aggregation, raise the problem of identifying the corresponding objects, and for that reason materialization is favored over query transformation. On the other hand, query transformation is more preferable in the case of Specialization and Category Generalization because the objects of the derived classes have a unique counterpart in the related classes and thus a query can be split into a number of subqueries such that no subquery is faced with the identification problem.

Important Features:

- it is embedded in an object-oriented programming environment and benefits from reusable software
- provides a concrete methodology for creating virtual classes based on a set of class constructors
- offers a hybrid approach to query processing
- allows for organizing views in different modules

5.3 OIS

The Operational Integration System (OIS) [GCO90] is a generalized integration tool which provides the application environments with a uniform interface for accessing data managed by heterogeneous systems. These systems are expected to be file systems, DBMSs, Information Retrieval Systems, remote Databank services or ad hoc applications. OIS has been partially developed in the framework of the Esprit Project 2109(TOOTSI). OIS is similar to CIS and are both described together in the following section (section 5.4).

5.4 CIS

The Comandos Integration System (CIS) [BNPS89, BNPS88] has been implemented as part of the ESPRIT project COMANDOS. It has been used for integrating several different application environments, including relational DBMSs, Graphical Databases and public Databanks.

System Architecture: The system architecture is depicted in Figure 5. A *client* is an application based on CIS, that accesses services provided by one or more servers. A *server* is the abstraction of (part of) a pre-existing application. The role of a server is to make available to clients a uniform object-oriented interface on top of the pre-existing application.

Common Data Model: The CDM, called Abstract Data Model in the CIS framework (or Integration Data Model (IDM) in the OIS framework), consists of the following basic constructs: objects, classes, methods (called operations) and instance variables (called definitional properties). The notion of a class is both intentional and extensional (see Section 3.1); a class is consider both as a set of objects and as a pattern for creating them. The model supports a number of additional features (see Section 3.1) such as independent and dependent objects, constant properties, the possibility of defining a property as the inverse of another property, and the notion of key. A property can be defined as optional or mandatory. Furthermore, properties can be variant, that is they can

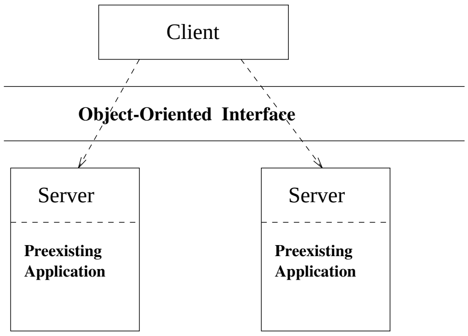

Figure 5: CIS System Architecture

be instances of several different classes. The query language, QL, is logic and query-based. Queries are issued against all instances (not members) of a class. Thus, the objects resulted from the query belong to the same class and the type of the result is known at compile time.

Translation: [BNPS89] introduce the concept of *Operational Mapping*. The Operational Mapping approach (section 3.3) is based on defining correspondences between operations instead of defining correspondences between data elements. An object-oriented *Abstract View* is defined on top of each local database system using the Abstract Data Model (the Abstract View corresponds to the component schema of the extended architecture, see Section 3). The Abstract View is defined as a set of operations on a set of abstract data. The Operational Mapping is defined as the implementation of these Abstract Operations in terms of primitive operations of the underlying systems. The Abstract Operations is a set of predefined generic operations classified as : (i) operations on classes, which include insertion and deletion of an object from a class and inspection of the instances of a class and (ii) operations on objects, which deal with object property manipulations. More complex operations can be built using the primitive ones. Some local systems may not be able to implement some of the generic operations. The *drop clause* specifies which operations cannot be applied to the related classes.

An implementation of the generic operations must be provided for each local system. The implementation of these operations realizes the OIS *object-at-a-time interface*, that is, an operation always returns a single object. The object-at-a-time interface uses *oid*s. The validity of an *oid* is bound to the duration of a specific client/server interaction. A special data structure is allocated by the server whenever an object is activated. An object is active if there is at least one client which has a reference to it. An *Active-Object-Table* contains references to the data structures of all active objects. An *oid* is the address of an entry in this table. The restriction that *oids* are valid only during a single client/server interaction makes the implementation of *oids* possible even when the component systems do not support the identification of objects directly (e.g., the component systems include file or graphical systems).

Query Processing: The CIS Query Processor (QP) at the client parses queries written in QL.

Since no integration is supported, queries access data only at one server, thus the resulting parse tree is sent to the QP of that server. The QP at the server, after completing the type checking, generates the query evaluation tree according to optimization rules. Finally, the query evaluation is performed using the object-at-a-time interface. The optimization adopts heuristic techniques that use the information stored in the data dictionary at the server.

Important Features:

- the introduction of the concept of operational mapping along with an implementation [BNPS89, GCO90]

5.5 The EIS/XAIT Project

The Object Management System (OMS) [PSH91, HZ90] is an object-based interoperability framework for Engineering Information Systems (EIS) designed at Xerox Advanced Information Technology (XAIT).

Common Data Model: The CDM (called FUGUE) is an object/function model that consists of three basic constructs: objects, functions and classes (called types). The query language is set-oriented and comprises a set of built-in functions that apply to collection objects. These functions take instances of specific set types as input arguments and produce set types as output. Built-in functions include functions for predicate-based selection of objects (Select), collection manipulation (Union, Intersection, Difference and Flatten - i.e. unnest collections), and creation of new types and instances. They also include an *object join* (OJoin) function that when applied to classes A and B it produces a class populated by pairs of objects (a, b) where a $\in$ A and b $\in$ B (this operation is similar to a form of subclassing). Built-in functions are mapped to local functions.

Integration: The global schema is defined through a view mechanism. The population of the virtual classes (called *derived types*) is defined by a query over the base classes. The objects that populate the virtual class are always assigned new *oid*s. The functions of a derived class may invoke functions from the base classes, but these function will be executed in the scope of the class where they were originally defined, that is, they will be applied not to the new objects but to the objects of the appropriate base class (delegation). The procedure that implements a function has its own view. Each client that requests the application of a function is assigned a view that provides the context in which it will operate.

Transaction Management: In [HHZ+92] the idea of cooperation between transactions in the context of Engineering Environments (see Section 4.2.3) is further pursued and a framework is developed for coordinating the different groups in an integrated organization. The model supports a hierarchy of groups. The topmost group represents the whole organization. Transaction management is implemented modularly, as a *Transaction Manager Hierarchy* which consists of a set of local transaction managers and a global transaction manager that coordinates the local managers. The algorithm employed by the global transaction manager is fixed, and is delivered with the framework. Each group provides its own local transaction manager. These local transaction managers have two parts: a group specific *protocol* and a uniform *capability* (over all groups) for coordinating with neighboring transaction managers. The protocols can be written in any convenient specification language. Each group provides its own correctness criterion relative to its protocol. Global correctness is relative to these individual protocols but the relation has not yet been formalized.

The long term goal is to provide a toolkit for building customized transaction managers. The toolkit will include the algorithm for the global transaction and a number of commonly-used protocols. Organizations will describe their structure and will either write their own protocol or select one from the ones provided. Note that the above transaction model relaxes the requirement for autonomy of local sites in two ways. First, the algorithm employed by local sites (groups) is known to the global transaction manager, and second, transactions of different groups can cooperate and share intermediate results.

Finally, since the transaction model is designed to be used with an object-based system it provides high-level operations that correspond to the functions supported by the system classes. Transactions, functions, protocols and transaction managers are modeled as objects.

