Dynamic Evolution of Integrated Schemas for Federated Objectbase Systems

by

Albert A. Allwyn D'Silva

A thesis

submitted to the faculty of graduate studies

in partial fulfillment of the requirements

for the degree of

Master of Science

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

DYNAMIC EVOLUTION OF INTEGRATED SCHEMAS FOR FEDERATED OBJECTBASE SYSTEMS

BY

ALBERT A. ALLWYN D'SILVA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Schema integration has been a topic of active research for several years and many methodologies have been presented. Most integration methodologies assume that the local databases participating in the federation do not change, and hence these methodologies ignore the post-integration phase. Typically, local schemas are unified to form one or more integrated schemas that do not change over time. However, changes made to the local schemas should be propagated to the integrated schemas so that the integrated schemas remain coherent. Schema evolution and integration are closely related and their association is evident in the post-integration phase of schema integration.

In this thesis, the local schemas considered for integration are dynamic in nature and integration is performed in a federated environment. The proposed methodology uses an axiomatic model of dynamic schema evolution to represent the local schemas. The axiomatic model serves as a common foundation for evolving schemas when several systems with different object models participate in a federation. Other work has shown that the popular Orion object model and the purely behavioral object model of Tigukat can be represented in the axiomatic model. Two additional object models, the GemStone and the O_2 models, are axiomatized to illustrate the diversity of the object models represented in the axiomatic form. A new method of generating the initial integrated schema is presented, wherein related types form a subtype hierarchy that describes a similar real world concept. The axiomatic model automatically supports the process of forming subtype hierarchies. The essential schema evolution operations that effect the integrated schema are presented in terms of the axiomatic model. Changes made to the local schemas are propagated to the integrated schema without the need to carry out the re-integration process repeatedly and in its entirety.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor and *guru*, Dr. Randal Peters for all his help and guidance. I greatly appreciate his total dedication and support during the course of my master's program.

I would like to thank the members of my committee Dr. Ken Barker and Dr. Bob McLeod for their comments and suggestions. I would also like to thank Dr. Kasi Periyasamy for his insights, ideas, and encouraging words.

I thank my parents, brother, and sisters for their love and untiring support throughout my life. My thanks go to my friends for all the wonderful times. They made my stay in Winnipeg a memorable one.

And finally, I thank God for all the opportunities in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need to share and exchange information among various organizations is the key factor that attributes to the importance of database interoperation. Today's applications call for information that span beyond the context of a single organization. For example, consider an expanding organization with operating locations spread out over a large geographical area. Each location may have its own database of its local employees and operations. To successfully manage such huge organizations, it is important that these operating locations share and exchange information among themselves. Sometimes, sharing of information among distinct, but related, organizations is vital to providing better service to their clientele. For example, travel agencies share part of their data with other agencies (possibly in a different geographical location) to meet customer requirements. Interoperation among such pre-existing, heterogeneous, and autonomous databases has been a topic of on-going research for the past several years. To overcome the problems of heterogeneity and autonomy, applications need access to the distributed databases in a homogenized fashion, allowing meaningful access and exchange of their data [11]. Many approaches of achieving *interoperability* have been proposed, and two of the most common techniques are the global schema approach of *multi-database systems* (MDBS) [24] and the federated schema approach of *federated database systems* (FDBS) [20].

The above represents an enterprise approach to interoperability. Another common approach to interoperability is through middleware services, such as those provided by CORBA [45]. In simple terms, CORBA is a standard for distributed objects. It provides a flexible communication and activation substrate for distributed heterogeneous object sources through Object Request Brokers (ORBs). In the remainder of the thesis we assume an enterprise approach to interoperability.

In the global schema approach, all databases are fully integrated to provide a single global conceptual schema. This global schema is sometimes called a *corporate* or *enterprise* schema. Distributed DBMSs fall under the global schema approach. The disadvantage is that it has only one level of management and all operations are performed uniformly. Moreover, a distrubuted DBMS does not distinguish between local and non-local users, and offers little or no autonomy to the participating databases. In the above travel agency example, it would be unwise to integrate all the databases completely, because the individual agencies would lose many administrative privileges.

The work by Sheth and Larson [38] uses the term multidatabase systems in the same context as distributed DBMSs is used above. Özsu and Valduriez [24] describe a MDBMS as consisting of a single conceptual schema. However, it has two levels of administration - the global schema level and the local schema level. Also, they state that MDBMSs provide full autonomy. The disadvantage of such a system is that there can be at most one global conceptual schema.

In the federated schema approach, while the component databases remain semi-autonomous, they allow partial and controlled sharing of their data. This approach is more desirable because the component databases have control over what is shared and what is not by providing only a part of the schema known as the *export* schema. Local database administrators can use these export schemas to define an *import* schema, representing remote information that is locally accessible. Again, referring back to the travel

agency example, each travel agent may make available a part of its database that the other agencies can import and use locally.

At the core of the interoperability problem is *schema integration*, which is the process of integrating multiple schemas into one or more integrated schemas. The choice of a *canonical* or *common* data model is vital to achieve good integration. All export schemas of the component databases are typically transformed into a common data model. The common data model should be simple and expressive enough to represent and compare various entities from different export schemas. Some of the more popular data models used include: relational model, entity-relationship (ER) model, and object-oriented (OO) models. The relational model has been thoroughly studied in the literature. However, this model lacks semantic expressibility and is often substituted with high-level semantic models such as the ER model and the OO models.

Since the early 1980's, there have been numerous research projects carried out in integrating databases. Most of the early work focuses on integrating relational databases. Batini et. al. [3] present a comprehensive survey of twelve early view and schema integration methodologies. They also propose some general guidelines for schema integration and these guidelines serve as a basis for most of the recent integration methodologies. The steps of these guidelines are as follows:

1. Pre-integration,
2. Comparison of the Schemas,
3. Conforming the Schemas, and
4. Merging to generate an Integrated Schema.

Details of the process involved with these guidelines appears in Section 2.3.

Schema evolution logically follows with integration, but few researchers have addressed this problem in the context of integrated schemas and it has been largely ignored in the lit-

erature. The need to support evolution after integration is overwhelming because, typically, user requirements change over time and these changes will reflect on the export schemas, and hence the integrated schema as well. Of course, after evolving a local schema, it is possible to re-generate export schemas and perform integration again, but this approach should be avoided because of the significant amount of re-integration necessary. In order to meet the *evolving* requirements of the federation users, schema changes should be faithfully propagated to the integrated schema without the need to re-integrate the entire schema.

This thesis addresses the problem of schema integration in a federated environment with an emphasis on the evolution of the integrated schema(s). A restructuring approach, rather than complete re-integration, is taken to handle evolution of the integrated schema(s). The thesis presents a formal model to support evolution at the integrated schema level based on an axiomatic model of schema evolution and the axiomatization of the export schemas.

1.1 Scope and Contributions

The following outlines the scope and contributions of the thesis:

Architecture: The work presented in this thesis follows the *tightly coupled federated database* approach. However, it can be extended to support integration of the *loosely coupled FDBS* as well. This classification is based on who manages the federated schema. In this thesis, it is assumed that a federation administrator is present and provides the inter-schema correspondence assertions.

Object Model: An axiomatic model of dynamic schema evolution [27] serves as the common data model. Two object models, GemStone and O_2, are reduced to the axiomatic model thereby making the task of comparing the schemas easier. This process is identified as stage 2 of the general guidelines of schema integration listed in the previous section. Other object models, Tigukat and Orion, have also been expressed in terms

of the axiomatic model [27]. This provides evidence that a variety of object models can be expressed in the axiomatic model and hence it serves as a good basis for integration.

Schema Integration Methodology: An integration methodology based on the axiomatic model is developed. This methodology utilizes existing approaches to carry the axiomatized export schemas to an axiomatized integrated schema. Equivalent types are identified using real-world semantics (RWS) [36] and type structure. Equivalence classes are formed by using the RWS correspondence assertions among types in different axiomatized export schemas. With the help of the axiomatic model, a type hierarchy is formed in each equivalence class. Such a type hierarchy represents a single domain in the real world, with *similar* real world objects. This method of forming a type hierarchy of related concepts is a new approach. Initial integration is done by integrating all the type hierarchies. Again, the axiomatic model automatically supports this process based on the subtyping relationships.

Integrated schema restructuring: The integration methodology supports the propagation of schema updates to the integrated schema without the need to re-integrate all the export schemas. The most common schema evolution operations that need to be propagated to the axiomatized federated schema are presented. These schema update operations are expressed in terms of the axiomatic model. All the update operations are followed by an illustrative example.

There are two assumptions made in this thesis: (i) all the export schemas are object-oriented (this simplifies the task of reducing them to the axiomatic model) and (ii) integration is done at the conceptual level. Conceptual level modeling focuses on capturing the application requirements and representing the applications in a natural and straightforward way without considering the implementation level details.

Furthermore, instance adaption to an evolved schema is not part of this thesis. Instance adaption is the process of coercing objects of the component schemas to coincide with the integrated schema. However, the handling of instance adaption is part of the superset problem of schema integration, known as database integration.

1.2 Organization of the thesis

The remainder of the thesis is organized as follows: In Chapter 2, a review of related work in schema integration and schema evolution is presented. These works serve as the motivating technologies of the thesis. In Chapter 3, an overview of the axiomatic model of dynamic schema evolution is presented. GemStone and O_2 object models are expressed in terms of the axiomatic model and will serve as a common model for integration. Other models such as Orion and Tigukat have been expressed in the axiomatic model by others [27]. In Chapter 4, inter-schema relationships are identified from the common axiomatic model. Conflict resolution and initial schema integration is covered in this chapter. In Chapter 5, the significance of dynamically restructuring the integrated schema is discussed. Various changes in the export schema that affect the integrated schema are identified and dealt with. The update operations are followed by an illustrative example. Chapter 6 contains concluding observations and the potentials for future work.

Chapter 2

Motivating Technologies

One of the fundamental objectives among the database research community is to support effective sharing and exchange of information among various database systems. This is known as database interoperability and the long term goal of this research is to develop architectures for interoperation of databases that span beyond the context of a single organization. Since these database systems are typically created independent of each other, they may differ in their *database management system* (DBMS), their underlying *data model* or their *conceptual schema*. DBMS can be characterized along three orthogonal dimensions: *distribution*, *heterogeneity*, and *autonomy* as shown in Figure 2.1 taken from [24].

The shaded portion of Figure 2.1 represents the DBMSs that require autonomy. These include the federated and multidatabase systems. In contrast, a distributed DBMS has no autonomy. The term multidatabase has conflicting meanings in the literature. In [24], a multidatabase system is defined as an interconnected collection of autonomous databases. However, this definition contrasts with the one in [38] where the definition of multidatabase system is more generic. According to the latter definition, the multidatabase systems encompasses the entire Figure 2.1. The scope of this thesis is limited to the shaded portion of Figure 2.1 and specifically to the point shown in the center labeled "Distributed Heterogeneous Federated DBMS".

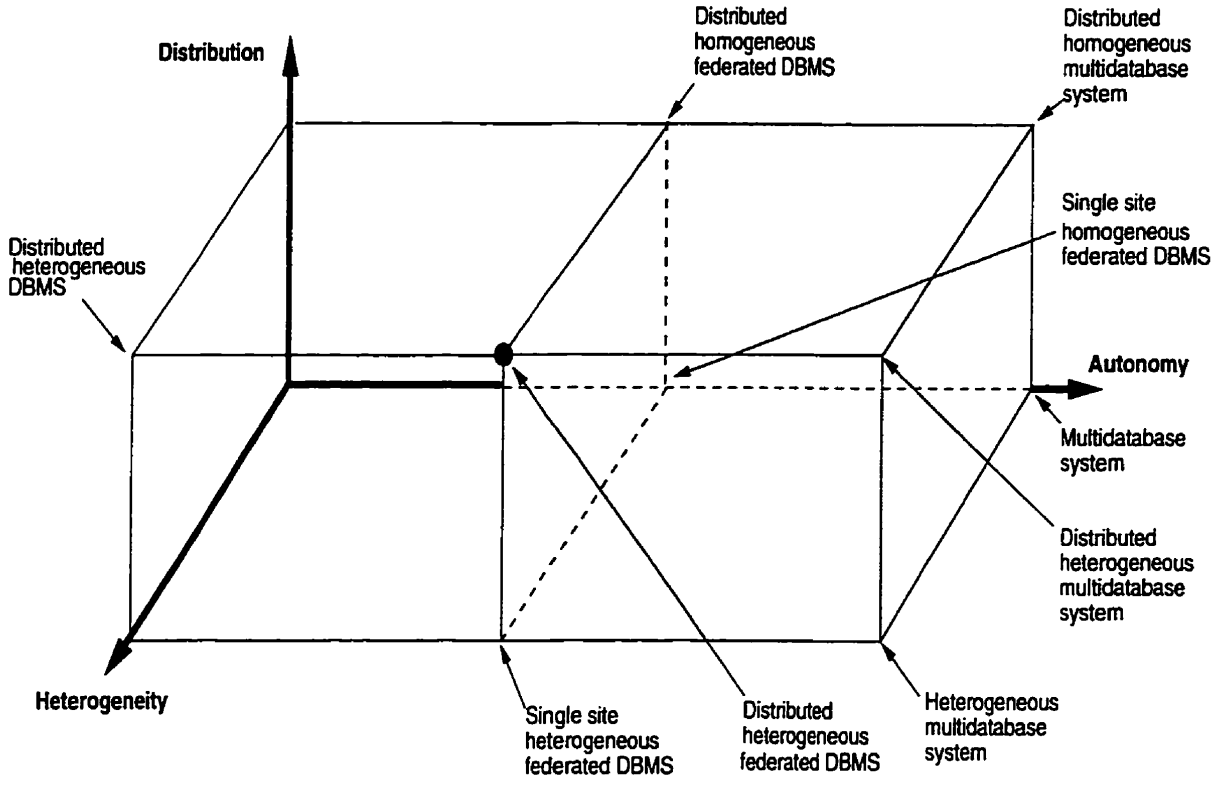


Figure 2.1: Characterization of DBMS

The autonomy dimension deals with the distribution of control, and indicates the degree to which individual DBMSs can operate independently. The distribution dimension deals with the distribution of data and control of the data (i.e. DBMS functionality). There are two possible approaches to data distribution dimensions: data is stored at a single site (centralized) or the data is physically distributed over multiple sites that communicate with each other over some form of communication medium. Heterogeneity in distributed systems may arise due to a number of reasons, but the most typical forms of heterogeneity are: hardware heterogeneity and data heterogeneity.

Data heterogeneity is one of the major obstacles in achieving database interoperability. Data may be distributed among multiple databases in different ways, and may not be structured identically. Sections 2.1 through 2.3 discuss the various issues that influence the interoperability of database systems while Section 2.4 gives an overview of schema evolution

in objectbase systems.

2.1 Advanced Data Models

A data model is a conceptual representation of the data structures that are required by a database. The data structures include the data objects, the associations between data objects, and the rules that govern operations on the objects. Typically, the operations that are done on the data objects are update and query operations. As the name implies, the data model focuses on the structure of the data required and how it is organized rather than what operations will be performed on the data. Operations are performed by the languages that accompany the data model.

2.1.1 Relational Models

The popularity of the relational model [8] is largely due to its simplicity and its strong support for query languages. The relational query algebra [9] is well-defined and this helps to achieve good query optimization strategies. Most of the earlier work on schema integration were based on the relational model [3]. However, the relational model lacks the ability to express a clear semantics of the data, which is indispensable for schema integration. Despite its limitations, many researchers choose the relational model over other models because of its powerful query language and its simplicity.

The relational model appears to be the choice model for *instance level integration.* Instance level integration is accomplished by examining the actual data values in the databases. The instance based approaches assume that different databases have a common key that aides in the identification of relationships. Sometimes the common key may be absent leading to what is known as *key equivalence problem.*

Lim et. al. [18] use the relational model as their common model to achieve instance level

integration. They propose key equivalence based on instance-level functional dependencies to identify tuples that may be equivalent. However, due to the limited information capacity of the relational schemas, there is no guarantee that the integrated schema will dominate the original schemas. A schema S_1 dominates another schema S_2 if there exists a mapping of information bases of S_2 to the information bases of S_1.

Miller et. al. [22] also use the relational model as their common model. This approach considers both *structural equivalence* and *key equivalence* for integrating databases. They augment the information capacity of a relational schema by introducing the notion of *schema intension* to capture the semantic contents of a schema.

2.1.2 Entity-Relationship (ER) Models

Many applications are awkward to represent in the "flat" relational model. This problem has been addressed by semantic data models, whose extensive constructs allow the user to view the data in a more natural way. One such semantically rich data model is the Entity-Relationship (ER) model. Much of the research in schema integration uses the ER model as an underlying model [38, 41]. The ER model [6] has better semantic expressibility than the relational model by allowing greater freedom in terms of naming and design perspectives. Some of the advanced features that are supported by the ER model are generalization and aggregation.

The ER model is a conceptual data model that describes the real world as a collection of entities and relationships. The model visually represents these concepts by Entity-Relationship diagrams. The basic constructs of the ER model are *entities*, their *attributes*, and their *relationships* to one another. Entities are concepts, real or abstract, about which information is collected. Attributes are properties that describe the entities. Relationships are associations between the entities. Several kinds of *integrity constraints* can be expressed in the ER model.

Some of the integration methodologies that employ ER model are Spaccapietra et. al. [41], Larson et. el. [17], and Ram and Ramesh [28]. The primary goal of all these methodologies is the reflection of semantics of the integrated schema from the underlying schemas. The integration methodology presented in [41] supports integration of all types of objects by viewing the schema as a graph with edges and nodes. Using the correspondence assertions between objects, they define a set of rules that are used to generate the integrated schema. Identifying relationships between objects using correspondence assertions was first proposed in [17].

2.1.3 Object-Oriented (OO) Models

The semantic richness of the OO model, together with its ability to support abstraction mechanisms, provide a better tool for discovering *semantic relationships*. Object-oriented Database Management Systems (OODBMSs) store a collection of database objects in a database, with each object representing a certain real-world entity. An object type describes a set of properties that defines the structure and behavior of the objects. The set of all objects of a particular type is called the extent of the type (aka class). This abstraction mechanism of grouping database objects into classes is known as *classification* and allows grouping of instances according to perceived similarities.

Each property has a domain from which its values are taken. The set of values held by properties determines the state of the objects. Sometimes the domain of a property may be a class. Such referential properties are called composite (or aggregation) relationships. Methods encapsulate the state of objects and prescribe their behavior. Methods are identified using their signature. Since methods can be reused by inheritance, conflicts may arise and proper conflict resolution techniques must be employed.

An OO schema is modeled as a set of classes that are linked by different types of relationships such as subtype (also called inheritance) and aggregation (composite) rela-

tionships. Subtyping is an abstraction that allows types to be built incrementally from other types and can be organized into hierarchies. A subtype inherits the semantics of its supertype and can also define additional properties. Aggregation groups interacting objects into a cluster, associating with the cluster those properties that govern the interaction, and then regards the cluster as an atomic entity whenever possible. Some OO models support *multiple inheritance* where a type has more than one supertype and inherits features from all of its supertypes. The advantage with multiple inheritance is the enhanced power in specifying types and an increased opportunity for reuse. A type can multiply inherit from distinct generalizations or from different types in the same generalization if some instances of those types overlap.

Some of the research performed with an object-oriented model as the common data model include, Clamen [7], Bratsberg [4], Gotthard et. al. [13], McLeod and Hammer [20] Reddy et. al. [30], and Thieme and Siebes [43].

Clamen [7] uses the technique of schema versioning to support evolution. The paper also highlights the similarities between schema evolution and integration and vaguely describes how his work on schema evolution can assist with the task of integration. Schema evolution and integration support is also presented in Bratsberg [4]. Their integration is based on maintaining two graphs, the intent graph and the extent hierarchy. However, they do not provide any specific schema update operations that are necessary after initial integration. Gotthard et. al. [13] present a view integration methodology by relying on a semantic model on the basis of structural object-orientation, by utilizing the expressiveness of the model.

The integration methodology presented in Reddy et. al. [30] involves acquisition of semantic knowledge of the objects of local schema objects. This helps in the identification of meta-properties and meta-values of these objects. McLeod and Hammer [20] introduce a mechanism for integrating type objects from heterogeneous sources. They identify and

resolve the semantic heterogeneity by employing a semantic dictionary for all the component objects. A formal treatment to schema integration is presented in Thieme and Siebes [43]. This model is unique because they also consider method integration. However, no support for evolution of integrated schema is presented.

GemStone [25], O_2 [12], Orion [1], and Tigukat [23] are some examples of object-oriented DBMSs. The object model of Tigukat is unique in that it is purely *behavioral* with a uniform object semantics. The model is behavioral because all access and manipulation of objects is based on the application of behaviors (methods). The uniformity of the model is accomplished by treating each component of information solely as objects. The Tigukat object model and its support for schema evolution through a formal axiomatic model serves as a basis for the work presented in this thesis.

2.2 Federated Databases

An important topic in database research is the support of effective sharing and exchange of information among various database systems, without sacrificing too much autonomy of those systems. With the advent of new technologies in communication networks, interconnection of related existing databases have become viable. A collection of cooperating, heterogeneous, autonomous database systems is termed as a *federation* or *federated database systems* (FDBS) and each individual database system in a federation is termed a *component database system* or *component* [20]. The federated database architecture was first introduced by Heimbigner and McLeod [14]. However, the multidatabase architecture proposed by Litwin [19] had many features in common with FDBS. While a component database system participates in a federation, it can also continue autonomously with its local operations. In a FDBS, a certain amount of autonomy of the individual databases is maintained by compromising between no integration and total integration. There is no centralized control in a federated architecture because the component database systems

administrator access to their data. A federation is built by a selective and well-defined integration of the components. The integration of component database systems is typically managed by the FDBS administrator and the users of the federation.

The various schemas that form the schema architecture of an FDBS are *local schema*, *export schema*, and *federated schema*. A local schema is the conceptual schema of a component database and is expressed in the native data model of the component DBMS. In a business environment it is quite likely that the databases may be expressed in a variety of data models. The export schema is a part of the local schema that is made available to the federation and its users. This controlled sharing of data helps exercise autonomy and limits the set of allowable operations that can be submitted on the corresponding local schema. To provide a uniform representation of all the export schemas, they are translated from the local data model to a common data model.

A federated schema is an integration of multiple export schemas. A FDBS can have multiple federated schemas to cater to the requirements of different user groups within the enterprise. Similarly, there could be a federation *database administrator* (DBA) for each federated schema or a single federation DBA for the entire FDBS.

The different architectures of FDBS are determined by arrangement of the various schemas. The activity of developing an FDBS results in creating a federated schema upon which actions such as query and updates are performed. The *loosely coupled federation* [38] has more than one federated schema. In this scheme, each federation user is the administrator of his/her own federated schema and this user is responsible for understanding the semantics of the objects in the other schema which he/she wishes to access. The user is also responsible for resolving any heterogeneity conflict that may arise due to this integration. In a *tightly coupled federation* [38], there are one or more federated schemas. Although it could exist, there does not necessarily have to be a single global conceptual schema as with distributed DBMSs and MDBMSs. A federation that allows the creation and management

of a single federated schema is known as a *single federation*, and those that allow the creation and management of multiple federated schemas are known as *multiple federation*. A single federation is very similar to a MDBMS, except that export schemas are used for the integration instead of the local conceptual schemas. In a tightly coupled federation, export schemas are created by negotiations between a federation DBA and component DBA.

Examples of loosely-coupled FDBS are Calida [15], MRDSM [19], and OMNIBASE [33]. However, Calida differs from the others since the federated schema is generated by the federation users instead of the local users. In MRDSM, the user formulates requests involving data from component DBSs. Based on these requests, loosely coupled federations are formed. Mermaid and Multibase are some examples for tightly coupled FDBS. Barker [2] quantifies the autonomy of various multidatabase systems and based on this quantification, it is shown that an FDBS has a high level of autonomy.

2.3 Schema Integration

Schema integration has been a topic of active research for several years and many methodologies have been proposed in the past decade. Batini et al. [3] give a comprehensive survey of schema and view integration methodologies and investigate twelve of the early integration methodologies. They point out several shortcomings and suggest some general guidelines that serve as a basis for most of the recent integration methodologies.

Current work in schema integration is divided into two schools, one that uses relational data models and the other uses semantic data models such as the entity relationship model and object-oriented model. However, the thrust of the research in database integration is shifting from relational models to entity relationship and object-oriented models. This has largely been due to the apparent inability of the relational databases to express the semantics being captured. At a very coarse level, most integration strategies include the following three steps.

Schema Translation: Schema translation is the process of mapping a schema represented in one data model to an equivalent schema represented in a different data model. In this thesis, the export schema presented by an underlying object model is converted into an equivalent representation in an axiomatic model. The objective of such translation is to achieve data model homogeneity and uniformity in the modeling constructs for later use during schema evolution. It also helps to provide a clear logical view of the entities in the export schemas. The schema evolution functionality of the GemStone object model and the O_2 object model are translated to an equivalent axiomatic model representation in the next chapter.

Inter-Schema Relationship Identification (IRI): This is a *conformation phase* in which the correspondence among schemas are determined. The user supplies correspondence assertions among various entities. After the general correspondences assertions are known, conflicts must be detected and resolved in this phase. The representation of the export schemas in the axiomatic model simplifies the task of identifying inter-schema relationships. Two major kinds of conflicts are *naming conflicts* and *type conflicts*. These conflicts are discussed in detail in the sections to follow.

Schema Integration: Once the conflicts are resolved, these schemas are integrated to form a federated schema. Schema integration is typically a bottom-up process and building a federated schema is one such example. Schema integration is usually done to avoid redundancy of data in organizations having several databases, and to share this data as well.

The term *Schema Integration* has been used in the literature to encompass methodologies for view integration and sometimes for database integration as well. There are however subtle differences among these terms as discussed below.

View Integration

View integration is the process of generating an integrated schema from multiple user views in the design phase of a new database. The important difference between view integration and schema integration is that schema integration attempts to integrate existing databases that are already populated. Typically, in view integration, users define views in a single data model, whereas in schema integration the underlying databases can be heterogeneous. As a result, view integration does not require the schema translation phase since it represents a planned integration.

Gotthard et. al. [13] present a view integration methodology using a object-oriented model. Their integration methodology identifies similarities among object types by first identifying assumption predicates based on the structure of the objects. After assumption predicates are known, factual similarities are identified and this phase usually requires user interaction. They also present rules that help to merge the similar types together.

Database Integration

Instance-level integration logically follows schema integration. In object-oriented terms, the integration of the classes is logically succeeded by the integration of its instances. Once the participating schemas are integrated, the instances can be integrated. The entire process of schema integration along with the instance level integration is known as *database integration.* Some methodologies base their work solely on the actual data values to accomplish integration. Most of this work focuses on integration of relational data models. Since integration is based on data values, any changes in those values would negate the integration effort.

Reddy et. al. [30] present an object-oriented database integration methodology that involves acquisition of semantic knowledge pertinent to the objects of the local schemas. One of the methodologies proposed in [4] does instance level integration for object-oriented

models. They consider objects, as opposed to classes as the basic unit of integration.

Perhaps, one of the biggest obstacles in automating the task of schema integration is the determination of *inter-schema relationships* and the subsequent resolution of conflicts. The process of comparing and identifying schema objects is very challenging due to the fact that different data models may represent the same real world entity in a multitude of differing representations. Such semantic heterogeneity occurs when there is a disagreement about the meaning, interpretation, or intended use of the same or related data. The major causes for semantic heterogeneity can be summarized into the following three points:

1. Naming Conflicts:

   Since the databases that participate in the federation are typically created independent of each other, each designer adapts his/her own viewpoints in modeling similar information. This can occur when designers give different names to the same information that exists in different databases or when they structure similar information differently in the different schemas. One example of a *naming conflict*, is to name a computer as **system** in one schema and **workstation** in another.

   When defining the same real-world object, different designers may not exactly perceive the same set of properties. Such conflicts are known as *descriptive conflicts.* Suppose in one schema a object type **sportscar** has the attributes *model-name*, *manufacturer*, *maximum-speed* and *price*, while in another schema the object type **car_model** has the attributes *name*, *horsepower*, *fuel-consumption* and *price*. Then the same real-world object is being represented in different ways depending upon the individual application requirements. Sometimes two entities that are semantically unrelated might have the same names. Such conflicts are known as *homonyms.*

2. Equivalent Constructs:

   Since the data model helps the user conceptualize the data and processing requirements of the application domains, expressive models such as object-oriented models

allow for a large number of modeling possibilities. This results in variations in the conceptual schema structure. For example, an object type **car** can have **owner** as an attribute (of an atomic type) or as a relationship to another object of type **person**. Even though the schemas may be structurally diverse, they can model the same real world entity.

3. Data Scaling Conflicts:

   Two entities in different databases that are semantically similar might be represented using different units and measures where there is a one to one mapping between the values of the two domains. For example, the price attribute of a car type might have values in dollars in one database and rupees in another.

Identifying these inter-schema relationships has been done in a variety of ways. Larson et. al. [17] base their work on attribute relationships. The general premise is that attributes that are common have several characteristics in common. These attribute characteristics are uniqueness, cardinality, domain, static semantic integrity constraints, dynamic semantic integrity constraints, and security constraints. These characteristics are called basic equivalence properties and different types of equivalences among attributes can be determined by using these characteristics. Based on the characteristics, four types of equivalence between attributes are generated: *equal*, *contains*, *contained-in*, and *overlap*. However, the definition of such attribute relationships is incomplete [36] since the existence of a mapping between equivalent characteristics alone cannot imply that the attributes are equal. In contrast, Sheth and Gala [36] propose reasoning about the attribute relationships in the semantic space. Their approach is based on the *real world semantics* (RWS) of an attribute. Each component of a data model has an associated real world semantics that represents the intended semantics as perceived by the designer. This RWS may not be completely captured by a particular instance of a model as represented in the schema. An attribute definition does not uniquely identify its RWS, and therefore the reasoning in the attribute

definition space does not mean that a corresponding reasoning in real world semantics is possible. That is, if two attributes are equal in the definition space, this does not mean they are equal in the RWS. Reddy et. al. [30] employ a similar method to identify object type correspondences. They integrate similar and related object types into a single integrated object type. However, it can be argued that the integration of similar object types can lead to the creation of a type hierarchy, based on the structure of the types that are related in RWS.

Kashyap and Sheth [37] present a methodology to measure semantic proximity among database objects based on the *context*, *abstraction*, *domain*, and *state* of the objects. The terms are defined as follows. *Context* of the objects is the named collection of the domains of the objects. *Abstraction* is the mechanism used to map the domains of the objects to each other; examples include 1-1 value mappings, generalization, and aggregation. *Domain* is the set of values from which the objects can take their values. *State* of an object is the extension of objects recorded in the database. They claim that two objects having different extensions can have the same real world semantics. Using this proximity measure, they define four types of semantic similarities: *semantic equivalence*, *semantic relationship*, *semantic relevance*, and *semantic resemblance*, with semantic equivalence being the strongest semantic proximity.

Fankhauser et. al. [11] model uncertainty in inter-schema relationships utilizing fuzzy and incomplete terminological knowledge together with semantic knowledge. Fuzzy set theory is used to compute semantic similarities among schemas. The weakness with this approach is that the assignment of fuzzy strengths is based on intuition and such certainty measures depends on the relation between the domains of the entities involved. Sheth and Kashyap [37] introduce a different mechanism for measuring uncertainty using fuzzy logic. In their model, uncertainty is expressed as a function of semantic proximities.

Determining behavioral equivalence requires the ability to compare methods and their

results when executed in different environments, and largely depends on the support of remote execution of procedures. Verv few of the existing integration methodologies consider behavioral integration, the reason being the complexity in determining the behavioral equivalence of classes. Even if the behavioral equivalences between the methods can be established, it is difficult to share a method that is implemented autonomously in one class to be reused in another class and expect the same result from the application of the methods in both the classes. Thieme and Siebes [43] present a formal integration methodology based on the abstract semantics of subclass order to compare classes. They define functional forms of methods and their specialization to aid the comparison of methods. Another approach to behavioral sharing is introduced in [10]. Their approach allows the local component to create local functions on remote objects, thereby decoupling the location of the data from the location of the methods that operate on it. They cite eight different situations based on the locality of the methods in which the behavioral sharing is accomplished.

2.4 Schema Evolution in Objectbase systems

Schema evolution is the timely change of the schema and the consistent management of these changes [27]. In a federated environment, schema evolution logically follows the integration phase. Over time, the changes in federation/user requirements may prompt for changes in the export schemas. In response to these changes, the integrated schema has to evolve to make it consistent with the user/federation requirements. Schema evolution and integration are closely associated and their association is clearly evident in the post integration phase of schema integration. However, this problem has been largely ignored in the literature and the majority of the integration techniques assume that the local schemas are basically static in nature. Clamen [7] discusses the implications between schema evolution and integration and explains how his work on schema evolution through versioning can support integration. However, they do not provide any concrete work on how indi-

vidual schema evolution operations on local schemas can be propagated to the integrated schema. Sull and Kashyap [42] address the importance of schema restructuring after integration and present an algorithm that self-organizes the integrated schema as the local databases evolve. However, their approach lacks formalism and does not consider schema reorganization due to modification of the type structure.

Several researchers have investigated schema evolution in object-oriented databases. Orion [1], Gemstone [25], and O_2 [12] are some of the commercially available OODBMSs that provide schema evolution support. There are two fundamental issues in schema evolution, *semantics of change* and *change propagation* [27]. Semantics of change deals with the effects of the schema changes on the type system. The second issue involves deciding when and how to modify the database objects in order to propagate the changes made to the schema.

One approach to change propagation in object-oriented databases, known as *filtering*, relies on the simultaneous maintenance of multiple versions of classes and objects. With these approaches [7, 4, 39], multiple versions of the same class exists in the same database and existing and new code can operate on existing and new objects without requiring either to be changed. The disadvantage, however, is that the designer must provide routines to make objects appear to be of the version of a class the code is expecting. This task would be even more difficult when multiple databases are integrated into multiple versions. The reason being the extra overhead in supporting versioning at the export schema level and at the federated schema level.

In the *conversion* approach, schema changes initiate an immediate modification of the objects that are affected. Sometimes, these modifications are delayed until the object that is modified is actually accessed. This approach is known as *screening*.

GemStone and Orion support schema evolution using a number of invariants. Evolution is defined in terms of operations that change individual type definitions. O_2 provides high

level operations to manipulate class hierarchies and provide better support in expressing type changes and preserving data.

The basic schema update operations for an object-oriented database are:

1. Create a new object type (class)
2. Delete an existing object type (class)
3. Add a subtype link between existing object types
4. Delete a subtype link between existing object types
5. Modify type: Add a new property
6. Modify type: Delete a property
7. Modify type: Change an existing property

A formal treatment of schema evolution in object-oriented models is presented in [27]. An axiomatic model is introduced that provides a solution for dynamic schema evolution by serving as a common, formal underlying foundation for describing schema operations. This common framework also makes the task of schema comparison easier. Using a set of axioms, the authors illustrate the dynamic schema evolution in Tigukat and Orion. The axiomatic model serves as common object model in this thesis. Two additional object models, GemStone and O_2, are expressed in terms of the axiomatic model in the next chapter. This provides good evidence that a variety of object models can be expressed in terms of the axiomatic model. It also serves as a good basis for schema integration and subsequent evolution of an component schemas of the integrated schema.