Important Features:

- the definition of view facilities [HZ90], also note that sharing is implemented by delegation
- extended transaction model that supports cooperation between transactions and user-specified correctness [HHZ+92]

5.6 DOMS

The Distributed Object Management System (DOMS) [BOH+92, MHG+92] that is being developed at the GTE Laboratories, is an object-oriented environment in which autonomous and heterogeneous local systems can be integrated and native objects can be implemented. The local systems are not limited to database systems but may be conventional systems, hypermedia systems, application programs, etc. A prototype DOMS was implemented connecting Apple Macintosh Hypercard applications, the Sybase relational DBMS and the ONTOS object DBMS. The prototype supports a simplified version of the data model and language and does not currently support concurrency control and recovery facilities but supports a limited form of "distributed commit".

System Architecture: DOMS architecture is depicted in Figure 6 adapted from [MHG+92]. The architecture is built based on the general principles of the Distributed Object-based Architectures described in Section 2. *DOMs* serve as object managers. A *Local Application Interface* (LAI) provides an interface between a DOM and a local system that allows the DOM to access local data and the local system to make requests to access objects from other local systems or to use DOM services.

Common Data Model: The CDM, called FROOM (Functional/Relational Object-Oriented Model), consists of three basic constructs: objects, functions and types. Functions model both state and behavior. The subtype relation is determined implicitly; any type that supplies the interface required by a type T is a subtype of T. FROOM distinguishes between implementation and interface, thus objects of the same type may support different implementations of the same function. FROOM supports Event-Condition-Action (ECA) rules (see Section 4.4.2). Rules, events, conditions, and actions are defined as object types. The definition of FROOM includes an object algebra that resembles an extended relational algebra. The object algebra includes a set of high-level functions, which, as in FUGUE, are defined for collections of objects, and create new collections as results. The functions provided by the algebra include functions that correspond to operations of the relational algebra (Select, Project, Join), standard set operators (Union, Intersect, Difference), functions for creating new *oid*s, and other miscellaneous functions. Current research is pursuing the definition of a more primitive "RISC" object model [MH92]. This model provides the definition of a small set of fundamental concepts to allow the definition of other object models (including FROOM) in terms of this single set. The basic approach involves incorporating at the object level constructs that are representation or meta-level constructs at other models, for example, types for describing methods, state, and object identifiers.

Integration: Integration is accomplished by defining views through queries. When objects involved in the query belong to local attached systems, DOMS maps these queries through object algebra expressions into expressions in the local query languages of the attached systems. Future work will tackle the difficult issue of providing general facilities for creating arbitrary objects and functions using algebra expressions. It will also address the problem of determining an optimum set of algebra functions for use in query optimization.

Transaction Management: The approach taken by DOMS is to identify an extended transaction model that would capture the capabilities of most extended transaction models, so as to provide the basis for a *programmable transaction management* facility. In this context, a Transaction Specification and Management Environment (TSME) has been suggested in [GHK93, GHKM94]. The TSME is a toolkit that supports the definition and construction of specific extended transaction models corresponding to application requirements. It provides a transaction specification language and a programmable transaction management mechanism that configures a run-time environment to support the specified transaction model.

DOMS transactions consist of a set of flat transactions, together with a set of transaction dependencies among them. Transactions support operations on the following types of objects:

- Local objects that represent data and functionality in local systems that support transactions.
- Local objects that represent data and functionality in local systems that do not support transactions (transactionless systems) but instead provide only primitive atomic operations.
- Native objects whose state and behavior are maintained by DOMS.

DOMS models transactions as objects. A *simple transaction object* is a flat transaction that issues operations only on native objects, or issues operations only on objects that are in the same local database system, or issues only a single atomic operation on a transactionless system. DOMS supports two general classes of extended transactions: multi-system and multidatabase transactions. *Multi-system transactions* are extended transactions that have constituent flat transactions. *Multidatabase transactions* are a special case of multi-system transactions in which flat transactions are submitted only to native objects or to objects supported by local database systems. In addition, local transaction can be executed autonomously at the local sites. Multidatabase transactions follow the traditional model of heterogeneous transactions presented at section 4.3

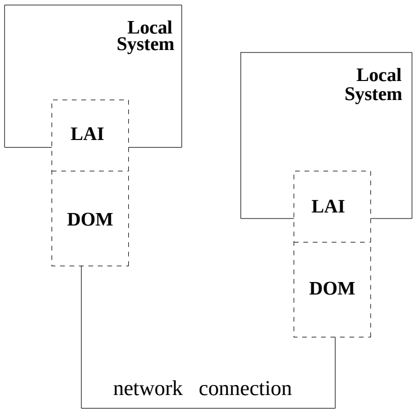

Figure 6: DOMS System Architecture

Multi-system transactions are modeled as complex transaction objects. Complex transaction objects are defined from simple transaction objects using *Dependency Descriptors*, (DDs). DDs are FROOM functions that describe the interrelations between transaction objects in terms of transaction dependencies. DDs will be implemented using ECA rules. The implementation of DDs for multidatabase transactions is complicated by the fact that the serialization orders of the transactions at each local site are not known (see Section 4.3). If the local histories are rigorous, DOMS will control the commitment order of the transactions, otherwise DOMS will use the ticket method (see Section 4.3).

Important Features:

- complete framework in the context of Distributed Object Architecture
- support of transaction management that includes operations at transactionless systems, however transaction correctness and recovery is not formalized especially under the presence of local transactions
- an attempt to apply state-of-the-art knowledge at all parts of the system and include most of the features that appear in the literature

5.7 UniSQL/M

UniSQL/M [KCGS93, Kim92] is a heterogeneous database system, being developed at UniSQL Inc, that allows the integration of SQL-based relational database systems and the UniSQL/X unified relational and object-oriented database system.

System Architecture: UniSQL/M is a full database system in that it supports a database definition language, a database manipulation language, automatic query processing, access authorization and distributed transaction management.

Common Data Model - Translation and Integration: The query language, called SQL/M, is an extension of ANSI SQL that incorporates object-oriented data modeling concepts. SQL/M supports view definition facilities and conflict resolution techniques. No translation is necessary since the data model of SQL/M is a superset of the relational data model. Tables and classes are uniformly called entities, columns and instance values are called attributes, and tuples and objects are called instances. The population of a class is defined using the member-of relation.

The integration of multiple entities (tables and classes) is accomplished by defining a *virtual class*. The population of the virtual class is defined by a query on the base entities. The attributes and the methods are defined by explicitly enumerating them along with their domain. It is the responsibility of the user to define both methods and attributes in conformity with the subtyping restrictions. No special *import* operation is defined. An entity A may be imported as the virtual class $V(A)$, by defining the population of $V(A)$ as equal to the population of A and the attributes and methods of $V(A)$ as equal to the attributes and methods of A. Hiding of attributes and methods can be achieved by not enumerating them.

[KCGS93] provides a taxonomy of possible conflicts along with the resolution techniques used by SQL/M. We present them using the framework introduce in Section 3. $V(A)$ stands for the virtual class that results from the importation of A.

1. *Identity Conflicts* are not discussed. Actually, it is not clear whether the model supports object identity or not.

2. *Schema Conflicts:*

   (i) *Naming conflicts* are handled by using renaming operations.

   (ii) *Structural Conflicts:*

   Since the different constructs supported by the model are entities and attributes (methods are not considered in [KCGS93]), the basic taxonomy can be adjusted as follows.

   (a) When the same concept is represented by different constructs of the data model, namely by an entity and an attribute, then an entity may be split into multiple parts, or two entities may be integrated into one by performing a vertical join.

   (b) When the same concept is modeled by the same constructs then:

   • If the constructs are classes (entities) and have the same intentions, a *union compatible join* is applied. If, in addition, there is an inclusion relation between the extents of the classes then one can be defined as the subclass of the other. In the special case where a missing attribute has an implicit value then a special expression is provided for determining this value. When the extent of a class A is a subset of the extent of a class B and the attributes and methods of A subsumes those of B (meaning they respect the subtype restrictions) then A and B are called *extended union compatible* and an *extended union compatible join* can be used; that is $V(A)$ may be defined as a subclass of $V(B)$. If both entities are tables, then to integrate multiple tables into a single class, *vertical join* is used.