Chapter 3

The Axiomatic Model

3.1 Overview

Representing all the export schemas in one unique model expedites inter-schema relationship identification in the later phases of schema integration. In addition, representation of the schemas in an axiomatic model help support evolution of the integrated schema. The proposed methodology uses the axiomatic model of schema evolution of Tigukat to represent the component schemas. The axiomatic model is an object-oriented (OO) model and supports OO properties such as subtyping and inheritance. The export schemas participating in the federation are reduced to their corresponding axiomatic model. The axiomatic model introduced in [27] is a formal specification of dynamic schema evolution in objectbase systems. All characteristics of schematic evolution are formalized into a well defined set of axioms that automatically maintain the complex schema relationships and properties.

Two input sets associated with each type in a schema are used by the axiomatic model. The first set, called *essential supertypes* and denoted by $P_e(t)$ (where t is a type in the schema), contains the types that must be maintained as supertypes of t for as long as it is consistently possible. The second set, called the *essential properties* and denoted by $N_e(t)$, contains the properties that must be maintained in t for as long as consistently possible.

Additional sets are defined and maintained as part of the axiomatic model because of their usefulness in certain operations of an OO model. These additional sets, however, are entirely derived from the essential supertype and property sets. The *native properties*, $N(t)$, of a type t is a set of properties that is not defined in any of the supertypes of t. That is, the native properties of a type t is a set of properties that are natively defined in t. However, native properties of one type t may be defined by other types that are not in a subtype relationship with the type t.

The *inherited properties*, $H(t)$, of a type t is the union of the properties defined by all supertypes of t. The *interface*, $I(t)$, is the union of the native properties $N(t)$ and inherited properties $H(t)$. The interface set serves as a specification of all properties maintained by a type.

Subtyping ($\preceq$) is a facility of object models that allows types to be built incrementally from other types. The usual notation $t \preceq s$ denotes that type t is the subtype of s. The term $\mathcal{T}$ denotes the set of all types in any export schema on which update operations are performed. The subtype relationship between types in $\mathcal{T}$ forms a partial order of elements in $\mathcal{T}$, thereby forming a type lattice.

Definition 1 *Type Lattice $\mathcal{L}$:* A type lattice $\mathcal{L}$ consists of a set of types $\mathcal{T}$ together with a partial order of the elements of $\mathcal{T}$ based on the subtype relationships ($\preceq$). ■

The *supertype lattice types*, $PL(t)$, is the set of all supertypes (includes immediate and essential) of t. The type lattice $\mathcal{L}_t$ of a type t, is the set $PL(t)$ including t, with a partial order of all the types based on the subtype relationships. This is known as *supertype lattice* and is a subset of the type lattices $\mathcal{L}$.

The *apply-all* operation, $\alpha_x(\mathrm{f},\mathcal{T}')$, is used to support the axioms. This operation applies the unary function f to the elements of a set of types $\mathcal{T}' \subseteq \mathcal{T}$ and the function f is defined over a single variable x.

Term	Description
$\mathcal{T}$	The set of all types of a system
$\mathcal{L}$	The type lattice of a system
s,t,t_i,$\top$, $\bot$	Type elements of $\mathcal{T}$
P(t)	Immediate supertypes of type t
$P_e(t)$	Essential supertypes of type t
PL(t)	All supertypes of type t
L_t	Supertype lattice of type t
N(t)	Native properties of type t
H(t)	Inherited properties of type t
$N_e(t)$	Essential properties of type t
I(t)	Interface of type t
α_x(f,$\mathcal{T}'$)	apply all operation

Table 3.1: Notations for axiomatic model

Table 3.1 presents the notation of the axiomatic model and Table 3.2 lists the set of axioms used to guide schema evolution. These axioms are summarized as follows:

Axiom of Closure. All types in $\mathcal{T}$ have supertypes in $\mathcal{T}$.

Axiom of Acylicity. There are no cycles in the type lattice formed from $\mathcal{T}$ and its partial order.

Axiom of Rootedness. A single type $\top$ in $\mathcal{T}$ exists and is the supertype of all types in $\mathcal{T}$.

Axiom of Pointedness. A single type $\bot$ in $\mathcal{T}$ exists and is the subtype of all the types in $\mathcal{T}$. This axiom is relaxed in this work.

Axiom of Supertypes. The set of immediate supertypes of a type t is exactly the subset of the essential supertypes that cannot be reached transitively through some other type.

Axiom of Supertype Lattice. The supertype lattice of a type t includes itself and recursively

all supertypes of t formed from the supertype lattices of its immediate supertypes.

Axiom of Interface. The interface of a type consists of the union of the native and inherited properties of that type.

Axiom of Nativeness. The native properties of a type are the subset of the essential properties that are not inherited.

Axiom of Inheritance. The inherited properties of a type is the union of the interfaces of its immediate supertypes.

Axiom of Closure	$\forall t \in \mathcal{T}, P_e(t) \subseteq \mathcal{T}$
Axiom of Acyclicity	$\forall t \in \mathcal{T}, t \notin \bigcup \alpha_x(PL(x), P(t))$
Axiom of Rootedness	$\exists \top \in \mathcal{T}, \forall t \in \mathcal{T} \mid \top \in PL(t) \wedge P_e(\top) = \{\ \}$
Axiom of Pointedness	$\exists \bot \in \mathcal{T}, \forall t \in \mathcal{T} \mid t \in PL(\bot)$
Axiom of Supertypes	$\forall t \in \mathcal{T}, P(t) = P_e(t) - \bigcup \alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$
Axiom of Supertype Lattice	$\forall t \in \mathcal{T}, PL(t) = \bigcup \alpha_x(PL(x), P(t)) \cup \{t\}$
Axiom of Interface	$\forall t \in \mathcal{T}, I(t) = N(t) \cup H(t)$
Axiom of Nativeness	$\forall t \in \mathcal{T}, N(t) = N_e(t) - H(t)$
Axiom of Inheritance	$\forall t \in \mathcal{T}, H(t) = \bigcup \alpha_x(I(x), P(t))$

Table 3.2: Axiomatization of subtyping and property inheritance

Using these axioms it is possible to present a uniform framework for specifying the semantics of the schema and how it evolves. Furthermore, it provides a common foundation for integration and the subsequent evolution of the export schemas.

The axiomatization of Tigukat and Orion object models are presented in [27]. In this chapter, the GemStone and the O_2 models are axiomatized to illustrate the diversity of object models represented in axiomatic form. This offers a common foundation for the integration and evolution of a reasonable set of schema models from both prototype and commercial OODBMSs.

3.2 Axiomatization of GemStone

Axiom	Support	Remarks
Acyclicity	Yes	Explicitly Supported
Closure	Yes	Not Explicitly supported. However, it is assumed that there are no classes outside the class hierarchy
Rootedness	Yes	Class **Object** is the root of the hierarchy
Pointedness	No	Single Inheritance. No support for pointedness
Superclasses	Yes	No mention of immediate superclasses
Superclass Lattice	Yes	Not explicitly mentioned. Can be inferred from the class hierarchy. Superclass lattice will be in the form of a chain or an ordered list.
Nativeness	Yes	Explicitly supported
Inheritance	Yes	Supports full inheritance
Interface	Yes	Native + Inherited

Table 3.3: Axiomatic Support for GemStone

In the GemStone object model, the terms *class*, *subclass*, and *superclass* relate to *type*, *subtype*, and *supertype* of the axiomatic model. There is no notion of types and a class holds the structure of objects and its instances. There are seven schema update operations defined in GemStone. Indexing of objects in GemStone is based on structure of objects rather than application of behaviors. Since the axiomatic model of schema evolution deals at the semantic level, implementation level details are not considered to be part of schema updates. The axiomatic support for GemStone is shown in Table 3.3.

The data model of GemStone does not accommodate multiple inheritance. This results

in the creation of a simple class hierarchy, unlike most of the object models (including Tigukat) that allow the formation of a type lattice through multiple inheritance. Another drawback with single inheritance is that new edges cannot be added to already existing classes. Edges can only be added when new classes are added to the hierarchy. Similarly, edges can only be dropped when classes are dropped. The GemStone model does not support schema update mechanisms for methods. It does however provide addition and deletion of indices. Since indices relate to the access of actual objects, instead of the schema, they are not considered in this work.

GemStone supports object migration and does not allow explicit deletion of classes. Deletion of a class is only possible if all objects in its extent are deleted first or if all the objects are migrated to another class, e.g., one of its subclasses.

Class **Object** is the root of the class hierarchy (axiom of rootedness). The axiom of pointedness is relaxed as GemStone does not specify a base class. The following is a list of evolution operations supported by GemStone. Each of these are examined in turn:

1. Add a named instance variable.
2. Remove a named instance variable.
3. Rename a named instance variable.
4. Add a class.
5. Remove a class.

Add a Named Instance Variable V: Adding a named instance variable V to a class C will add it to $N_e(C)$, i.e, $N_e(C) = N_e(C) \cup \{V\}$. Due to full inheritance, GemStone explicitly propagates the changes to all the subclasses of C. Axiomatically this is done as $\forall C_1 | C \in PL(C_1), (N_e(C_1) = N_e(C_1) \cup \{V\})$. Conflict resolution is carried out as required by GemStone. The operation is rejected if prior to the addition

$\exists C_1 | C \in PL(C_1)$ and $V \in I(C_1)$. In other words, the operation is rejected if the instance variable V is present in any of the subclasses of C.

Remove a Named Instance Variable V: Removes V from $N_e(C)$ if and only if V is contained in N(C) (i.e., if V is a native property of C and not inherited). Axiomatically, this can be represented as $(V \in N(C))$ leads to $(N_e(C) = N_e(C) - \{V\})$. Subclasses do not lose this property until the delete operation is repeated specifically for each of the subclasses. Although the addition of a new variable propagates the variable to the subclasses, deletion of a native variable will not remove its definition from the subclasses.

Rename a Named Instance Variable V in Class C: This operation is equivalent to changing the semantics of an attribute to reduce possible ambiguity. Renaming is allowed if and only if V is natively defined in C. Axiomatically, $(V \in N(C))$ leads to $(N_e(C) = (N_e(C) - \{V\}) \cup \{V'\})$, where V' is the new variable name. As done earlier with the addition of a new variable, the change is propagated to every subclass of C. In axiomatic terms $\forall C_1 | C \in PL(C_1), (N_e(C_1) = (N_e(C_1) - \{V\}) \cup \{V'\})$. Renaming operation is rejected if the variable V is inherited from C's superclass (i.e, V $\notin$ N(C)).

Add a Class C_1: Create a new class C_1 and add C_1 as a subclass of a given class C. This operation is equivalent to creating C_1 and assigning $P_e(C_1) = \{C\}$. Type C_1 inherits the properties of C and in GemStone this is even stronger as the definition of C are propagated to C_1. Axiomatically this is expressed as $N_e(C_1) = I(C)$. New instance variables are added separately to the newly added class.

Remove a Class C: Removes C from the class hierarchy. The instances are migrated to a subclass of C before the class is deleted. A class is not removed if it has any instances. If $C \neq$ **Object** then $\forall C_1 | C \in P(C_1), P_e(C_1) = P_e(C)$. Note that any class C can have only one immediate supertype because of the single-inheritance hierarchical structure

imposed by GemStone. If C is not the leaf node of the hierarchy, C is removed from the P_e of all of C's immediate subclasses and all these subclasses are made a direct subclass of the immediate superclass of C. The operation is rejected if C = **Object**.

The following schema operations are not supported by GemStone:

1. Addition of an edge.

2. Deletion of an edge.

The above two operations are not defined since the GemStone model does not support multiple inheritance.

3.3 Axiomatization of O_2

In the O_2 model, types are not created explicitly and appear only as components of classes. Furthermore, they do not appear in the inheritance hierarchy. Types define the internal structure of classes. Since the axiomatic model identifies both methods and attributes as properties, adding attributes to the types and adding methods to classes can be combined into the operation of adding properties to classes.

Another difference between schema updates in O_2 and other models is that O_2 supports parameterized updates so that updates can be configured according to the designers update semantics. Certain operations in O_2, like addition of properties to a type and addition of a new type, give the designer an option, in deciding the right update strategy. There are two options that can be chosen when a property is added to a type. The designer can either specify the property or inherit the property from a different type. Another update operation in O_2 that allows parameterized update is the addition of a type. Again, the designer has multiple options. The added type can be a new or a old type and the position of the new type in the type lattice can be specified (either between existing types or at the leaf node level). Table 3.4 lists the axiomatic support for O_2.

Axiom	Support	Remarks
Acyclicity	Yes	Explicitly Supported
Closure	Yes	Not Explicitly supported. However, it is assumed that there are no classes outside the class hierarchy
Rootedness	Yes	Class **Object** is the root of the hierarchy
Pointedness	No	No support for pointedness
Superclasses	Yes	No mention of immediate superclasses
Superclass Lattice	Yes	Not explicitly mentioned. Can be inferred from the class lattice
Nativeness	Yes	Explicitly Supported
Inheritance	Yes	Supports full inheritance
Interface	Yes	Native + Inherited

Table 3.4: Axiomatic Support for O_2

The schema update operations supported by O_2 are listed below. Each operation is discussed in turn:

1. Add a property to a class.
2. Drop a property from a class.
3. Add an edge between classes.
4. Remove an edge.

5. Add a new class.

6. Delete an existing class.

Add Property V to Class C (with associated type T): This operation adds V to $N_e(C)$. The add method operation in O_2 has following syntax:

$$\text{add\_method} < V > \text{ in } < C > [< Signature >, < body >]|||[from < C' >]$$

The parameters $[< Signature >, < body >]$ imply that the method is locally defined in class C. This is equivalent to adding V to $N_e(C)$. The second parameter $[from < C' >]$ implies that it is an inherited property. This again means V is added to $N_e(C)$, but as an effect of the axiomatic model, V $\in I(C')$, V $\in$ H(C), and V $\notin$ N(C). Adding an attribute A to type t is performed axiomatically by adding it to the associated class C of t since the types define the attributes of the class. That is, $N_e(C) = N_e(C) \cup \{A\}$. Conflict resolution of O_2 is carried out in both cases.

The operation is rejected if a similar property has already been defined in any subclass C_1 of C. Similarly, the operation is rejected if a subclass C_1 of C multiply inherit the same property. In axiomatic terms, the operation is rejected if $\exists C_1 | C \in PL(C_1)$, $V \in H(C_1)$. O_2 also checks for behavioral inconsistencies due to the addition of new properties. This is an implementation level process and is below the high-level semantics of the axiomatic model.

Drop a Property V from Class C (with associated type T): Dropping property V removes it from $N_e(C)$. Only native properties can be deleted. Axiomatically, $N_e(C) = N_e(C) - \{V\}$ if and only if $V \in N(C)$. Dropping attributes from type T and dropping methods from class C have the same operation. Conflict resolution of O_2 is performed to maintain consistency.

O_2 ensures behavioral consistency by maintaining meta-information and by the help of a method dependency graph. Each method has information as to which portion of the type it uses. If the deletion of a behavior results in a structural inconsistency, the operation is rejected. That is, deletion of a method can cause signature incompatibility.

Add an edge ($C_1 \preceq C$)**:** Adding an edge establishes a subtype relationship between the two types. This is established with $P_e(C_1) = P_e(C_1) \cup \{C\}$. Both classes C and C_1 must already exist in the class hierarchy. Conflict resolution is carried out and the axiom of acyclicity is maintained.

Remove an edge ($C_1 \preceq C$)**:** O_2 handles removal of edges in different ways - according to the designers update semantics.

- Two exclusive update semantics for the supertypes property exist:
  1. Dropping the edge $C_1 \preceq C$ is achieved by making C_1 the subclass of all the direct superclasses of C. In the axiomatic model, this can be expressed as $P_e(C_1) = P_e(C_1) \cup P(C)$, or
  2. Class C_1 is made the direct subclass of class OBJECT. $P_e(C_1) = \{\text{OBJECT}\}$. This update operation is applicable when C is the only supertype of C_1. i.e, when $P(C_1) = \{\text{C}\}$.
- Propagation of properties is done in one of the following ways:
  1. Properties inherited from C are made local properties. That is, $N_e(C_1) = N_e(C_1) \cup I(C)$, or
  2. Properties inherited from C and not redefined in C_1. Axiomatically, these properties exist in H(t) but not in $N_e(t)$ However, no explicit update operation need to be carried out since the axiomatic model automatically handles this condition.

Add a New Class C_n: O_2 facilitates adding a new class C_n in any position of the class lattice. This is done by specifying a single superclass C and a single subclass C_1 (if any). C must be the immediate superclass of C_1. The equivalent axiomatic representation of this operation is as follows:

1. If no superclass is specified, create a new class C_n where $P_e(C_n) = \{\text{OBJECT}\}$.
2. If only superclass C is specified, create a new class C_n with $P_e(C_n) = \{\text{C}\}$.
3. If both the superclass C and subclass C_1 is specified ($C_1 \preceq C_n \preceq C$):
   The axiomatic model does not explicitly specify subtypes property. It is however possible to add a new class between already existing classes using 3 schema update operations.

Step 1: Create C_n with $P_e(C_n) = \{\text{C}\}$

Step 2: Drop edge $C_1 \preceq C$ as $P_e(C_1) = P_e(C_1) - \{C\}$

Step 3: Make C_n the superclass of C with $P_e(C_1) = P_e(C_1) \cup \{C_n\}$. The final ordering of the classes will be $C_1 \preceq C_n \preceq C$

New classes in O_2 can be added with only one superclass. Additional superclasses are added to $P_e(C_n)$ by adding new edges.

Delete an Existings Class C: Deleting a class C deletes it from the class lattice. If the class is not a leaf node, remove the class and augment set P_e of all subclasses of C with the immediate superclasses of C_1 Axiomatically, this is expressed as $\forall C_1 | C \in P(C_1), P_e(C_1) = P_e(C_1) \cup P(C)$

The delete operation is rejected if

- The class extent is not empty.
- C is OBJECT or any other system defined class.

- A behavior in some other class references the deleted class as an argument or return type.

3.4 Example Application Schemas

In order to demonstrate the effectiveness of the axiomatic model and to show the applicability of the model, a running example with two export schemas is presented. The example application considered in this thesis is the interoperability of library schemas. In this example federation, a university library shares and exchanges information with a public library. The federation users have limited access to both the public library and the University library.

The University library filters the data that is being shared with the federation by limiting the set of materials that can be accessed by the federation. This filtering process is done by making available to the federation only a part of its schema, called the export schema. Such selective sharing helps exercise autonomy and conforms to the local user requirements. Similarly, the public library database limits the access to the federation by defining its own export schema.

Figure 3.1 represents an axiomitized schema that is a part of public library schema made available to the federation. Figure 3.2 represents a portion of University library database that is made available to share with other applications/users. The schemas presented here could have been axiomatized from different object models such as GemStone and O_2. The given example schemas will be referenced from time to time as we the concepts in the remaining chapters are introduced and explained.

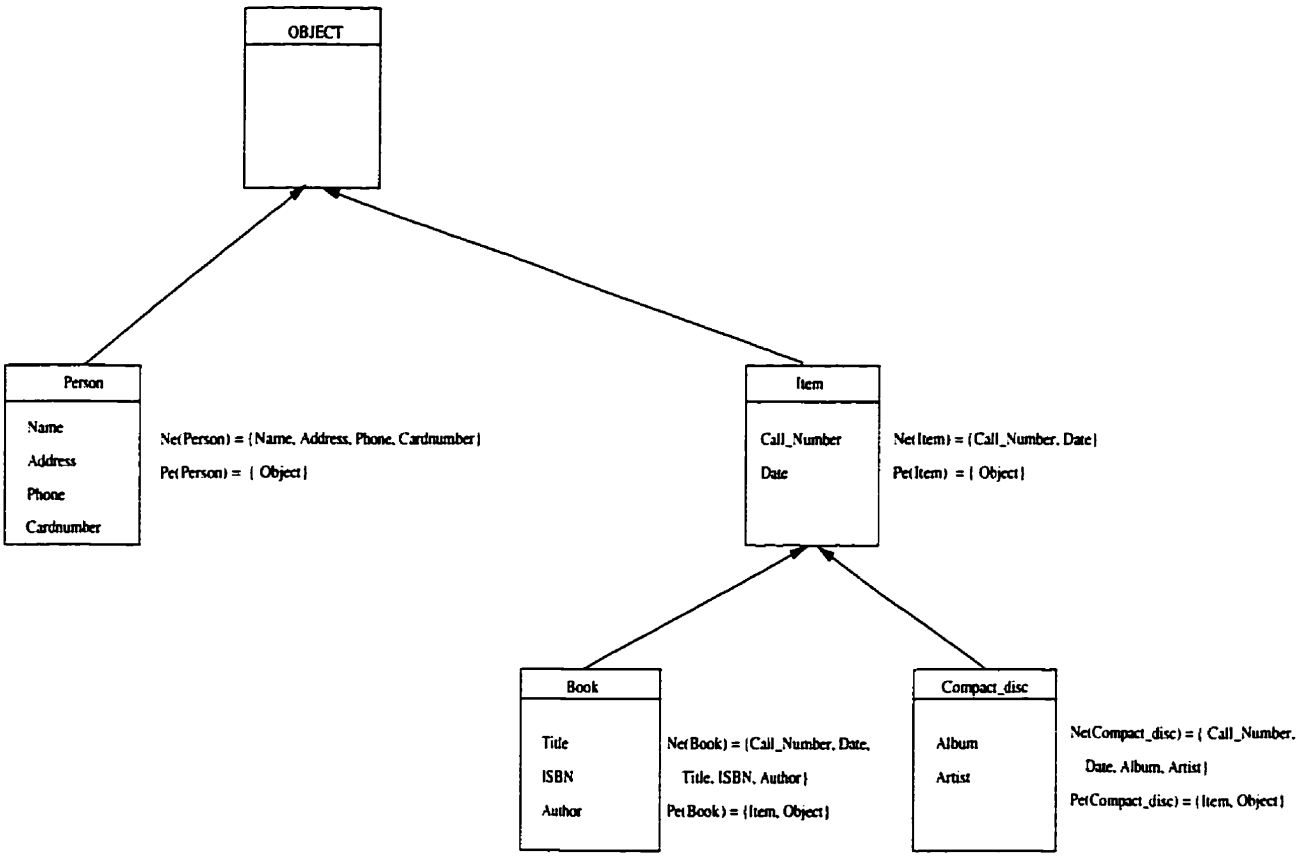


Figure 3.1: Public Library Schema

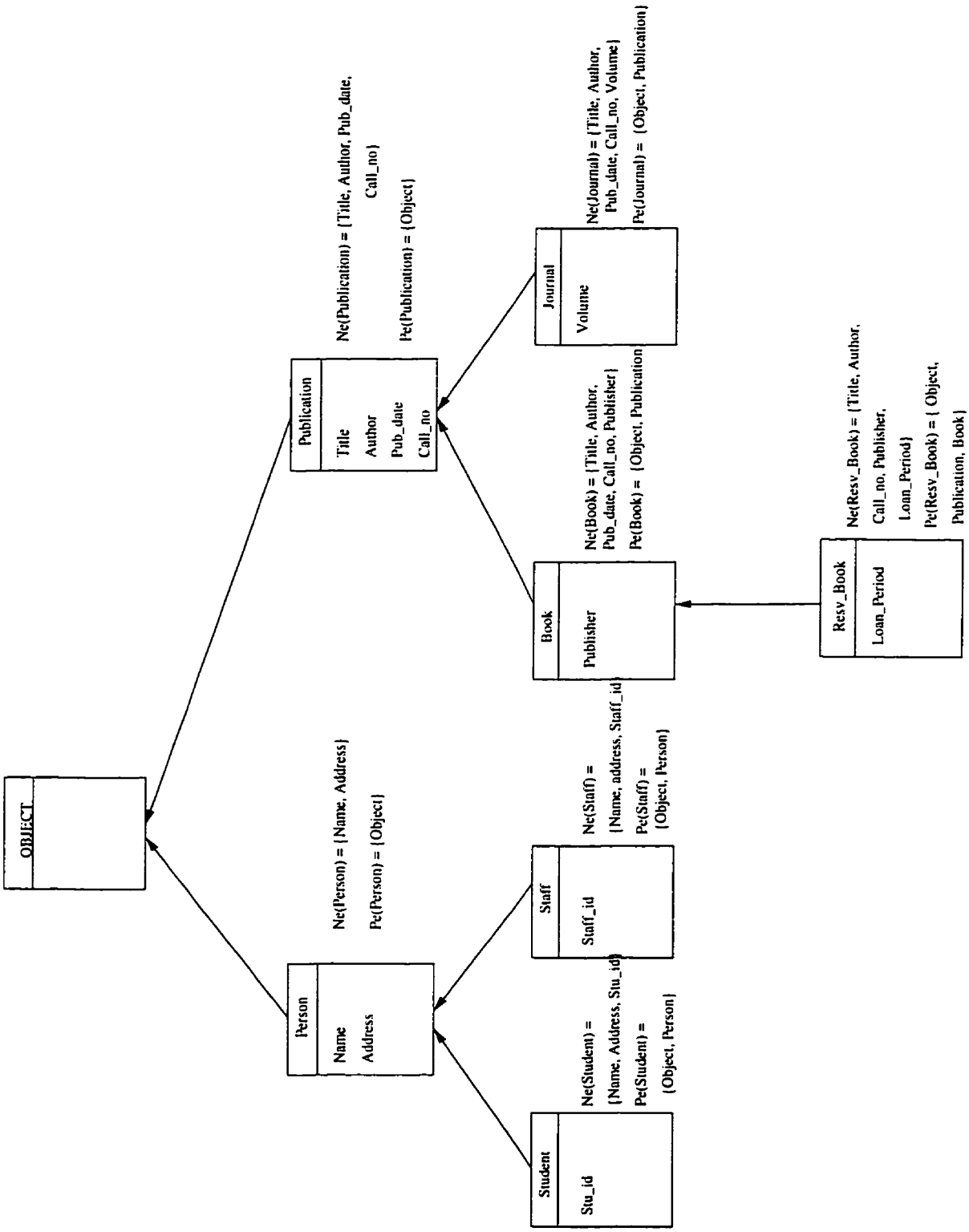


Figure 3.2: University Library Schema

Chapter 4

Initial Schema Integration Methodology: Preparing for Integrated Schema Evolution

The main objective of the inter-schema relationship identification phase is to identify related objects and classify the relationships among them. The difficulty in identifying the *equivalence* between types stems from the fact that the databases participating in a federation are typically created independent of each other. It is not mandatory that there exist a *one to one mapping* between objects in two equivalent object types.

Every type has an associated intent and an extent. Identification of equivalence between object types is based on both the intent and the extent of the types.

Definition 1 *Intent:* An intent is a named set of properties that is used to describe the implementation and the interface of objects. In the axiomatic representation, the intent of type object t is given by $\mathrm{I}(t)$, the interface of the type. ■

Definition 2 *Extent:* An extent is the set of all objects created from the intent. In some models (eg. Tigukat), the extent is axiomatically maintained through a class, while in others the extent must be explicitly maintained by placing objects into appropriate collections. ■

Since similar objects (in real world semantics) may be represented by types with different intensional properties in different schemas, more than just the schematic knowledge is

required to establish relationships between object types of different schemas. Thus the notion of RWS of types introduced in [36] is adapted.

Definition 3 *Real World Semantics (RWS):* The real world semantics of an object type is the set of real world instances of that type in any moment in time. ■

Therefore, the RWS of a type defines the collection of objects that form the extent of that type in any moment in time. However, it should be noted that the extent of an object type may only represent a subset of real world instances that may be present at any time. Sometimes two different types with different intents may actually represent the same real world instances, or at least a subset of their extents may overlap. Thus, the equivalence between types can only be established based on the extent of those types.

Definition 4 *RWS Equivalent:* Two types t_1 and t_2 are said to be RWS equivalent if at least a subset of their extents overlap. In other words, if two types t_1 and t_2 in two different schemas represent the same real world instances, then $\text{RWS}(t_1) \equiv \text{RWS}(t_2)$. This equivalence assertion in RWS is represented by the relation "$\equiv$". ■

In the sections that follow, we abbreviate the RWS equivalence of two types as $t_1 \equiv t_2$.

4.1 IRI Identification

The inter-schema relationship identification (IRI) is based on the *intensional* properties and the real world semantics of the object types. This approach of identifying inter-schema relationships using the real world semantics has been employed in several methodologies. Since the schema does not explicitly have the knowledge of *extensional* information of types, human intervention is required.

Figure 4.1 shows a road-map to generating the initial integrated schema. The process starts at the bottom of the Figure 4.1, shown as stage (1)with the individual export schemas

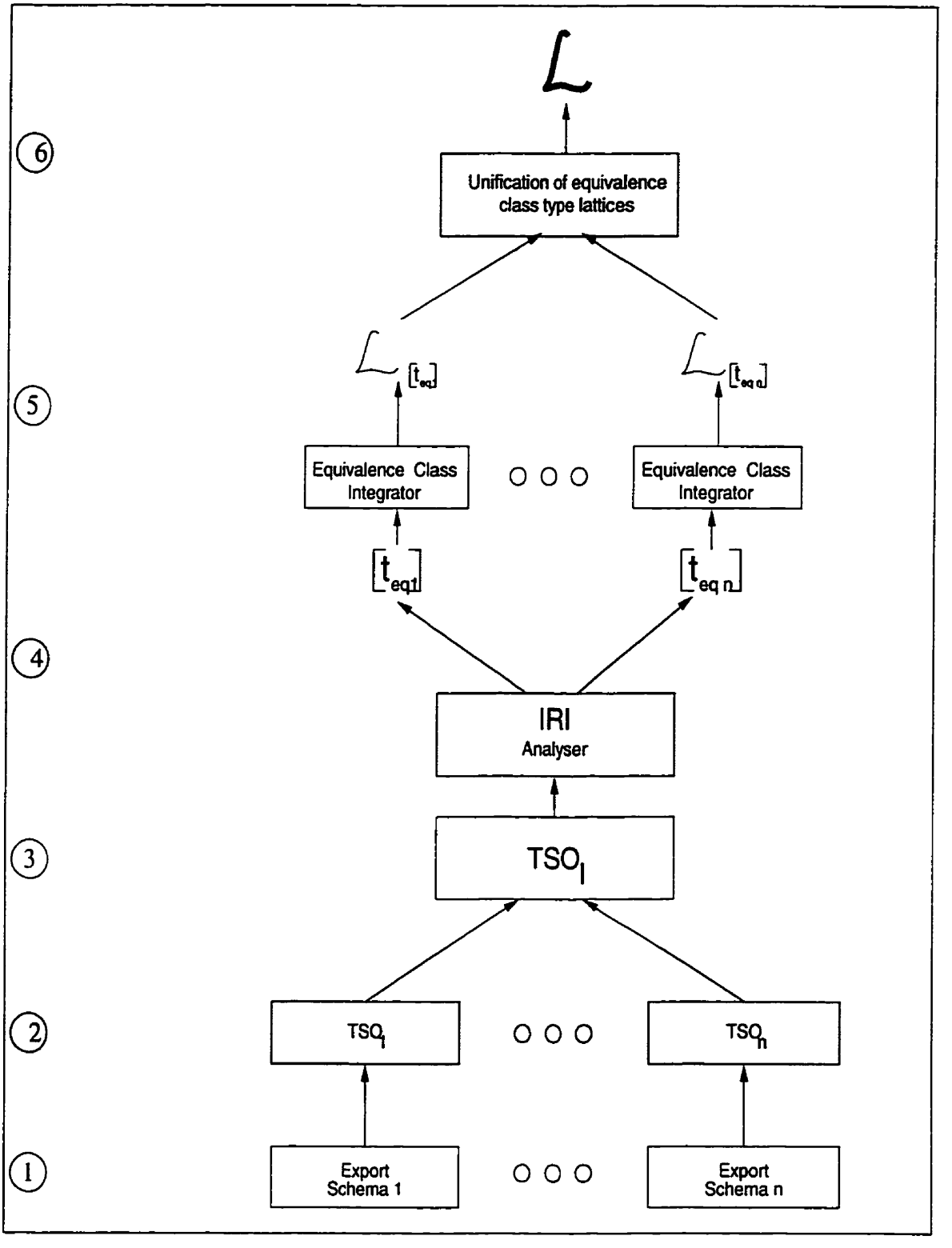


Figure 4.1: Generation of Initial Integrated Schema

from the participating database systems. The types in each export schema $i = 1, \cdots n$ are represented as a set TSO_i, shown as stage (2) of Figure 4.1.

The set of object types considered for integration (stage (3) of Figure 4.1) is given by:

$$TSO_I = \cup_{i=1}^{n} TSO_i$$

It is assumed that in each TSO_i there are no redundant object types as defined in Definition 5 below. The input to the IRI analyzer is a set of all underlying schema types i.e., it is the union of all types in the export schemas participating in the federation.

Example 1. From the example schemas introduced in the previous chapter, the TSO_i for the two export schemas are given as:
TSO_1 (of the University library schema) = {Person, Student, Staff, Publications, Book, Journal, Resv_Book} and
TSO_2 (of the public library schema) = {Person, Item, Book, Compact_disc}
Therefore, the union of all types TSO_I = {Univ-Person, Univ-Student, Univ-Staff, Univ-Publication, Univ-Book, Univ-Journal, Univ-Resv_Book, Pub-Person, Pub-Item, Pub-Book, Pub-Compact_disc}

The types in this example are prefixed with the corresponding schema name.

Similar types (in RWS) in TSO_I may have different names and structural content (intensional properties) due to different modeling constructs. The RWS equivalence of all types in TSO_I is asserted by the designer.

When types from different schemas are integrated, it is very likely that two types from different schemas represent the same real world instances. When such types are RWS equivalent and have the same intentional properties, it would be unnecessary to represent them as two different types in the integrated schema. During initial integration, it is imperative that such redundant object types be identified and replaced with a single type that hold the extent of the redundant types.

Definition 5 *Redundant Object Types:* Given two types t_1, $t_2 \in TSO_I$, t_1 and t_2 are redundant types if and only if $N_e(t_1) = N_e(t_2)$ and $t_1 \equiv t_2$. ■

In Definition 5, the choice of $N_e(t)$ over $I(t)$ for the purposes of comparison of type intents can be debated. Since $I(t)$ is the union of native and inherited properties of a type, it may be possible for $I(t)$ to contain some of the inherited properties that are not essential to the construction of the type t.

Disjoint equivalent classes of types in TSO_I are constructed based on the RWS assertion (again, the constructs of each type in these sets may be different). The IRI analyzer (shown as stage ④ of Figure 4.1) basically identifies the equivalence classes of types from the set TSO_I and groups together types that are equivalent in RWS.

Definition 6 *Equivalence Class* $[teq]$*:* A set of object types $t_1, t_2 \cdots t_n \subseteq TSO_I$ forms an equivalence class $[teq]$ if all the types in the set are RWS equivalent with each other. That is, $\forall t_i, t_j \in [teq]$, $t_i \equiv t_j$. An equivalence class (EC) is given by

$$[teq] = \{t \in TSO_I | t \equiv teq\}$$

That is, a type t in TSO_I is in the equivalence class $[teq]$ if t is RWS equivalent to any type t_{eq} in $[teq]$. ■

Ideally, the integrated schema would have a single type representing each equivalence class. However in reality, the integrated schema may have a single type or a subschema based on the structural similarity of the types in each equivalence class.