- When the same construct is an attribute then, if one attribute is of a primitive type and the other is a complex object the *projection of the aggregation hierarchy* is used; that is only one of the components of the complex attribute is selected in the virtual class. Default *coercion operations* (e.g., from integer to real) are provided for resolving conflicts between attributes having different primitive types. In addition, a concatenation operation is provided for attributes of the primitive type string. If the attributes of two classes A and B are instances of classes related by a subclass relation that may imply the same subclass relation between A and B. Resolution techniques for resolving conflicts between attributes that are complex objects of different classes are not discussed.

3. *Semantic Conflicts* discussed in [KCGS93] refer to *different representation for the same data.* These different representations include different units, different precision or different expressions. They are handled by *homogenizing representations*, that is, by defining the correspondence between different representations and using arithmetic expressions or look-up tables to convert from one to the other.

4. *Data Conflicts* are not discussed.

Transaction Management: The first release of UniSQL/M presumes two-phase commit support in local database systems but it supports concurrency control, global deadlock detection/resolution, and distributed database recovery [Kim92].

Important Features:

- systematic treatment of conflicts
- commercial database system already released

5.8 Carnot

The Carnot project at MCC [WCH+93, HJK+92, WSHC92, TLM+92] is addressing the problem of logically unifying physically-distributed, enterprise-wide heterogeneous information, coming from a variety of systems including database systems, database applications, expert systems and knowledge bases, business workflows and the business organization itself.

System Architecture: Carnot has developed and assembled a large number of generic facilities. These facilities are organized into five sets of services (see Figure 7 adapted from [WCH+93]):

- *Communication services* provide the user with a uniform method for interconnecting heterogeneous equipment and resources.
- *Support services* implement basic network-wide utilities. An important component of the support services is a distributed shell environment called Extensible Service Switch (ESS).
- *Distribution services* support relaxed transaction processing and a distributed agent facility.

- *Semantic services* provide a global (enterprise wide) view of all the resources integrated within a Carnot-supported system.
- The *Access services* provide mechanisms for manipulating the other four Carnot services

The Extensible Service Switch (ESS): ESS [TLM+92] provides interpretive access to communication resources, local information resources and applications at a local site. The switch can be thought of as a component of a distributed command interpreter that is used to implement heterogeneous distributed transaction execution and generalized workflow control. It may also be viewed as a high level programmable communication front-end for applications in a distributed information system. ESS is constructed on top of *Rossette*. Rossette is a high performance implementation of an interpreter for an Actor-based model [Gul86] enhanced with object-oriented mechanisms for inheritance.

Common Data Model - Translation and Integration: In addition to database schemas, Carnot [HJK+92] considers the integration of knowledge-base systems and process models. Instead of translating the schemas (models) of the local resources into a common data model, Carnot compares and merges them with Cyc [CHS91], a common-sense knowledge base. Cyc, besides its common-sense knowledge of the world, has knowledge about most data models and about the relationships among them. The common language is called *Global Context Language* (GCL) and is based on extended first-order logic.

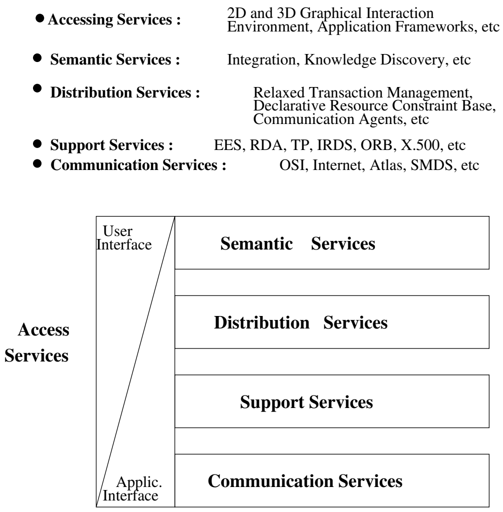

Figure 7: Carnot System Architecture

A resource is integrated by specifying a syntax and a semantics translation between the resource and the global context. The syntax translation provides a bidirectional translation between the local resource management language and GCL. The semantics translation is a mapping between two expressions in GCL that have equivalent meaning. This is accomplished by a set of *articulation axioms*. An articulation axiom has the form $ist(G, \phi) \Leftrightarrow ist(C_i, \psi)$, where ϕ and ψ are logical expressions and *ist* is a predicate that means "is true in the context". This axiom says that the meaning of ϕ in the global context G is the same as that of ψ in the local context C_i. After integration one can access the resources through the global view using GCL. Carnot provides a graphical tool, the Model Integration Software Tool (MIST), that automates some of the routine aspects of model integration.

Query Processing - Transaction Management: Carnot's Distributed Semantic Query Manager (DSQM) [WSHC92] executes queries and/or updates against integrated information resources. DSQM has been implemented as an ESS actor object. DSQM's query graph generator module accepts an SQL string and generates a query graph using information from a global dictionary. The query graph is passed to the semantic augmentation module. This module uses articulation axioms to expand the query graph to include other sources that contain relevant information. If the original query included modifications in a database, the query graph is passed to the relaxed transaction augmentation module, which uses the information stored in a *declarative resource constraint base* to determine which other databases should be modified and creates separate query graphs for each one of the modifications. The separate query graphs are then related to each other using *a transaction graph* that defines the relaxed transaction semantics to be used. A language, based on the ACTA formalism, is being designed and implemented, which will be used to specify the relationships among subtransactions in the transaction graph. An optimal query plan is then generated for each query graph. Finally, the query plans and the transaction graph are passed to the ESS script generator, which generates a script to be executed at each site.

Important Features:

- the use of a common-sense knowledge base as the global schema instead of a data model [HJK+92, CHS91]
- the implementation of the ESS

Object-Orientation in Carnot

Carnot does not follow any of the three dimensions of object-orientation introduced in Section 1.1. It is included in this report because it takes advantage of the development of object technology in the implementation of various of its tools. Such tools include the ESS that is an Actor object; an object-oriented deductive environment called LDL++ used for application development - this can be viewed as providing an object-oriented external schema on top of the global schema (see Section 3) and a 3D visualization tool.

5.9 Thor

Thor [LDS92] is an object-oriented distributed DBMS being implemented at MIT. Thor is intended to be used in heterogeneous distributed systems to allow programs written in different programming languages to share a universe of persistent objects in a convenient manner. Thor is not a multidatabase system since it does not support the integration of pre-existing systems but rather a distributed database system that allows different systems to share information by means of objects of the Thor's universe. A prototype of Thor, called TH has been implemented in Argus [Lis88].

System Architecture: Thor is intended to run in a distributed environment. Some of the nodes are Thor servers, which store the objects in the Thor universe. Others are client nodes where users of Thor run their programs. The Thor system runs frontends (FEs) at the client nodes, and backends (BEs) and object repositories (ORs) at the servers. Every resilient object resides at one of the ORs. A user always interacts with Thor via an FE, which typically resides at the user's workstation. Each FE acts on behalf of a single principal client. An FE is authenticated to the ORs with which it interacts using the Kerberos authentication service. The client program interacts with Thor by executing Thor commands such as start or terminate transactions and run operations. An FE makes use of BEs and ORs to carry out these client requests. FEs and BEs perform operations and understand types. ORs are concerned only with managing the resilient storage for objects. Delays in accessing objects from several different ORs are handled by caching objects at the FEs. Caching also reduces the load at the ORs servers. Another method used to reduce delay is to combine all calls that can be performed at one OR into a larger "combined operation".