Example 2. In the example schemas, two equivalence classes are formed. In this case, the user asserts that the type Person from TSO_1 is RWS equivalent ($\equiv$) to the type Person from TSO_2. Since the types Student and Staff are subtypes of type Person in TSO_1, these types are categorized in the same EC as well. Similarly, the RWS equivalence assertion between type Publications in TSO_1 and type Item in TSO_2 creates the second equivalence class $[teq]_2$

They are given as:

$[teq]_1$ = {Univ-Person, Univ-Student, Univ-Staff, Pub-Person}

$[teq]_2$ = { Pub-Item, Univ-Publication. Pub-Compact_disc, Univ-Book, Pub-Book, Univ-Journal, Univ-Resv_Book}

4.2 Object Type Integration in Equivalence Classes

In stage ⑤ of Figure 4.1, each equivalence class is replaced by an equivalent structure representing a lattice of types in the equivalence class. Before integration of different equivalence classes is carried out, the types in each equivalence class should be integrated to form a *mini-world concept.* That is, all type objects in the equivalence class will form a lattice of object types that are related to a similar concept within the mini-world based on the real-world semantics. From Example 2, it can be seen that the equivalence classes are formed in such a way that the object types are semantically related. The first equivalence class $[teq]_1$ relates to the kind of materials that are available in the library. The second equivalence class $[teq]_2$ gives a list of object types that relates to people who borrow books from the library.

Integration of types in each $[teq]$ is performed by identifying the subtype relationships among types in $[teq]$. Once the subtype relationships are established, their partial ordering gives rise to an equivalence class type lattice.

Definition 7 *Equivalence Class Type Lattice:* Given an equivalence class $[teq]$, the equivalence class type lattice is formed based on the subtype relationship ($\preceq$) of all types in $[teq]$. This is represented as $\mathcal{L}_{[teq]} = \langle [teq], \preceq \rangle$. ■

The process of identifying and establishing the subtype relationships is shown as stage ⑤ of Figure 4.1. The remainder of the section deals with how the essential supertypes of types in $[teq]$ are modified to form the equivalence class type lattices.

To integrate the object types in the equivalence class [*teq*], we need to know how these types are structurally related to each other. In the Tigukat object model, behaviors specify the semantics of the properties. Therefore, there is no notion of attributes and methods. However, most of the current data models do not support this behavioral approach. Hence it is necessary to classify properties into stored attributes and computed methods for the purpose of comparison of their equivalence.

Once we have a set of types that are RWS related, the next step is to determine how these types form a generalization hierarchy within each equivalence class. Attribute equivalence relationships can be established using various properties of the attributes. These properties include: *uniqueness property, lower and upper cardinality constraints, the domain of the attribute, static and dynamic integrity constraints, set of allowable operations* that can be done on the attribute, and *the scale of the attribute* [17]. Once the attribute relationship is identified, appropriate mapping functions are defined so that the equivalent properties map among themselves. Such mapping functions that resolve functional and data dependencies among the attributes are presented in [17]. Once the transformation functions are established between all equivalent attributes, the types in [*teq*] can be integrated.

The equivalence among methods is identified by using their signatures (arguments, return type etc.) [43]. Semantic equivalence among methods can be determined using techniques such as *covariance* and *contravariance* [21]. A covariant relation characterizes the specialization of code. In other words, the signature of the method is redefined in such a way that the new types of both the argument and result conform (are a subtype of) the method being redefined. Contravariance is just the opposite of covariance and is not a practical approach in most cases. In this case the signature of the method is redefined in such a way that a new result type conforms to the original result type but the original argument types conform to the new types. Identifying behavioral equivalence based on

these approaches is discussed in [5, 21]. Although these approaches provide some way of identifying equivalence, they are by no means sufficient to determine the exact semantic equivalence. Conflict resolution and sharing behaviors among various databases is an open research problem.

Structurally, the relationship between two types t_1 and t_2 in $[teq]$ is:

1. **Equivalent** if $N_e(t_1) = N_e(t_2)$
2. **Subtype** if $N_e(t_1) \subseteq N_e(t_2)$
3. **Overlapping** if $N_e(t_1) \cap N_e(t_2) \neq \emptyset$
4. **Disjoint** if $N_e(t_1) \cap N_e(t_2) = \emptyset$

Using the above four relationships, the types in each equivalence class $[teq]$ are redefined. Integration of types in $[teq]$ is carried out in three steps that (a) remove redundant types, (b) establish subtype relationships, and (c) create virtual types. These operations are part of stage (5) of Figure 4.1. The details of these steps are explained below:

Remove redundant types: $\forall t_1, t_2 \in [teq]$, $[teq]$ is redefined as

$$[teq]_r = \{t_{in} | (N_e(t_1) = N_e(t_2)) \Rightarrow t_{in} = t_1 R_{join} t_2)\}$$

Where R_{join} migrates the objects of both t_1 and t_2 into a new type t_{in}. Operation R_{join} also removes types t_1 and t_2 from $[teq]$. Types t_1 and t_2 are joined into type t_{in} and $\forall t \in TSO_I$ replace all occurrences of t_1 and t_2 in $P_e(t)$ with t_{in}. Furthermore, $P_e(t_{in})$ $= P_e(t_1) \cup P_e(t_2)$. Note that $[teq]$ has fewer types once the structurally equivalent types have been removed.

Establish Subtype Relationships: Form the partially ordered set $\mathcal{L}_{[teq]}$ based on the subtype relationship among types in $[teq]$.

$$\forall t, t_1 \in [teq], P_e(t_1) = P_e(t_1) \cup \{t\} \text{ if and only if } (N_e(t_1) \subseteq N_e(t))$$

$N_e(t_1)$ would remain the same but some of the native properties in t_1 would be inherited rather than being natively defined.

Create Virtual Types: Types with overlapping properties are integrated with the use of virtual type t_v. The properties of this type is given by $N_e(t_v) = N_e(t_1) \cap N_e(t_2)$, where $t_1, t_2 \in [teq]$. Equivalence class $[teq]$ is supplemented and redefined as:

$$[teq] = \{t_v | (\forall t_1, t_2 \in [teq])(N_e(t_1) \cap N_e(t_2) \neq \emptyset \Rightarrow (t_1 \preceq t_v \wedge t_2 \preceq t_v))\}$$

Moreover, the following modifications are done to t_1, t_2, and t_v to establish appropriate subtyping relationships:

$P_e(t_1) = P_e(t_1) \cup \{t_v\}$

$P_e(t_2) = P_e(t_2) \cup \{t_v\}$.

$P_e(t_v) = \{Object\}$

The type OBJECT is chosen as the essential supertype of the virtual type t_v. Although it would have been better if we had chosen $P_e(t_v)$ to be some subset of $P_e(t_1) \cup P_e(t_2)$, complications may arise if t_1 and t_2 are from different export schemas. Therefore, it is safer to assign type OBJECT as the essential supertype of t_v. This approach has been used extensively in OO view mechanisms [35].

Example 3. In the example schemas, the two equivalence classes form two distinct subtype hierarchies. Based on the above relationships the type lattice is formed. Figure 4.2 shows the type lattice that is formed from equivalence class $[teq]_1$. It can be seen from Example 2 that $[teq]_1$ has three types from TSO_1 and one type from TSO_2. The subtype relationship between type *Person* from TSO_1 and type *Person* from TSO_2 are established as discussed in the above section. Figure 4.3 shows the type lattice that is formed from equivalence

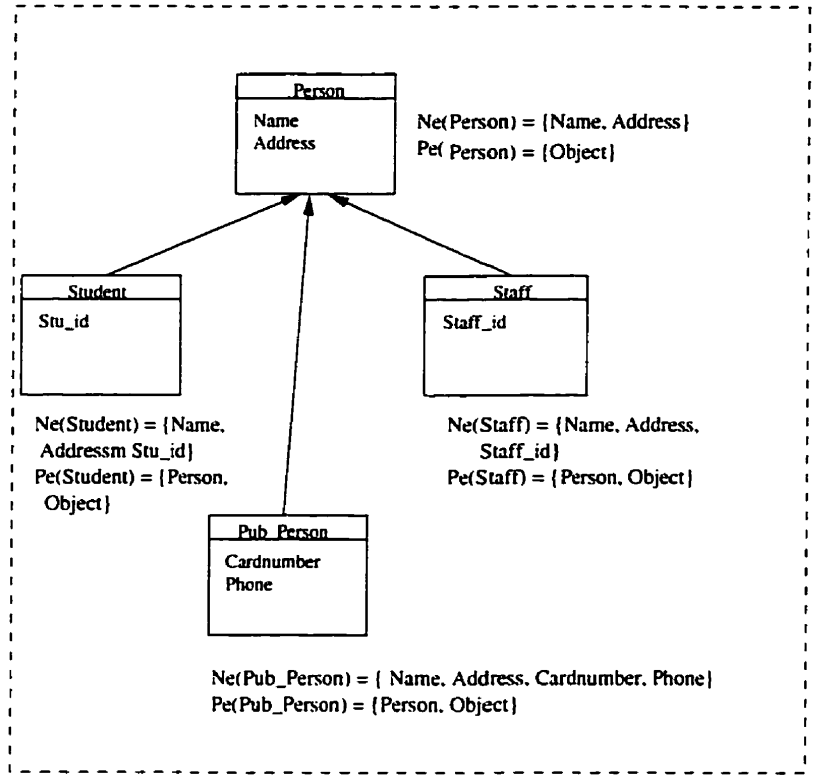


Figure 4.2: Type Lattice Formed from Equivalence Class $[teq]_1$

class $[teq]_2$. Equivalence class $[teq]_2$ is comprised of four types from TSO_1 and three types from TSO_2. The subtype relationships among these types are redefined to form the type lattice as shown in Figure 4.3.

4.3 Integration of Equivalence Class Type Lattices

After the types in each equivalence class $[teq]$ are integrated, the resulting equivalence class type lattices $\mathcal{L}_{[teq]}$, can be unified to form a single integrated lattice $\mathcal{L}$. This step is shown as stage (6) of Figure 4.1. The EC is formed in such a way that the inner types (types other than the root nodes) of the lattice are self contained without subtype relationships with types in other EC lattices. Therefore the integration of the EC lattices is usually a process of establishing subtype relationship with the root type of each EC type lattice.

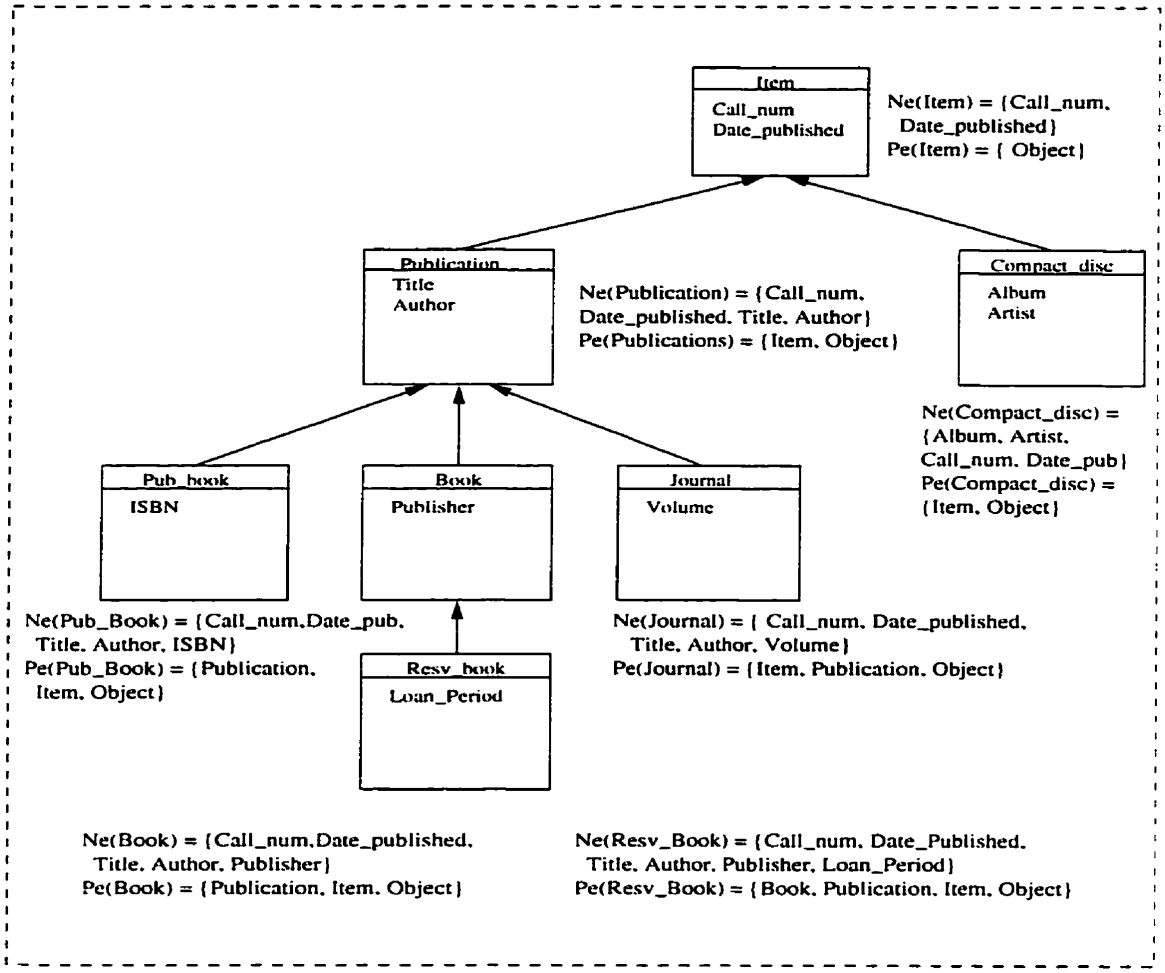


Figure 4.3: Type Lattice Formed from Equivalence Class $[teq]_2$

Example 4. It can be seen from Figure 4.4, that the generation of the integrated schema is usually the process of unifying the equivalence class type lattices as shown in stage (6). The unification is done by establishing the subtype relationship of all root types of the EC type lattices to the type OBJECT. In the example schemas, the integrated schema is formed by establishing subtype relationships between types *Person* and *Item* with type OBJECT.

Since the EC type lattices represent a single domain in the real world, with similar real world objects, it is very unlikely that there would be subtype relationships between two EC type lattices. Therefore, it is sufficient to integrate the EC lattices with a link to type

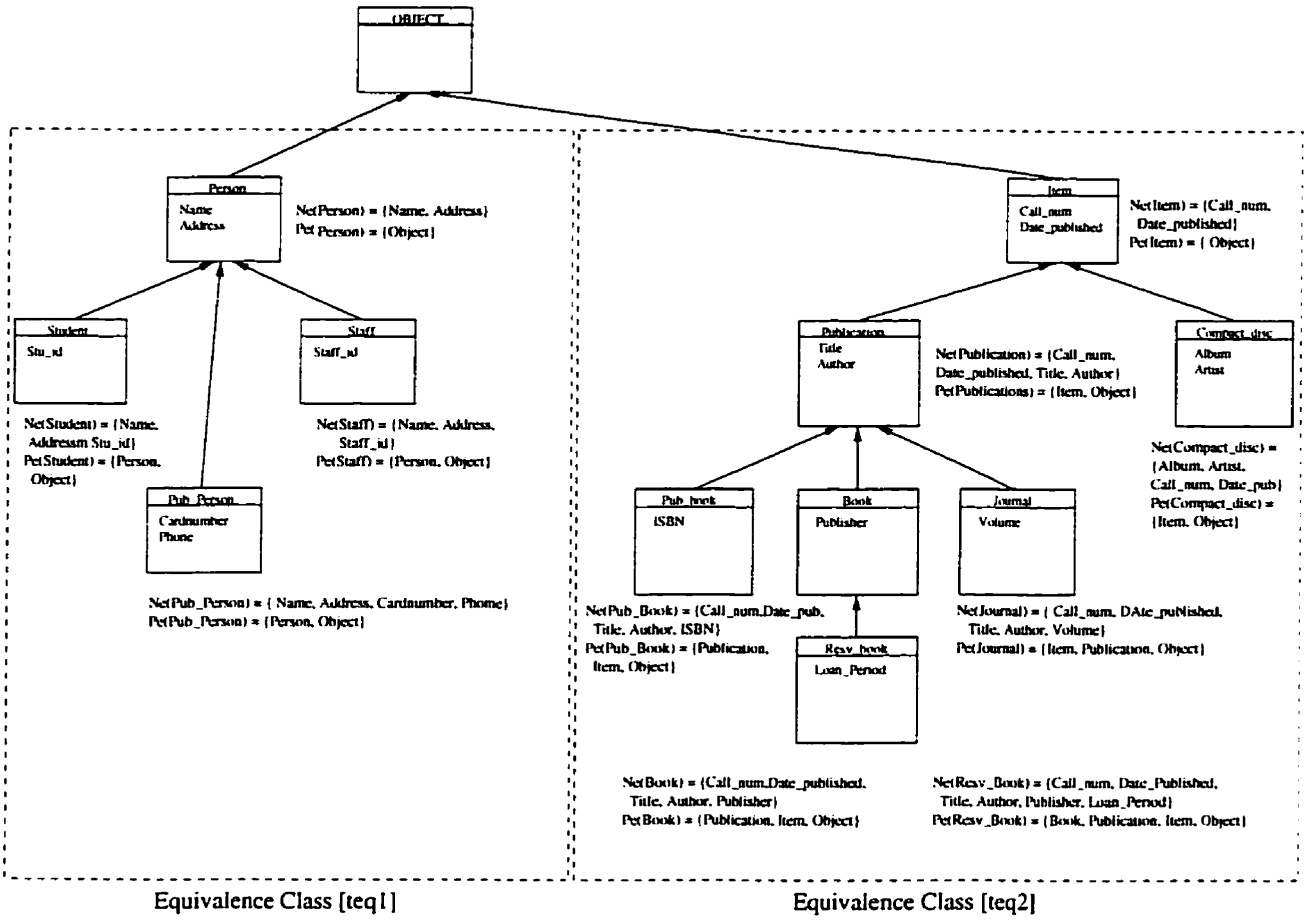


Figure 4.4: Unified Schema

OBJECT.

Chapter 5 deals with maintaining the incremental schema evolution. Schema update operations on the unified schema are presented in Chapter 5 and these operations help to consistently maintain the unified schema with respect to the evolving export schemas.

Chapter 5

Dynamic Restructuring

This chapter deals with the management of schema evolution within integrated federated schemas. An integrated schema must evolve in response to the schematic changes in the participating export schemas. These changes are made in order to meet the needs of the federation users. The semantic integrity of the integrated schema should be maintained when changes are made to the underlying schemas that were used to produce the integrated schema. For example, changes made to the local schemas may influence changes to the export schema, which in turn influence the integrated schema. This is evolution at the initial level where schematic evolution is carried out in the local schemas. The remainder of the chapter presents evolution operations that handle evolution of export schemas by propagating these changes to the integrated schema without the need for complete re-integration of the schemas.

When a local schema is modified, changes to the integrated schema might become necessary. The modifications to the local schema will affect the export schema based on the requirements of the federation users. However, these changes will reflect on the export schemas only if the federation administrator and the federation users wish to do so. Moreover, since each database is considered autonomous, a local evolution should not affect the objects in remote databases.

Operation		
Add Type	Delete Type	Modify Type
subtyping	type deletion	add behavior
		drop behavior
		add subtype link
		drop subtype link

Table 5.1: Schema Update Operations

The evolution of integrated schema is performed when the participating export schemas are modified. These modifications are propagated to the integrated schema based on the axiomatic properties and conditions. Referring back to Figure 4.1 of Chapter 4, the changes to the export schemas are presented at stage ①. Based on these changes, the integrated schema (shown as stage ⑥) is modified. Using the axiomatic properties, stages ②through ⑤ can be bypassed in propagating schema changes when certain update conditions are met. These conditions are outlined in the following sections when the individual schema update operations are discussed in detail.

The most basic schema update operations are *addition*, *deletion* and *modification* of object types as listed in Table 5.1. A few other update operations apart from the ones listed in Table 5.1 are possible. However, most of these operations can be expressed as a combination of the basic schema update operations. Other operations such as creating, dropping, and updating object instances do not affect the schema and are not considered part of the basic update operations.

There are two possible directions from which the schema changes can be initiated. Since we are considering schema integration and evolution in federated objectbase systems, the appropriate direction of initiating the schema changes is *bottom-up*. That is, schema modifications made to the export schema is propagated to the integrated schema. By

propagating changes only in this direction, no modifications need to be made to the local schemas as part of the integration process. This ensures that the autonomy of the local objectbases are preserved and the applications running on the local objectbases are not jeopardized. In contrast, another direction in which schema changes may be initiated is *top-down*. That is, changes made to the integrated schema are propagated to the export schema and ultimately to the local objectbases. However, such changes will demean the autonomy of the local objectbases and the local applications may become vulnerable. The bottom-up approach is the focus of this thesis.

In the following subsections, the basic schema evolution operations on export schemas are examined in detail. The effects of the schema changes on the integrated schema and the propagation of the changes to the integrated schema are discussed as well.

5.1 Add New Types

A new type may be created in one of the local schemas and this may prompt for the addition of the new type to the export schema (after negotiations between the local user and the federation administrator). This type may either be added as a leaf node of the export schema or in-between two existing types. Addition of a new type in-between two existing types is supported in data models such as O_2. Such an operation can be performed in three stages as shown in the axiomatization of O_2 in Chapter 3. Sometimes the type that is added may not have any supertype if it does not specialize from already existing types. In this case, the new type may either form a new equivalence class (EC) or will be categorized as part of an existing EC formed from another export schema.

In response to the addition of a new type t in one of the export schemas, say TSO_1, the user/federation administrator has to provide equivalence assertions for the new type t (possibly against already existing types in other TSO's) if it does not have subtype relationships with some other type in TSO_1. Each time a new type is added, two sets of

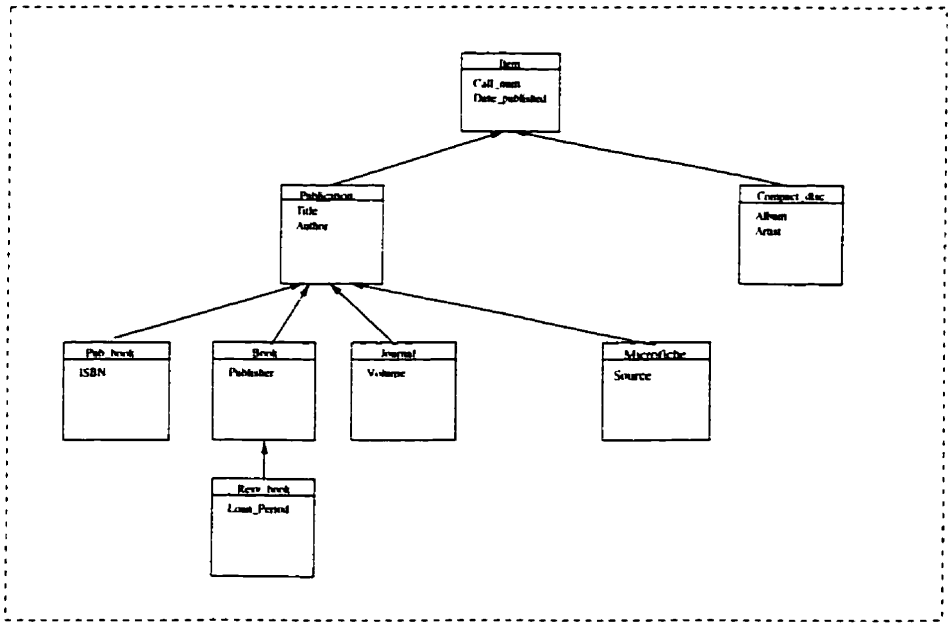


Figure 5.1: Equivalence class $[teq]_2$ after type Microfiche is added

properties, the essential supertypes $P_e(t)$, and the essential properties $N_e(t)$ are provided. The set $P_e(t)$ determines which equivalence class this new type will fit into. Since during initial integration, all equivalent types are classified into one EC, the RWS of a new type t can be determined by the types defined in $P_e(t)$.

If $P(t)$ has only one type t_1, then the RWS can be easily determined. If $t_1 \in [teq]$, then t will also be in $[teq]$ and RWS$(t) \subseteq$ RWS(t_1). The objects in the extent of type t will be more specialized than objects in the extent of type t_1. In other words, the set of real world objects corresponding to type of t is a subset of the set of objects corresponding to t_1. In the example a new type Microfiche with P_e ={ Publication } is added to the University library schema as shown in Figure 5.1. Since Publications is in EC $[teq]_2$, the new type will also be added to EC $[teq]_2$ in the integrated schema.

If $P(t)$ has more than one type, such that all the types are in the same $[teq]$ (multiple inheritance from the same equivalence class), then t will also be categorized into the same equivalence class $[teq]$. That is, $\forall t_1, t_2 \in P(t) | t_1, t_2 \in [teq] \Rightarrow t \in [teq]$ For example, a type

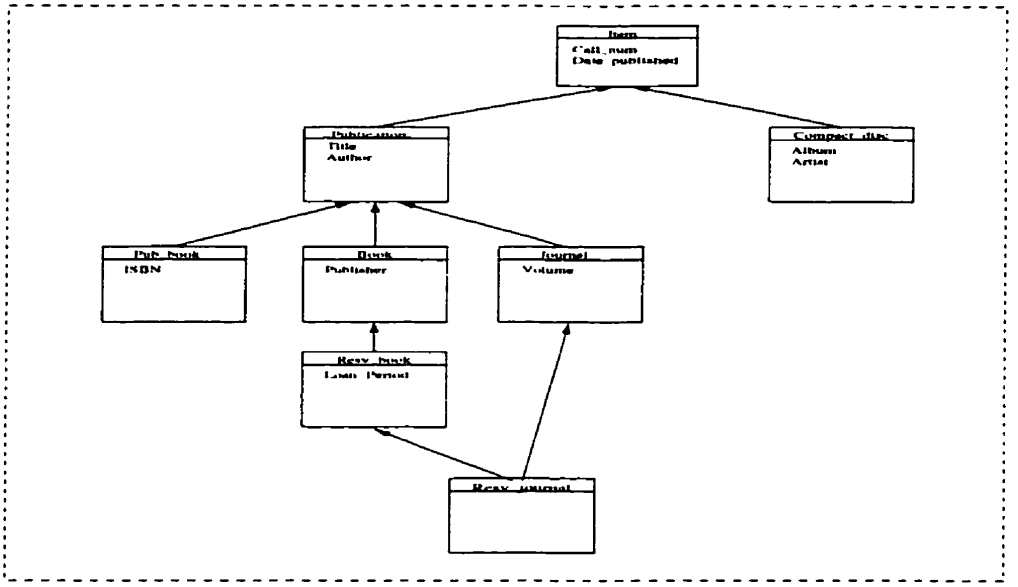


Figure 5.2: Equivalence class $[teq]_2$ after addition of Resv_journal

Resv_Journal is added, with P_e = { Resv_Book, Journal} as shown in Figure 5.2. Since Resv_Book and Journal are in the same EC, Resv_Journal will also be in the same EC.

If $P(t)$ has types that fall into different $[teq]$ (due to multiple inheritance as in the case where there is inheritance from distinct type hierarchies). That is, the new type t inherits from different equivalence classes (ignoring the fact that all the type hierarchies are rooted at the type OBJECT). In such cases, a new EC is formed as follows:

$$\{\exists t_1, t_2 \in P(t) | t_1 \in [teq]_1 \text{ and } t_2 \in [teq]_2 \Rightarrow t \in [teq]_3\}$$

If P(t) = {Object}, then RWS equality is checked against the root types of all equivalence classes from other export schemas. Addition of a new type without subtype relationships with existing types pose additional stages to be carried out. Since the new type does not have a supertype, it is impossible to identify the new type as part of already established equivalence classes without RWS assertions with some other type. Let us assume that an equivalence class $[teq]_3$ was formed from *only* one export schema, say TSO_1, during initial integration. If an RWS equivalent type t (that is $\forall t_1 \in [teq]_3, t \equiv t_1$) is added to TSO_2,

then t will not have a subtype within TSO_2. Therefore the designer should state the RWS equivalence of type t to $[teq]_3$. In the above two cases the designer can explicitly state which EC the new type would fit into.

5.2 Delete Existing Types

A type can be removed from one of the export schemas at either the leaf node level or in the middle of a type hierarchy. A single type t_e in an EC may represent types from different export schemas because of the similarity in their RWS and intent. Therefore, deletion of a type t from an export schema should not result in the deletion of an equivalent type t_e in an EC, unless the type t_e contains the extent of just t. In other words, prior to deleting a type t_e from $[teq]$, it should be ensured that the existence of t_e is solely dependent on t.

Assuming that type t is from TSO_1, the following condition should be satisfied to perform this update operation:

$$\nexists t_1 \in TSO_2 | t_1 \equiv t_e \text{ and } TSO_2 \neq TSO_1$$

The above condition ensures that the type t_e does not contain the extent of another type t_1 from a different export schema, apart from that of type t.

If the type t_e ($t_e \equiv t$) is the only type in the EC $[teq]$ in which t belongs to, the type is deleted and the $[teq]$ is removed from the integrated schema. If the type t_e is a leaf node in $[teq]$, t_e can be deleted as well. That is, if $\nexists t_2 | t_e \in PL(t_2)$ then delete t_e. In Figure 5.2, if type Compact_disk is removed from the Public Library schema (from TSO_2), this type can be removed from the $\mathcal{L}_{[teq]_2}$ as well since no other type inherits from type Compact_disk.

If the type t_e is not a leaf node, deleting t_e will pose new problems. Since the subtypes of t_e may be from different export schemas as opposed to that of t, these subtypes should not lose the properties that are natively defined in t_e. By creating a virtual type that natively defines the native properties of t_e (i.e, $N(t_e)$), multiple definition of the same behaviors in

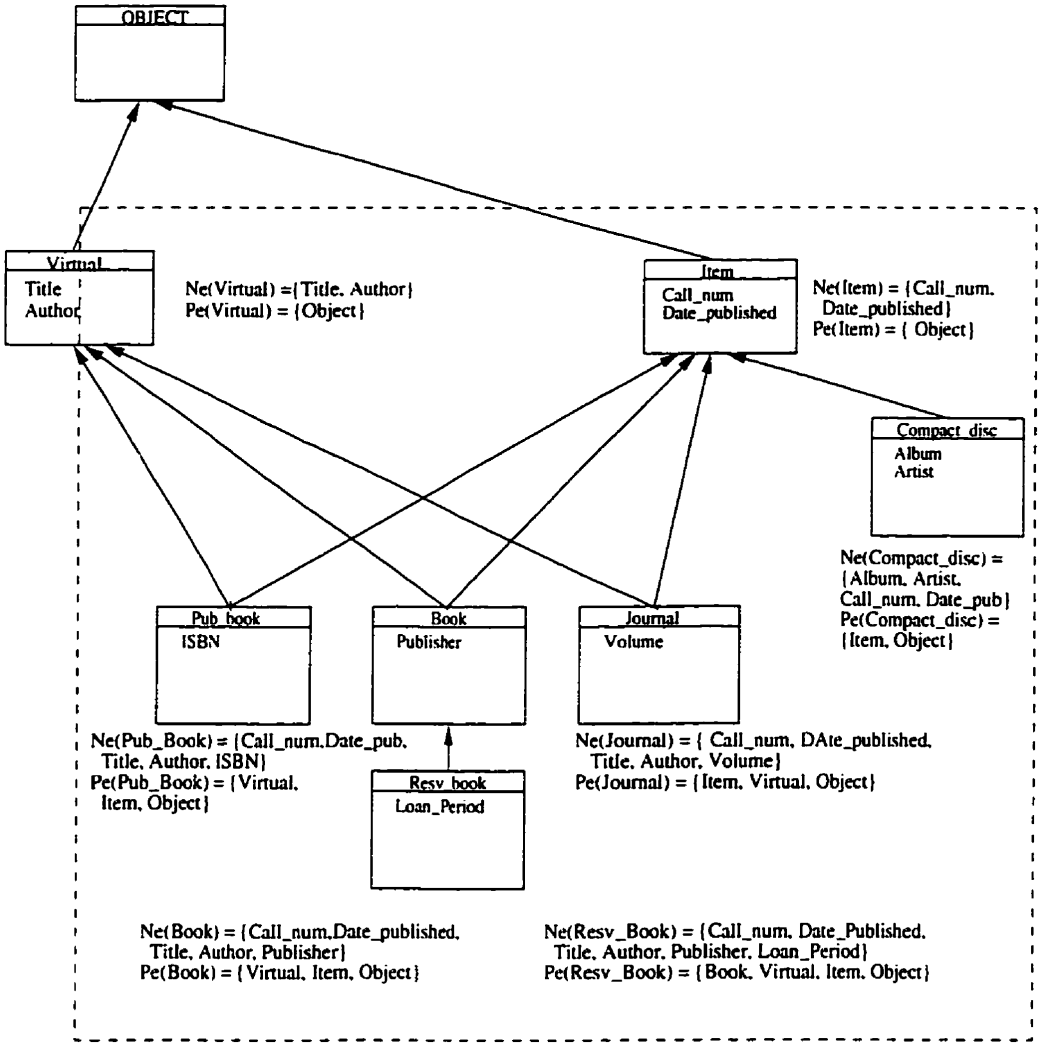


Figure 5.3: Equivalence class $[teq]_2$ after deletion of Publication

all the subtypes can be avoided. Therefore, a similar behavior will not be defined as native property in more than one type in an EC lattice. This approach considerably simplifies the update operations such as addition and deletion of behaviors since equivalent behaviors are defined as native in only one type in an EC lattice.

Axiomatically, if $\exists t_2 | t_e \in P(t_2)$, Create t_v such that $N_e(t_v) = N(t_e)$ and $P_e(t_v) = Object$.

The following condition ensures that the subtypes of t_e form subtype relationship with the virtual type t_v and will no longer have subtype relationship with type t_e.

$$\forall t_2 \in [teq] | t_e \in P_e(t_2) \Rightarrow P_e(t_2) = P_e(t_2) \cup t_v - t_e$$

The essential supertype of the virtual type t_v is defined as type OBJECT since t_v has an empty extent and is added only to avoid multiple definitions of the same properties.

As shown in Figure 5.3, deletion of the type Publication, will create a virtual supertype with its immediate subtypes being Pub_book, Book, and Journal. By creating the type Virtual, properties Title and Author are only defined once in $[teq]_2$.

If the type t_e has the extent of another type from a different export schema, no explicit schema update is made. Axiomatically, if $t \equiv t_e \equiv t_1$ and $N_e(t) = N_e(t_e) = N_e(t_1)$, and $t \in TSO_1$ and $t_1 \notin TSO_1$, t_e is not deleted. In other words, only those instances of t are removed from $[teq]$ and there will be no change in the structure of the EC Lattice $\mathcal{L}_{[teq]}$.