Common Data Model: The Thor data model is language-independent in that it is not being embedded in a particular programming language. It provides a type system that allows programs written in different programming languages to share data. A type is defined by a specification; specifications are independent of the programming language used to access the type's objects and of the language used to implement the type. Thor also provides access to objects through both navigation and queries. It supports full indexing for queries over sets of abstract objects. Thor does not support the integration of pre-existing database systems but provides for information sharing among heterogeneous applications through a number of persistent objects which are being shared among the applications.

Query Processing: Queries run at ORs. When a set object is used at an FE, meta-data about the set is sent to the FE, but the elements of the set are not. The meta-data includes information about indexes. Using this information the FE can make decisions about how to carry out the query most effectively.

Transaction Management: Thor is not a heterogeneous DBMS but a homogeneous distributed object-oriented DBMS. Transaction management in Thor is similar to the traditional transaction management in a distributed database system with the difference that the basis of the concurrency control are objects instead of pages or segments. Concurrency control and recovery are provided for individual objects. Objects become persistent only when the transaction that made them persistent commits. Two-phase commit is used as the commitment protocol. The coordination of concurrent transactions from different FEs at the ORs will be accomplished by using an optimistic schema. A primary copy schema will be used for replication.

Important Features:

- a different approach to the problem of handling heterogeneous information, that allows heterogeneous system to conveniently share information stored in the form of Thor's objects

- addressing performance issues, such as object-caching and combined operations, as well as physical storage issues which are not considered by HDBSs since such issues are being handled by the local systems

5.10 The InterBase Project

In this section, we describe two prototype systems developed at Purdue university as part of the InterBase project [BCD+93]. FBASE [Mul92] concentrates on data modeling issues, while InterBase* [ME93] provides complete transaction support. Currently, work is undertaken to integrate the data model of FBASE in the InterBase* system.

5.10.1 FBASE

FBASE [Mul92] is an object-oriented multidatabase system that can be characterized as decentralized.

Common Data Model: The FBASE model uses the core characteristics of the object-oriented model, i.e. classes, objects and methods (functions) to define a class hierarchy appropriate for modeling the schemas of the component systems. The FBASE class hierarchy is depicted in Figure 8 adapted from [Mul92]. Each component database system is considered to be an instance of the predefined class *Database*. Commands such as Create, Insert, Update, Select are predefined methods of the objects of the *Database* class. Each object of the *Database* class is considered to be a collection of instances of the class *Relation*. *Relations* have several predefined methods such as project, Cartesian product, union, minus. The FBASE query language, called Federated SQL (FSQL), is an extension of SQL. It extends SQL in the following ways:

- It allows the specification of remote system data. Remote system data are specified by prepending the name of the remote system to the relation name being accessed.

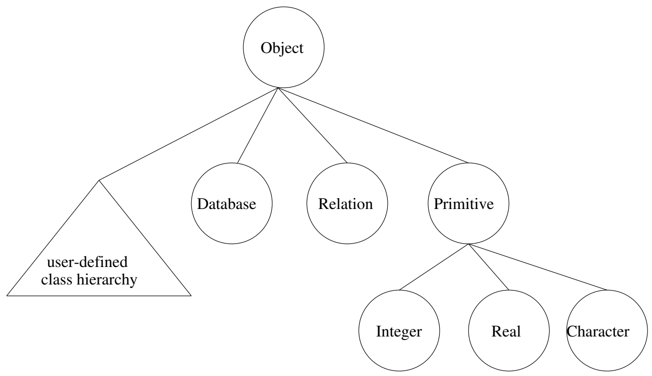

Figure 8: FBASE Class Hierarchy

- Complex objects can be defined and referenced.
- Object methods can be defined and referenced.

Translation and Integration: Each component has a *private schema* that describes the data available to its local users (corresponds to the local schema of the 5-level architecture, see Section 3), an *export schema* that describes the data that other systems may import from it (corresponds to the export schema) and an *import schema* that describes the data at other component systems that the system knows about (corresponds to the export schema of those systems). No federated schema is created. The data structures used to represent the three schemas are stored at each component system as relations. In addition, a special data structure, called directory, contains a list of remote sites that the component system knows about. A component system can be integrated as an importer (i.e. it can execute global queries), exporter (i.e. global queries can access its data), or both, and various degrees of integration can be supported. Special FBASE servers may perform query language translation and data format translation for each different exporter system.

Important Features:

- the class hierarchy provides a uniform way of mapping different data models to the object-oriented model

5.10.2 InterBase*

Currently InterBase* runs on an interconnected network with a variety of hosts such as Unix workstations and IBM mainframes. It supports global application accessing many local systems, including SAS, Sybase, Ingres, DB2, and Unix utilities.

System Architecture: InterBase* consists of four types of components (see Figure 9 adapted from [ME93]):

- *InterBase* Servers* maintain the data dictionary and are responsible for processing global queries.
- *InterBase* Clients* connect to InterBase* servers and issue global queries.
- *Component Database Systems (CDBSs)* are the systems being integrated.
- *Component System Interfaces (CSIs)* act as an interface for the InterBase* servers to the component systems. The CSIs are responsible for translating global queries to the native query language of the local systems, and for translating data from the native format to the global format.

Common Data Model: InterBase* uses the same language, called InterSQL, both as a query language and as a transaction specification language. InterSQL combines IPL [CBE93], the transaction specification language of InterBase, with FSQL used in FBASE (see previous section on FBASE) and adds high level support for atomic commitment. Translation and integration are not currently supported. To access data stored in a local system, the user must issue queries expressed in the native language. The characteristics of InterSQL related to transaction specification are described in the next section.

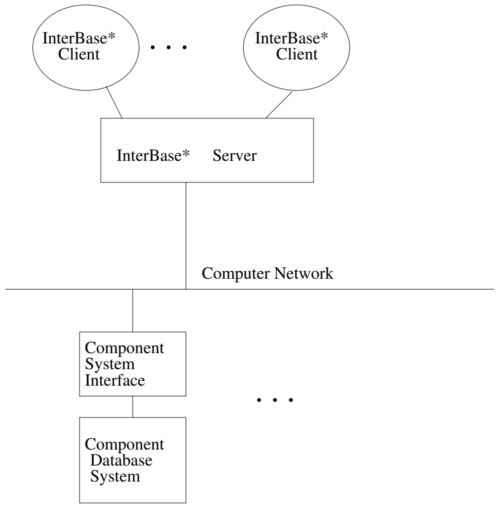

Figure 9: InterBase* System Architecture

Transaction Management: InterBase* supports the Flex transaction model [ELLR90] which is an extended transaction model. A Flex transaction is composed of a set of tasks. For each task, the model allows the user to specify a set of *functionally equivalent* subtransactions, each of which, when completed, will accomplish the desired task. The model also allows the specification of dependencies on subtransactions that might take the form of *internal* or *external dependencies*. Internal dependencies define the execution order of subtransactions, while external dependencies define the dependency of a subtransaction execution on events that do not belong to the transaction (such as the start/end events). Finally it allows the user to control the isolation granularity of a transaction through the use of compensating transactions.

Commitment in InterBase* is specified at the subtransaction level, the desired commitment method is specified for each subtransaction. Each subtransaction may use various and multiple commitment methods. The three basic commitment methods allowed are:

- *Prepare:* The subtransaction is executed to a prepare-to-commit state, where is guaranteed to be committable, but can still be aborted. This method will be provided for systems that support a visible prepare-to-commit state.
- *Reservation (Redo):* The subtransaction is redone (or re-executed) until it successfully commits.
- *Compensation (Undo):* The subtransaction is committed independently of the global transaction, and if the global transaction ultimately aborts, the subtransaction is undone.