Consider an example where there are two export schemas as given by its TSO_i

$TSO_{TravelAgent\ \mathrm{A}} = \{\text{Person\_info}, \cdots\}$ and $TSO_{TravelAgent\ \mathrm{B}} = \{\text{Customer\_info}, \cdots\}$

If Person_info $\equiv$ Customer_info and N_e(Person_info) = N_e(Customer_info), then in one of the equivalence class $[teq]$, Person_info and Customer_info will be replaced by a single type, say Cust_info, to hold their extents. Therefore the deletion of type Person_info in $TSO_{TravelAgent\ \mathrm{A}}$ will not remove the equivalent type Cust_info in $[teq]$ since Cust_info contains the extent of type Customer_info as well.

5.3 Add New Property

A new property can be added to a type t in an export schema. In order to perform seamless addition of new property into existing types, it would be useful if a list of all properties is maintained for each EC. Therefore, on subsequent addition of behaviors to types, all that is needed is a check to see if a similar property exists in the same EC. If a similar property exists, then the property V is added either by adding a virtual type or with

subtype relationship with the type that already defines the property. Let EC_V be the set of all properties in any EC.

Based on the initial RWS assertion and property equivalence, the equivalence class $[teq]$ and the equivalent type $t_e \in [teq]$ can be established. Let V be the new property that is added to type t. i.e., V is added to $N_e(t)$.

If $V \in I(t_e)$ and $V \notin N_e(t_e)$, then $N_e(t_e) = N_e(t_e) \cup V$. That is, if the property V exists in t_e as part of its interface but not part of its essential properties, V is made essential.

If $\exists\ V_1 \in EC_V | V_1 = V$, either a subtype relationship is created between the type t_{e1} in which V_1 is natively defined or a virtual type is created if $t_{e1} \notin P_e(t_e)$.

If types t_{e1} and t_e already have a type t_{ev} *exclusively* to integrate them such that $P(t_e)$ $= \{t_{ev}\}$ and $P(t_{e1}) = \{t_{ev}\}$, the property V is added to type t_{ev}.

$N_e(t_{ev}) = N(t_{ev}) \cup V$

Since the t_e and t_{e1} are the only types that have subtype relationships with the type t_{ev}, adding a property V as essential to t_{ev}, will add V to the interface of t_e and t_{e1}.

For example, let a property Title be added to the type Compact_disc. Since there already exists a property Title defined in Publication, and types Publication and Compact_disc are the only two types that have subtype relationships with type Item, the property Title is defined as a native property in type Item as shown Figure 5.4.

If the types t_e and t_{e1} do not have any property in common (or other types apart from t and t_1 may be inheriting properties from a common supertype). Create virtual a type t_v such that $N_e(t_v) = V$. The newly created type t_v is made the essential supertype of types t_{e1} and t_e. Thus, the type t_{e1} will no longer have the property V as natively defined. However, this property will be inherited from the type t_v and will remain an essential property. Property V is then made essential to the existence of type t_e.

$N_e(t_v) = V$

$P_e(t_{e1}) = P_e(t_{e1}) \cup t_v$,

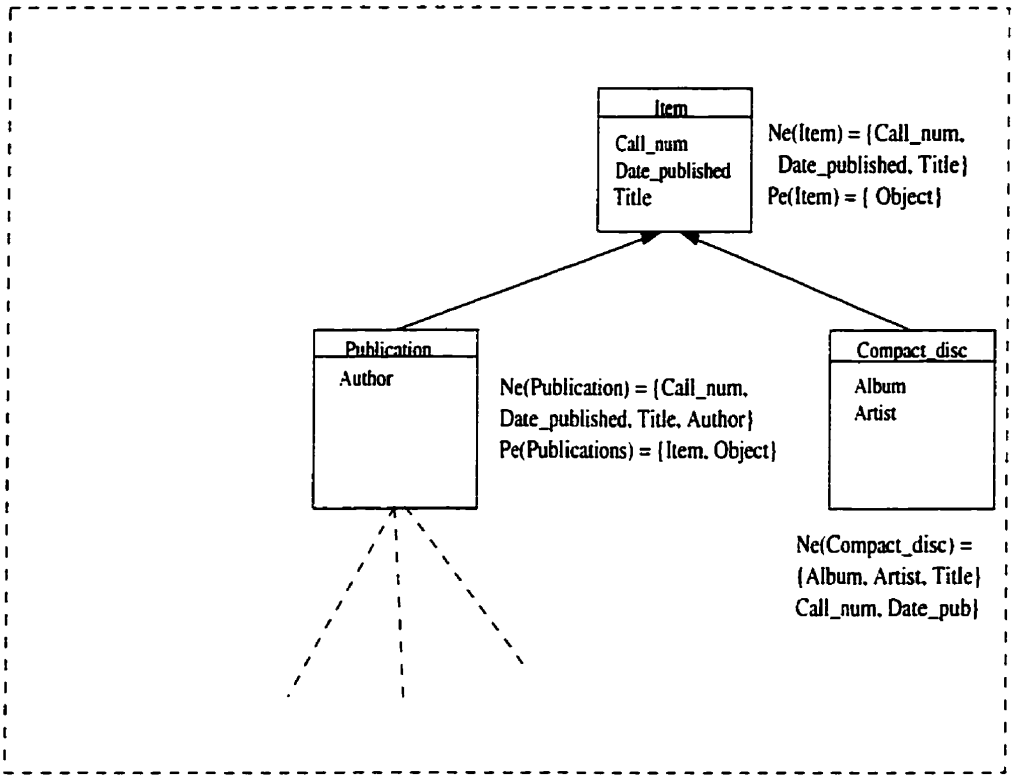


Figure 5.4: Equivalence class $[teq]_2$ after addition of property Title in Compact_disc

$P_e(t_e) = P_e(t_e) \cup t_v$, and

$N_e(t_e) = N_e(t_e) \cup V$.

Supposing a new property Phone is added to the type Staff in TSO_1. Since type Staff is in $[teq]_1$ and there already exists a property phone in type Pub_phone, a virtual class is created to hold the property Phone as shown in Figure 5.5.

If a property similar to V does not exist in $[teq]$, V is added to t_e. In other words, none of the types in the equivalence class $[teq]$ into which t_e belongs, have a property similar to V. That is,

$\forall V_1 \in EC_v$, if $\nexists V_1 \equiv V$,

If the above condition is satisfied, the new property can be added to type t_e and the set EC_v can be updated.

$N_e(t_e) = N_e(t_e) \cup V$ and

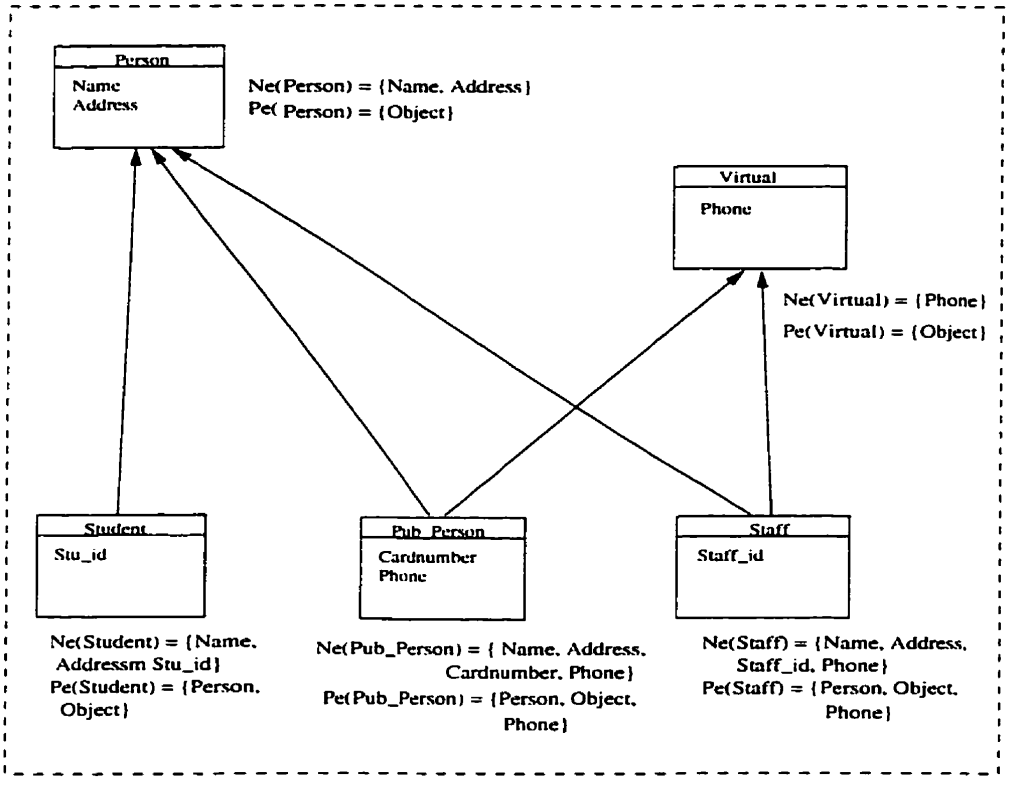


Figure 5.5: Equivalence class $[teq]_1$ after addition of property Phone in Staff

$EC_v = EC_v \cup V$.

In all the above cases, if the type represents extent of types other than type t, null values are assigned to V for those extents.

5.4 Delete Existing Property

This operation deletes a property from a type in one of the export schemas. Certain properties of a type can be deleted from an export schema if the federation application no longer requires this information. Let V be the property being deleted from type t from the set TSO_1. Based on the previous RWS assertions, the equivalent class into which t belongs can be determined. Let $t_e \equiv t$ be a type in the equivalence class $[teq]$ such that $N_e(t_e) = N_e(t)$.

During the initial integration of types in the equivalence classes, any two types (either from the same TSO_i or not) that have one or more behaviors in common are integrated either by a subtype relationship or by creating a virtual type. Therefore, any two types in $[teq]$ cannot have the same behavior defined as native property.

Supposing the property $V \in N(t_e)$, then the following conditions will always be true:

$\nexists t_1 \in [teq] | V \in N_e(t_1)$ and $t_e \notin P_e(t_1)$

The above condition ensures that the set $N_e(t_1)$ will not contain the property V unless it is inherited from t_e.

In the equivalent type t_e, the property V can either be defined natively or be inherited from its supertypes. If the property V is natively defined in t_e, this property can be removed from $\mathcal{L}_{[teq]}$ if t_e does not contain the extents of another type $t_1 \notin TSO_1$.

In the simplest scenario, the extent of t_e is only the extent of t and t_e is the leaf node of $\mathcal{L}_{[teq]}$. Since none of the types in $[teq]$ inherits from t_e, it is safe to remove V from t_e. Axiomatically, $N_e(t) = N_e(t) - V$ (Since $\nexists t_1 | t \in P_e(t_1)$). In Figure 5.6(a), the deletion of property Phone from type Person (from TSO_2) will not alter the structure of $[teq]_1$. In this case the property Phone is removed.

However, if property Cardnumber is deleted from Person (from TSO_2) as well, the type Pub_Person will be deleted from $[teq]_1$. That is, if $N_e(t_e) = \{\ \}$, delete type t_e. The resulting equivalence class $[teq]_1$ is shown as Figure 5.6(b).

If type t_e is in between two (or more) types and if property V is natively defined in t_e, then the property V is removed from the type t_e and virtual type t_v is created with the property V. Subtype relationships are established between type t_v all the subtypes of t_e.

$N_e(t_e) = N_e(t_e) - V$

$N_e(t_v) = V$, and

$\{\forall t_1 | t \in P_e(t_1), P_e(t_1) = (P_e(t_1) \cup t_v\}$.

If property Title is removed from type Publications, a virtual type is added with prop-

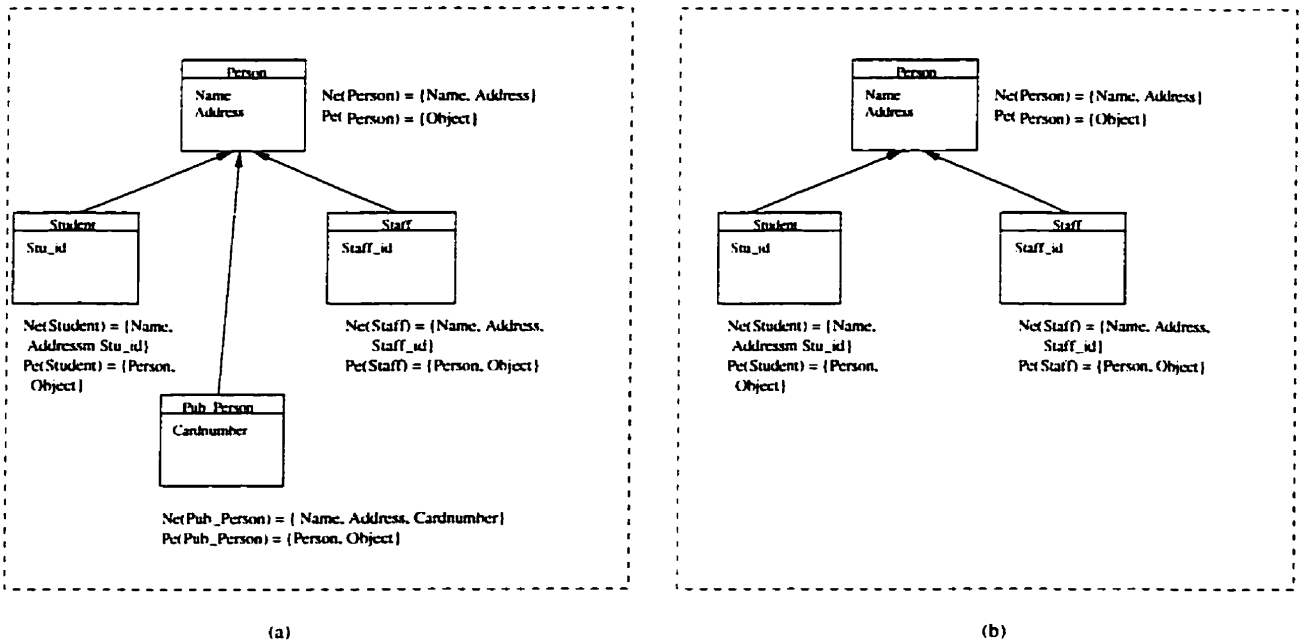


Figure 5.6: (a) Equivalence class $[teq]_1$ after deletion of property Phone from Person (from TSO_2) (b) Equivalence class $[teq]_1$ after further deletion of property Cardnumber from Person (from TSO_2).

erty Title as shown in Figure 5.7

If property V is inherited by t_e from one of its supertypes, the property V cannot be deleted. Also, if the type t_e has the extent of another type (other than t) from a different export schema, no explicit schema update is made. In the above two cases, rather than explicitly deleting the property V from t_e, a null value is assigned to the property V for the extent of the type t.

5.5 Add Subtype Relationships

A new subtype relationship may be added between existing types in an export schema. Only object models that support multiple inheritance can perform this update operation.

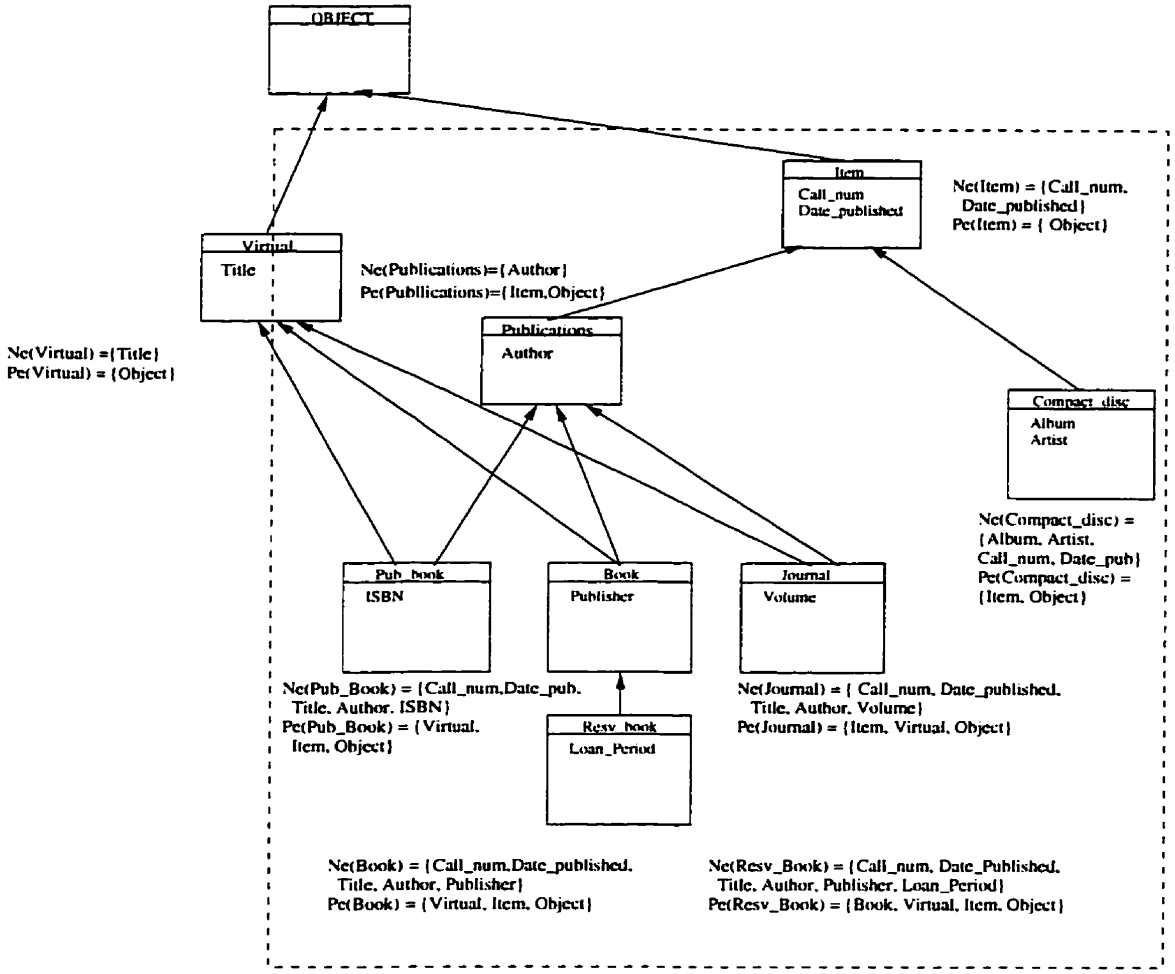


Figure 5.7: Equivalence class $[teq]_2$ after deletion of property Title from Publication

Let t and t_1 ($t, t_1 \in TSO_i$) be the two types between which a subtype relationship $t_1 \preceq t$ is added. By establishing this subtype relationship, type t_1 acquires all the native properties of type t. These properties will be added to $I(t)$ automatically by the axiomatic model. However, to make these properties as essential properties of t_1, they should be explicitly done so, by the *add property* operation as explained in Section 5.3.