InterSQL provides transaction specification facilities to allow the user to define Flex tasks and appropriate commitment methods. An InterSQL program consists of the following fundamental components: objects and types, subtransactions definitions, dependency description among subtransactions, preference descriptions and a reservation list. *Objects* serve as results of, and as arguments to, subtransactions. Objects are categorized by *types* and are capable of participating in a specific set of subtransactions. A *subtransaction definition* specifies the name of the subtransaction, the system at which it executes, its arguments, the commands to be executed and the commitment methods that can be used to commit it. In addition, the definition of a subtransaction may include *guards* (described below) and time options such as the valid time period of its execution and its maximum execution time. When the dependency conditions and the time constraints for a subtransaction are satisfied, its guard is evaluated and if it holds then the execution of the transaction is granted, otherwise the execution is delayed until after another evaluation of the guard. The *dependency description* part of the InterSQL program allows the user to define the execution dependencies between subtransactions. When the dependency conditions and the time constraints for a subtransaction are satisfied, its guard is evaluated and if it holds then the execution of the transaction is granted, otherwise the execution is delayed until after another evaluation of the guard. The *preference description* allows the user to specify the conditions under which a subtransaction is preferred over another. Finally, the *reservation list* represents explicit reservation actions to be taken in an attempt to ensure that the subtransaction can be committed.

Important Features:

- the use of an extended transaction model
- the use of a Transaction Specification Language to support the extended transaction model
- the treatment of commitment

5.11 The FIB Project

The Federated Information Bases (FIB) project at Georgia Tech [NSA+94] focus mainly on the semantic interoperability problems encountered in multidatabase systems.

System Architecture: The architecture of FIB is shown in Figure 10 adopted from [NSA+94]. The flow of control between the components is explained below in the *Query Processing* Section.

Common Data Model: FIB's CDM, called CANDIDE, is a terminological knowledge representation model that supports classes, attributes (instance variables), instances, and disjoint classes (classes whose subclasses cannot have any common instances). So far it does not provide methods or any other form of behavioral support. The database schema consists of two partially ordered lattices, one for the class taxonomy and one for the attribute hierarchy. In the attribute hierarchy, each attribute can have at most one parent attribute along with an associated domain. This domain is either an instance, or a set of instances of a class described in the class taxonomy. The domain of an attribute must be a subclass of the domain of its parent attribute. An attribute appearing in a class definition can be qualified by additional value constraints on its domain. These constraints must logically imply the constraints on each attribute of its superclasses.

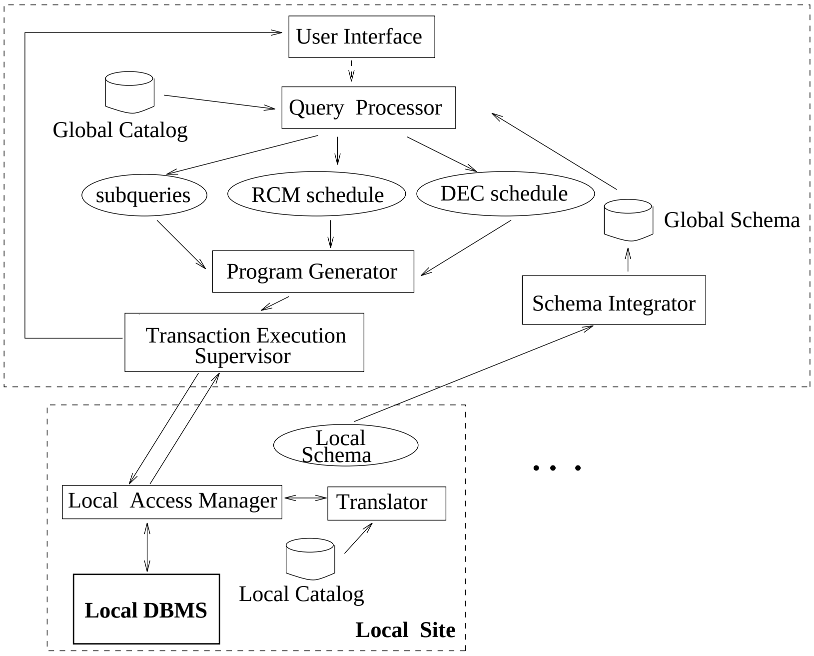

Figure 10: FIB System Architecture

The same language is used both as a DDL and a DML. Querying is based on the notions of subsumption and classification. A class A *subsumes* a class B if and only if every instance of B is also an instance of A, i.e., A is a superclass of B. The subsumption relationship is computed on the basis of whether the attribute constraints for class A logically imply the attribute constraints for class B. *Classification* is a search technique which correctly places new classes into an existing lattice. The correct location for a class A is immediately below the most specific classes which subsume A and immediately above the most general classes subsumed by A. Querying by classification is the process of specifying a query object and then searching for objects which are structurally related to this object using classification.

Translation: The mapping of a relational schema into the CANDIDE data model is described in [NSA+94]. Each relation is mapped to a class, while tuples are mapped to instances of a class. The key of a relation is associated with the class name. The values within an attribute domain become instances of a class representing the attribute.

Integration: The schema integration process is divided into two distinct phases [SGN93]. In the first phase, the user gives as input to the integration module a set of attribute relationships. Two attributes a_1 and a_2 can be related as follows: a_1 *is-equivalent-to* a_2, a_1 *is-above* or *is-below* a_2 in the attribute hierarchy, or they may be unrelated. In the second phase, the classification and the subsumption techniques are employed to deduce the class relationships automatically. The relationships that are identified between two classes are: *subsume*, *equivalent* (each class subsumes the other), *disjoint* (the classes have no common instances), *overlap* (none of the classes subsumes the other and are not disjoint), and unrelated. Schema merging operators are automatically applied to pairs of classes according to the nature of relationships between them to generate the global schema. Any changes in the underlying component schemas require only re-classification of the global schema along with any new attribute correspondence. A schema integration tool based on this approach is described in [SSG+91].

Query Processing: The flow of control among the FIB's components is shown in Figure 10. The user interface allows a user to browse the global schema and formulate queries. The query processor is responsible for accepting the user queries and generating the subqueries with the help of a global catalog. The RCM scheduler detects any dependencies and parallelism and generates a schedule for the subqueries. The results of the subqueries must be combined to generate the final results. The result combination is performed by one of the component databases. The DEC scheduler generates a schedule for the result combination operations. The program generator generates the program to be executed by the transaction execution supervisor. The execution order of subqueries and the sequencing information is encoded in the generated program using constructs adapted from DOL [ELLR90] (a transaction specification language which is a predecessor of IPL described in the previous section).

The execution supervisor coordinates the execution of the subqueries as specified by the DOL program. First it set-ups connections with the Local Access Managers (LAM). Next, it sends the subrequests to the specified LAMs in the specified order. The subrequests are translated into the local model using information from a local catalog. The subqueries are executed by the local database system and the results are translated back to the CDM. The result combination is performed at a suitable local site. The LAM at this site executes the result combination as it would execute a subquery. First, it waits for the partial results from subqueries at other local sites. Then it creates temporary tables for these results and executes result combination operations. The final result is sent to the user interface.

Important Features:

- the use of classification to perform query processing and schema integration
- automation of the schema integration process

5.12 Conclusions

Tables 2, 3, 4 and 5 present a comparative analysis of the systems. In Table 2 we characterize as complete systems, systems that, in addition to providing an integration framework and a transaction model, support network communication and various operating system facilities. Thor is different from the other systems described in that it does not support the integration of pre-existing systems.

System Architecture

DOMS, EIS, and OIS/CIS support an object-based architecture. DOMS in particular, being the most recent of them, provides the functionality and adapts the terminology of most of the proposed architectural standards. In an object-based architecture, all resources are modeled as objects and all provided services are modeled as object methods. Object managers handle objects and the communication between them.

Translation and Integration

From the systems described, OIS, CIS, FBASE and InterBase* can be characterized as non-federated since they do not support the creation of a global schema. All other systems fall in the category of federated databases. OIS and CIS propose an Operational Mapping Approach. Following this approach, each local database must provide a minimum interface, in the form of an implementation for a predefined set of generic operations. These operations are basic operations such as operations for accessing the instance variables (components) of an object or operations for creating new objects. More complex operations are built on top of the generic operations.

All federated systems define the global schema by using the view definition facilities of their query language. FIB bases integration on classification, a technique that explores the structure of a class to automatically place it in a given class taxonomy by applying appropriate merging constructors. The ViewSystem provides the most comprehensive set of class constructors. EIS and DOMS define an object algebra for their model and pose the question of optimality for the set of class constructors in terms of query optimization and expressive power. An interesting issue is how virtual classes share the functionality of their base classes. Inheritance is the most popular method, but the classical definition of inheritance from a subclass to a superclass must be formally generalized to provide for classes constructed by methods other than subclassing. EIS introduce the use of delegation as a means for information sharing. The usefulness of all approaches to sharing needs to be evaluated by performance studies. Other interesting issues in object-oriented integration include method resolution and the treatment of identifiers for imaginary objects.

Transaction Management

Most systems that discuss transaction management (DOMS, Carnot, EIS, InterBase*) support extended transaction models. The general trend is for "customized" or "programmable" transaction management. According to this approach, the user will specify the criterion of correctness in terms of desired relationships between different subtransactions. DOMS and Carnot propose ACTA and DOL based transaction specification languages respectively. This work is based on earlier work on extended transaction models done in InterBase [ELLR90] and Omnibase [RS91]. In addition, InterBase* provides an elaborate treatment of the commitment problem.

System	Type	Integrated Systems
Pegasus [ADK+91, ADD+91, AAD+93]	Complete Data Management System	Information Systems of various data models
ViewSystem [KDN91]	Tool, Environment that supports integration	Information Bases
CIS/OIS [GCO90, BNPS89, BNPS88]	Integration Tool	file systems, Information Retrieval Systems, Databanks
OMS [HZ90, HHZ+92]	Framework	Engineering Information Systems
DOMS [MHG+92, BOH+92]	Complete System	not just database systems but also hypermedia applications conventional applications, etc
UniSQL/M [KCGS93, Kim92]	Multidatabase System	SQL-based relational databases and the UniSQL/X database system
Carnot [WCH+93, WSHC92]	Complete System	database systems, knowledge-base systems and process models
Thor [LDS92]	Distributed DBMS that provides sharing of objects among heterogeneous distributed systems	not applicable
FBASE [Mul92]	Integration Framework and prototype system	database systems
InterBase* [ME93, BCD+93]	Complete System	database systems and Unix utilities
FIB [NSA+94, SGN93]	Multidatabase System	database systems

Table 2: Heterogeneous Systems

System	Data Model	DD/DM Language	Translation
Pegasus	Iris data model	HOSQL Extension of SQL Query based (*)	During importation Supports automatic translation of relational models
ViewSystem	VODAK data model	DML Programming-based and set-oriented	During importation Resources mapped to methods Special Smalltalk library routines (classes) support translation of most common information sources
CIS/OIS	Abstract data model (OIS) Integration data model (CIS)	QL Extension of a logic-based query language Query-based	Operational mapping
OMS	FUGUE model	Extension of a functional-based query language Set-oriented	Not discussed
DOMS	FROOM	Extension of a functional-based query language Set-oriented	Not discussed
UniSQL/M	Unified relational and object-oriented model	SQL/M Query-based	Not necessary, CDM is a superset of the relational model
Carnot	Instead of a CDM, uses a common-sense knowledge base, called Cyc	GCL Global context language Based on extened first-order logic	Special frames defined for common information sources
Thor	Based on Argus	Based on Argus Programming-based	Not applicable
FBASE	Object-oriented Defines a class hierarchy to model the integrated systems	FSQL Extension of SQL Query-based	Performed by special FBASE servers
InterBase*	Object-oriented	InterSQL Based on FSQL Provides transaction specification facilities	Performed by special servers, called CSIs
FIB	CANDIDE Terminological knowledge-based Empasis on structure rather than on behavior	Based on classification	Performed at runtime by special translation modules

(*) the characterization of languages is based on definitions given in Section 2.2

Table 3: Data Model and Translation

System	Integration (*)		
	Importation	Derived Classes	Conflicts
Pegasus	By queries Virtual classes called producer types and the query that defines them, producer expression	By queries Virtual classes called unifying types Function inherited from base classes by unifying inheritance	Domain mismatch (semantic) Naming and schema mismatch (schema) Object identification (identity)
ViewSystem	Maps external information sources to methods Virtual classes called external classes	By applying constructors Constructors supported: specialization, generalization, grouping, aggregation Virtual classes called derived classes	Not discussed
CIS/OIS	Not supported		
OMS	By queries and functions (constructors) Virtual classes called derived classes Object-algebra defined with a set of functions that produce new sets of objects from existing ones		Not discussed
DOMS	By queries and functions (constructors) Object-algebra defined with a set of functions that produce new sets of objects from existing ones		Not discussed
UniSQL/M	By queries		Comprehensive treatment
Carnot	Uses articulation axioms to express mappings between two expressions that have equivalent meaning		Not discussed
Thor	Not applicable Provides for information sharing among heterogeneous sytems through a universe of objects		
FBASE	Not supported		
InterBase*	Not supported		
FIB	Relationships between the base classes induced by a classification method Class constructors are then applied automatically		Not discussed

(*) the description follows the methodology introduced in Section 2.4

Table 4: Integration

System	Transaction Management
Pegasus	Not supported
ViewSystem	Not supported
CIS/OIS	Not supported
OMS	Nested (cooperative) transactions Goal: Customize transaction management No formal definition of correctness
DOMS	Goal: Programmable transaction management Two types of transactions : (i) multisystem transactions (extended transactions defined using ECA rules) (ii) multidatabase transactions (traditional heterogeneous transactions)
UniSQL/M	Supports concurrency control Assumes a prepare-to-commit state
Carnot	Goal: User-specified correctness Provides a language based on ACTA for defining relationships between subtransactions
Thor	Not a multidatabase, but a (homogeneous) distributed DBMS Basis of concurrency control are objects Commitment protocol : 2PC Optimistic concurrency control algorithm Primary copy schema for replication
FBASE	Not supported
InterBase*	Supports the Flex extended transaction model Provides transaction specification language for defining the model Provides an elaborate treatment of commitment at the subtransaction level
FIB	Not supported Constructs based on the DOL transaction specification language may be used to specify dependencies among subqueries

Table 5: Query and Transaction Processing

6 Summary

Using object-oriented techniques to build heterogeneous databases is a promising approach. Objects provide a natural model of a heterogeneous environment. Modeling resources as objects and their services as methods hides the heterogeneity of their implementation and respects their autonomy. At a lower level, providing an object-oriented model for the data in the heterogeneous database facilitates the expression of relations and the resolution of conflicts that exist between entities at different component database systems. Finally, object technology offers an efficient method of modeling and implementing heterogeneous transaction management and of supporting the use of semantic information to allow more concurrency.

Unfortunately, the abundance of models and techniques makes the study and evaluation of object-oriented approaches intricately difficult. In this paper we have presented a unifying analysis of the process of building object-oriented heterogeneous database systems. Various methods have been examined and a number of real-life systems have been comparatively studied. We believe that this comprehensive review will enhance our understandability of the issues, substantiate the use of object-oriented techniques and help put into perspective existing and future projects.

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.

[AAK+93] Joseph Albert, Rafi Ahmed, Mohammed Ketabchi, William Kent, and Ming-Chien Shan. Automatic importation of relational schemas in pegasus. In *Proceedings of the RIDE-IMS*, pages 105–113, April 1993.

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

[ADD+91] Rafi Ahmed, Philippe DeSchedt, Weimin Du, William Kent, Mohammed Ketabchi, Witold Litwin, Abbas Rafii, and Ming-Chien Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, 24(12):19–27, December 1991.

[ADK+91] Rafi Ahmed, Philippe DeSchedt, William Kent, Mohammed Ketabchi, Witold Litwin, Abbas Rafii, and Ming-Chien Shan. Pegasus: A System for Seamless Integration of Heterogeneous Information Sources. In *COMPCON 91*, pages 128–136, March 1991.

[B+89] Robert Bretl et al. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. ACM Press, 1989.

[BBG89] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of The Association for Computing Machinery*, 36(2):230–269, April 1989.

[BCD+93] Omran A. Bukhres, Jiansan Chen, Weimin Du, Ahmed K. Elmagarmid, and Robert Pezzoli. InterBase: An Execution Environment for Heterogeneous Software Systems. *IEEE Computer*, pages 57–69, August 1993.

[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.

[BEM92] Omran A. Bukhres, Ahmed K. Elmagarmid, and James G. Mullen. Object-Oriented Multidatabases: Systems and Research Overview. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 27–34, Baltimore, Maryland, USA, November 1992.

[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.

[BGM89] Gordon S. Blair, John J. Gallagher, and Javad Malik. Genericity vs Inheritance vs Delegation vs Conformance vs ... *JOOP*, pages 11–17, September/October 1989.

[BGMS92] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1(2):181–239, 1992.

[BGS91] Yuri Breitbart, Dimitrios Georgakopoulos, and Abraham Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transaction on Software Engineering*, 17(9):954–960, September 1991.

[BHG87] Philip A. Bernstein, Vassos Hadjilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addisson-Wesley, 1987.

[BHP92] M.W. Bright, R. Hurson, and Simin H. Pakzard. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, pages 50–60, March 1992.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

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

[BNPS88] E. Bertino, M. Negri, G. Pelaggati, and L. Sbatella. The comandos integration system: an object-oriented approach to the interconnection of heterogeneous applications. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 213–218, September 1988.

[BNPS89] E. Bertino, M. Negri, G. Pelaggati, and L. Sbatella. Integration of Heterogeneous Database Applications through an Object-Oriented Interface. *Information Systems*, 14(5):407–420, 1989.

[BOH+92] Alejandro Buchmann, M. Tamer Ozsu, Mark Hornick, Dimitrios Georgakopoulos, and Frank A. Manola. A Transaction Model for Active Distributed Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan Kaufmann, 1992.

[BR88] B. R. Badrinath and Krithi Ramamritham. Synchronizing Transactions on Objects. *IEEE Transactions on Computers*, 37(5):541–547, May 1988.

[Cat93] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[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.

[CHS91] Christine Collet, Michael N. Huhns, and Wei-Min Shen. Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, 24(12):55–62, December 1991.

[CL88] Tim Connors and Peter Lyngbaek. Providing Uniform Access to Heterogeneous Information Bases. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 162–173, September 1988.

[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.

[CS91] Malu Castellanos and Felix Saltor. Semantic Enrichment of Database Schemas: An Object Oriented Approach. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 71–78, April 1991.

[CT91] Bogdan Czedjo and Malcolm Taylor. Integration of Database Systems Using an Object-Oriented Approach. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 30–37, April 1991.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In R. Hull, Morrison R, and D. Stemple, editors, *Database Programming Lnaguages, Proceedings of the 2nd International Workshop*. Morgan Kaufmann, 1989.

[DBM88] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are Objects too: A Knowledge Model for an Active, Object-O riented database system. In J.W. Schmidt, S. Ceri, and M.Missikoff, editors, *Advances in Database Technology - EDBT '88*, pages 127–143. Springer Verlag, 1988.

[DE89] W. Du and A. K. Elmagarmid. Quasi Serializability: A correctness criterion for global concurrency correctness in Interbase. In *Proceedings of the International Conference on Very Large Databases*, Amsterdam, 1989.

[DEL+82] C. Devor, R. Elmasri, J. Larson, S. Rahimi, and J. Richardson. Five-schema Architecture Extends DBMS to Distributed Applications. *Electronic Design*, pages 27–32, March 18 1982.

[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.

[DKT88] Horst Duchene, Manfred Kaul, and Volker Turau. VODAK Kernel Data Model. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 174–192, September 1988.

[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.

[EP990] Special Issue on Heterogenous Databases, A. Elmagarmid and C. Pu, eds. *ACM Computing Surveys*, 22(3), September 1990.

[FHM92] Doug Fang, Joachim Hammer, and Dennis McLeod. An Approach to Behavior Sharing in Federated Database Systems. In *Proceedings of the International Workshop on Distributed Object Management*, pages 66–80, Edmonton, Canada, August 1992.

[Gal92] Leonard J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proceedings of the 1st International Conference on Information and Knowledge Management*, 1992.

[GCO90] R. Gagliardi, M. Caneve, and G. Oldano. An Operational Approach to the Integration of Distributed Heterogeneous Environments. In *Proceesings of the PARBASE-90 Conference*, pages 368–377, Miami Beach, Florida, March 1990.

[GHK93] Dimitrios Georgakopoulos, Mark Hornick, and Piotr Krychniak. An Environment for the Specification and Management of Extended Transactions in DOMS. In *Proceedings of the RIDE-IMS*, pages 253–257, April 1993.

[GHKM94] Dimitrios Georgakopoulos, Mark Hornick, Piotr Krychniak, and Frank Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proceedings of the 10th International Conference on Data Engineering*, February 1994.

[GPN91] J. Geller, Y. Perl, and E. J. Neuhold. Structure and semantics in OODM Class Specification. *SIGMOD Record*, 20(4):40–43, December 1991.

[GPNS92] James Geller, Yehoshua Perl, Erich Neuhold, and Amit Sheth. Structural Schema Integration with Full and Partial Correspondence using the Dual Model. *Information Systems*, 17(6):443–464, 1992.

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

[Gro92] Object Management Group. Object Management Architectue Guide. OMG Document Number 92.11.1, September 1992.

[GRS91] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In *Proceedings of the 7th International Conference on Data Engineering*, pages 314–323, Kobe, Japan, April 1991.

[GSCS93] Manuel Garcia-Solaco, Malu Castellanos, and Felix Saltor. Discovering Interdatabase Resemblance of Classes for Interoperable Databases. In *Proceedings of the 2nd International Workshop on Interoperability in Multidatabase Systems*, pages 26–33, 1993.

[Gul86] Agha Gul. *Actors*. The MIT Press, 1986.

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

[HHZ+92] Sandra Heiler, Sara Haradhvala, Stanley Zdonik, Barbara Blaustein, and Arnon Rosenthal. A Flexible Framework for Transaction Management in Engineering Environment. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 88–121. Morgan Kaufmann, 1992.

[HJK+92] Michael N. Huhns, Nigel Jacobs, Tomas Ksiezyk, Wei-Min Shen, Munindar P. Singh, and Phillip E. Cannata. Enterprise Information Modeling and Model Integration in Carnot. In Jr Charles J. Petrie, editor, *Enterprice Integration Modeling, Proceedings of the First International Conference*, pages 290–299. The MIT Press, 1992.

[HW91] Maurice P. Herlihy and William E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43:25–61, 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.

[KCGS93] Won Kim, Injun Choi, Sunit Gala, and Mark Scheevel. On Resolving Schematic Heterogeneity in Multidatabase Systems. *International Journal of Parallel and Distributed Databases*, 1:251–279, 1993.

[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.

[Ken93] William Kent. The objects are coming! *Computer Standards and Interfaces*, 15, 1993.

[KFM+95] Wolfgang Klas, Peter Fankhauser, Peter Muth, Thomas Rakow, and E. J. Neuhold. Database Integration using the Open Object-Oriented Database System VODAK. In Ahmed Elmagarmid and Omran Bukhres, editors, *Object-Oriented Multidatabases*. Prentice Hall, 1995. to appear.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[Kim92] W. Kim. The UniSQL/M System. Personal Communication, September 1992.

[KKS92] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying Object-Oriented Databases. In *Proceedings of the 1992 ACM Sigmod Conference*, pages 392–402, 1992.

[KLK91] Ravi Krishnamurthy, Witold Litwin, and Wiliam Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. In *Proceedings of the ACM SIGMOD*, pages 40–49, 1991.

[KS91] Won Kim and Jungyun Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems'. *IEEE Computer*, 24(12):12–17, December 1991.

[Kul93] Krishna G. Kulkarni. Object orientation and the sql standard. *Computer Standards and Interfaces*, 15:287–301, 1993.

[Kul94] Krishna G. Kulkarni. Object-Oriented Extensions in SQL3: A Status Report. In *Proceedings of the 1994 ACM Sigmod Conference*, page 478, May 1994.

[LDS92] Barbara Liskov, Mark Day, and Liuba Shira. Distributed Object Management in Thor. In *Proceedings of the International Workshop on Distributed Object Management*, pages 1–15, Edmonton, Canada, August 1992.

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

[Lis88] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[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.

[LMR90] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LNE89] J. Larson, S. Navathe, and R. Elmarsi. A Theory of Attribute Equivalence in Databases with Applications to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, April 1989.

[ME93] James G. Mullen and Ahmed Elmagarmid. InterSQL: A Multidatabase Transaction Programming Language. In *Proceedings of the 1993 Workshop on Database Programming Languages*, 1993.

[MH92] Frank Manola and Sandra Heiler. An Approach to Interoperable Object Models. In *Proceedings of the International Workshop on Distributed Object Management*, pages 326–330, Edmonton, Canada, August 1992.

[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.

[MKSA92] James G. Mullen, Won Kim, and Jamshid Sharif-Askary. On the Impossibility of Atomic Commitment in Multidatabase Systems. In *Proceedings of the 2nd International Conference on System Integration*, Morristown, New Jersey, 1992.

[MNE88] Michael V. Mannino, Shamkant Navathe, and Wolgang Effelsberg. A Rule-Based Approach for Merging Generalization Hierarchies. *Information Systems*, 13(3):257–272, 1988.

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

[Mul92] James G. Mullen. FBASE: A Federated Objectbase System. *International Journal of Computer Systems Science and Engineering*, 7(2):91–99, April 1992.

[NSA+94] Shamkant Navathe, Ashoka Savasere, Tarek Anwar, Howard Beck, and Sumit Gala. Object Modeling Using Classification in CANDIDE and its Application. In A. Dogac, M. T. Ozsu, A. Biliris, and T. Selis, editors, *Advances in Object-Oriented Database Systems*. Springer Verlag, 1994.

[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.

[OV91] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1991.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

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

[Pit95] Evaggelia Pitoura. Extending an Object-Oriented Programming Language to Support the Integartion of Database Systems. In *28th Annual Hawaii International Conference on System Sciences (HICSS-28)*, pages 707–716, Maui, Hawaii, Jan 1995.

[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.

[PSH91] Girish Pathak, Bill Stackhouse, and Sandra Heiler. EIS/XAIT Project: An Object-Based Interoperability Framework for Heterogeneous Systems. *Computer Standards and Interfaces*, 13:315–319, 1991.

[RC92] Krithi Ramamritham and Panos K. Chrysantis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In *Proceedings of the International Workshop on Distributed Object Management*, pages 120–140, Edmonton, Canada, August 1992.

[RS91] Marek Rusinkiewicz and Amit Sheth. Multitransaction for Managing Inderdependent Data. *IEEE Data Engineering Bulletin*, 14(1), March 1991.

[RTL+91] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software Practice and Experience*, 21(1), January 1991.

[SBCE93] Tony Schaller, Omran A. Bukhres, Jiansan Chen, and Ahmed K. Elmagarmid. A Taxonomic and Analytical Survey of Multidatabase Systems. Technical Report CSD-TR-93-040, Purdue University, 1993.

[SCGS91] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. *SIGMOD Record*, 20(4):44–48, 1991.

[SGN93] A. P. Sheth, S. K. Gala, and S. B. Navathe. On Automatic Reasoning for Schema Integration. *International Journal of Intelligent and Cooperative Information Systems*, 2(1), March 1993.

[Ska91] Andrea H. Skarra. Localized Correctness Specification for Cooperating Transactions in an Object-Oriented Database. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):79–106, 1991.

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

[SLCN88] Amit P. Sheth, James A. Larson, Aloysius Cornelio, and Shamkant B. Navathe. A Tool for Integarting Conceptual Schemas and User Views. In *Proceedings of the 4th International Conference on Data Engineering*, pages 176–183, February 1988.

[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.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of OOPSLA '86*, pages 38–45, September 1986.

[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.

[SS84] Peter M. Schwartz and Alfred Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[SS90] Marc H. Scholl and Hans-Jorg Schek. A Relational Object Model. In *Proceedings of the International Conference on Database Theory - ICDT '90*, pages 89–105, December 1990.

[SSG+91] Ashoka Savasere, Amit Sheth, Gunit Gala, Shamkant Navathe, and Howard Markus. On Applying Classification to Schema Integration. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 258–261, April 1991.

[SST92] Marc H. Scholl, Hans-J Schek, and Markus Tresch. Object Algebra and Views for Multi-Objectbases. In *Proceedings of the International Workshop on Distributed Object Management*, pages 336–359, Edmonton, Canada, August 1992.

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

[SZ89] Andrea H. Skarra and Stanley Zdonik. Concurrency Control and Object-Oriented Databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 359–421. ACM Press, 1989.

[Tay92] C.J. Taylor. A Status Report on Open Distributed Processing. *First Class (Object Management Group Newsletter)*, 2(2):11–13, June/July 1992.

[TK78] D. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework. *Information Systems*, 3(4), 1978.

[TLM+92] Christine Tomlinson, Greg Lavender, Greg Meredith, Darrell Woelk, and Phill Cannata. The Carnot Extensible Service Switch (EES) - Support for Service Execution. In Jr Charles J. Petrie, editor, *Enterprice Integration Modeling, Proceedings of the First International Conference*, pages 493–502. The MIT Press, 1992.

[TTC+90] Gomer Thomas, Glenn R. Thompson, Chin-Wan Chung, Edward Barkmeyer, Fred Carter, Marjorie Templeton, Stephen Fox, and Berl Hartman. Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22(3):237–265, September 1990.

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

[WCH+93] Darrell Woelk, Phill Cannata, Michael Huhns, Wei-Min Shen, and Christine Tomlinson. Using Carnot for Enterprise Information Integration. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 133–136, January 1993.

[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.

[Wei89] William E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[Wei91] Gerhard Weikum. Principles and Realization of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WSHC92] Darrell Woelk, Wei-Min Shen, Michael Huhns, and Phill Cannata. Model Driven Enterprise Information in Carnot. In Jr Charles J. Petrie, editor, *Enterprice Integration Modeling, Proceedings of the First International Conference*, pages 301–309. The MIT Press, 1992.

[ZE93] Aidong Zhang and Ahmed K. Elmagarmid. A Theory of Global Concurrency Control in Multidatabase Systems. *VLDB Journal*, July 1993.

[ZP93] Aidong Zhang and Evaggelia Pitoura. A View-Based Approach to Relaxing Global Serializability in Multidatabase Systems. Technical Report CSD-TR-93-082, Purdue University, 1993.