A subtype relationship between two type in an export schema is likely to be established only between two RWS related types. That is, in order to establish a subtype relationship, the two types should have at least a subset of their extent in common. Therefore, it can be claimed that the types t and t_1 will be represented within a single equivalence class $[teq]$.

So the addition of a subtype relationship will not result in the creation of a new equivalence class.

Let the types in $[teq]$, t_e and t_{e1} be equivalent to t and t_1 respectively, based on the RWS equivalence and property equivalence. Therefore, the edge $t_{e1} \preceq t_e$ will be established in $[teq]$. It should be ensured that there are no cycles formed after the insertion of the new edge in the class hierarchy. Since the types in any given $[teq]$ may be from different schemas, the subtypes of the type t_{e1} acquire the properties defined in t_e due to the addition of the edge $t_{e1} \preceq t_e$. Therefore, null values will be assigned to the newly acquired properties. The type t_{e1} is added to the set of essential supertypes of type t_e. Axiomatically,

$P_e(t_e) = P_e(t_e) \cup t_{e1}$

In the example schemas, if an edge is established between types Resv_Book and Journal, type Resv_Book will acquire the property Volume as part of its interface but will not be defined as essential property until explicitly done so. This update operation is shown in Figure 5.8.

5.6 Delete Subtype Relationships

Removing edge $t_1 \preceq t$, will not change the EC of t, since all the types in an EC represent the subset or superset of the same real-world concept.

Let the types equivalent to t and t_1 be t_e and t_{e1} in $[teq]$ respectively, based on the RWS equivalence and property equivalence.

Therefore, the deletion of subtype relationship is done axiomatically as,

$P_e(t_{e1}) = P_e(t_{e1}) - t_e$.

Since the subtypes of t_{e1} may have instances from different export schemas, deletion of subtype relationship may in-turn delete some of its essential properties.

Therefore, to remove an edge $t_{e1} \preceq t_e$, it should be ensured that the edge to t_e is added to all the immediate subtypes of t_{e1}. That is, $\forall t_{e2} |\ t_{e1} \in P(t_{e2})$, the following conditions

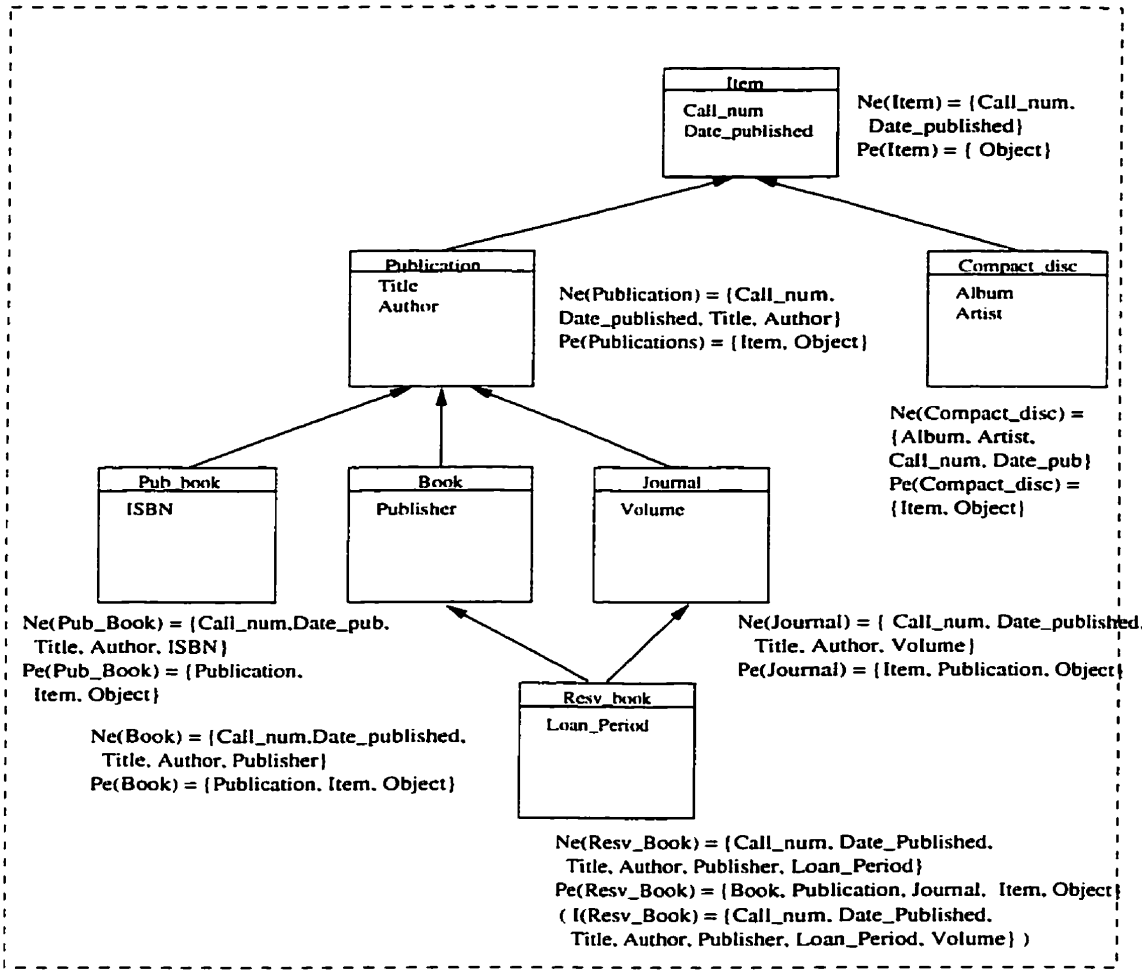


Figure 5.8: Equivalence class $[teq]_2$ after addition of edge Resv_Book $\preceq$ Journal

should hold:

i) $P_e(t_{e2}) = P_e(t_{e2}) \cup P(t_{e1})$, and

ii) $N_e(t_{e2}) = N_e(t_{e2}) - N(t_{e1})$

Referring back to Figure 5.2, if the edge between types Resv_journal and Resv_book is removed, the type Resv_journal will establish subtype relationship with the immediate supertypes of Resv_book, which is type Book in this case. The resulting equivalence class $[teq]_2$ will be modified as shown in Figure 5.9.

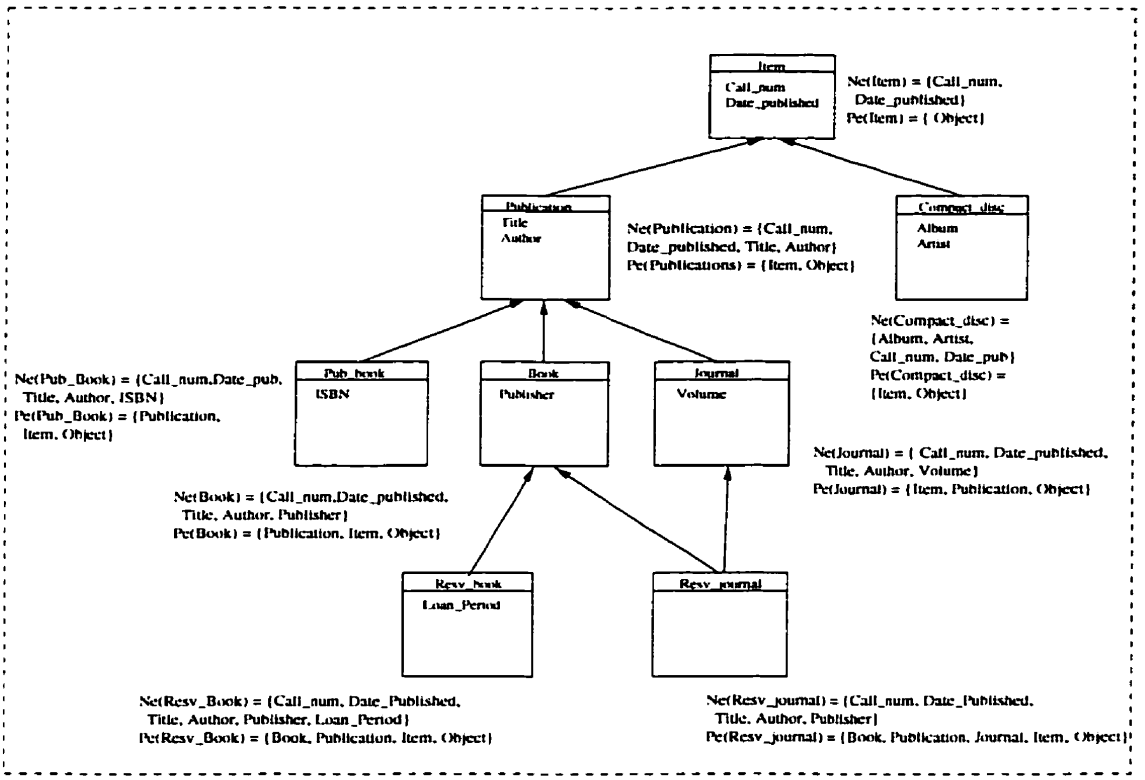


Figure 5.9: Equivalence class $[teq]_2$ after deletion of edge between Resv_book and Resv_journal

Chapter 6

Conclusions and Future Work

The methodology presented in this thesis manages the evolutionary aspects of an integrated schema in the framework of a tightly coupled federated objectbase system. The thesis also addresses the importance of the post-integration phase of schema integration. The axiomatization of export schemas provides a strong common model and serves as a foundation for identifying structural equivalences of types.

In Chapter 3, the axiomatization of two object models, GemStone and O_2 are presented. The axiomatization of other models such as Orion and Tigukat appear elsewhere [27]. This illustrates the diversity of object models supported by the axiomatic model and serves as a common foundation for comparison and identification of equivalent constructs.

In Chapter 4, a framework to identify the equivalence between types is presented. This follows the approach taken in [36] and carries through the aspects of the axiomatized schemas. A new integration methodology is presented based on the existing methodologies. This methodology helps in the creation of an axiomatized integrated schema. Unlike existing approaches, integration is done by identifying real-world equivalent types and forming equivalence classes. A type lattice is created with each equivalence class before integrating all the type lattices. This axiomatized integrated schema helps maintain incremental updates.

In Chapter 5, a framework that supports evolution of the integrated schema is presented. The update operations to the export schema that affect the integrated schema are presented with examples. The main contributions of this thesis are summarized as follows:

- It identifies and examines the importance of supporting evolution at the integrated schema level.
- Diversity in the representation of export schemas within the axiomatic model is expanded by the axiomatization of GemStone and O_2 object models.
- Schema integration is done by determining equivalence classes and integrating types within the equivalence class. The type lattice represents an integrated inheritance hierarchy. The axiomatized integrated schema provides support for incremental updates.
- It presents a formal method based on the axiomatic model of dynamic schema evolution to support evolution at the integrated schema level.

Further research is required to achieve better interoperability among heterogeneous information sources and schema integration is a key technique for providing interoperability. Identifying behavioral equivalence and sharing type methods among objects from different schemas is a challenging task and remains unresolved.

The generation of an integrated schema was done by identifying the real-world semantics and the structural correspondences of object types. The identification of inter-schema relationship currently requires human intervention to produce useful connections. It is possible to partly automate this process if there is a clear understanding of the semantics of the objects represented in the database.

An area of research that has not been considered in this thesis is the adaption of the object instances to the evolved schema. This is the process of coercing objects of the local schemas to coincide with the integrated schema.

In this thesis, the schema changes are initiated by the local schemas and finally propagated to the integrated schema. When autonomy and integrity of the local databases are unimportant, schema changes can be initiated by the integrated schema and propagated down to the local schemas. However, the feasibility and applicability of such systems have to be studied.

Bibliography

[1] J. Banerjee, W. Kim, J. Kim, and H. Korth. Semantics and Implementations of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, May 1987.

[2] K. Barker. Quantification of Autonomy on Multidatabase Systems. *The Journal of System Integration*, pages 1–26, 1993.

[3] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[4] S. Bratsberg. *Integrating Independently Developed Classes*, pages 328–332. Distributed Object Management. Morgan Kaufmann Publishers, August 1994.

[5] G. Castagna. Covariance and Contravariance: Conflict Without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, February 1995.

[6] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[7] S. Clamen. Schema Evolution and Integration. *Distributed and Parallel Databases*, 2(1):101–126, January 1994.

[8] E. Codd. A Relational Model for Large Shared Data Banks. *ACM Communications*, 13(6):377–387, October 1970.

[9] E. Codd. Relational Completeness of Data Base Sublanguages. volume 6 of *Computer Science Symposium on Data base Systems*, pages 65–98. Prentice-Hall, 1971.

[10] V. Fang, J. Hammer, and D. McLeod. *An Approach to Behavior Sharing in Federated Database Systems*, pages 66–80. Distributed Object Management. Morgan Kaufman, 1994.

[11] P. Fankhauser, M. Kracker, and E. Neuhold. Semantic vs. Structural Resamblance of Classes. *Sigmod Record*, 20(4):59–63, December 1991.

[12] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database Systems. In *Proceedings of the 21st VLDB Conference*, pages 170–181, Zürich, Switzerland, 1995.

[13] W. Gotthard, P. Lockermann, and A. Neufeld. System-Guided View Integration for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):1–22, February 1992.

[14] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[15] G. Jacobson, G. Piatetsky-Shapiro, C. Lafond, M. Rajinikanth, and J. Hernandez. CALIDA: A Knowledge-based System for Integrating Multiple Heterogeneous Databases. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 3–18, Jerusalem, Israel, June 1988.

[16] P. Johannesson. A Logical Basis for Schema Integration. In *RIDE-IMS '93 Interoperability in Multidatabase Systems*, Vienna, Austria, April 1993.

[17] A. Larson, S. Navathe, and R. ElMasri. A Theory of Attribute Equivalence in Databases with Applications to Schema Integration. *IEEE Transactions on Software Engineering*, 15(1):449–463, April 1989.

[18] E. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity Identification in Database Integration. In *Proceedings of the International Conference on Data Engineering*, pages 294–301, Vienna, Austria, April 1993.

[19] W. Litwin. An Overview of the Multidatabase System MRDSM. Proceedings of the ACM National conference, pages 495–504, Denver, October 1985.

[20] D. McLeod and J. Hammer. An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(1):51–83, March 1993.

[21] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, New Jersey, 1997.

[22] R. Miller., Y. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1):3–31, January 1994.

[23] T. Özsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal*, 4(3):445–492, 1995.

[24] T Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[25] D. Penny and J. Stein. Class Modification in the Gemstone Object-Oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 111–117, October 1987.

[26] R. Peters. *TIGUKAT: A uniform behavioral objectbase management system.* PhD thesis, University of Alberta, 1994. Available as University of Alberta Technical Report TR94-06.

[27] R. Peters and T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions in Database Systems*, 22(1):75–114, March 1997.

[28] S. Ram and V. Ramesh. A Methodology for Interschema Relationship Identification in Heterogeneous Databases. In *Proceedings of the 28th Annual Hawaii International Conference on Systems Sciences*, pages 263–272, 1995.

[29] V. Ramesh and S. Ram. Schema Integration: Past, Current and Future. In A. P. Sheth, editor, *Heterogeneous Database Systems*, chapter 5, pages 2–34. Morgan Kaufman, 1995.

[30] M. Reddy, B. Prasad, P. Reddy, and A. Gupta. A Methodology for Integration of Heterogeneous Databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):920–933, December 1994.

[31] A. Rosenthal and L. Seligman. Data Integration in the Large: The Challenge of Reuse. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[32] J. Rumbaugh, M. Blaha, W Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[33] M. Rusinkiewciz, R. Elmasri, B. Czejdo, D. Georakopoulous, G. Karabatis, A. Jamoussi, L. Loa, and Y. Li. OMNIBASE: Design and Implementation of a Multidatabase System. In *Proceedings of the 1st Annual Symposium in Parallel and Distributed Processing*, pages 162–169, Dallas, Texas, May 1989.

[34] I. Schmitt. Flexible Integration and Derivation of Heterogeneous Schemata in Federated Database Systems. Preprint Nr. 10, Fakultät für Informatik, Universität Magdeburg, November 1995.

[35] M. Scholl, L. Christian, and M. Tresch. Updatable Views in Object-Oriented Databases. In *International Conference on Deductive and Object-Oriented Databases*, pages 189–207, December 1991.

[36] A. Sheth and S. Gala. Attribute Relationships: An Impediment in Automatic Schema Integration. In *Workshop on Heterogeneous Database Systems*, Chicago, December 1989.

[37] A. Sheth and V. Kashyap. So Far (Schematically) Yet So Near (Semantically). In *Proceedings of the SDS-5 Semantics of Interoperable Database Systems*, pages 283–312, November 1992.

[38] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):184– 235, September 1990.

[39] A. Skarra and S. Zdonik. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pages 393–415. MIT press, 1987.

[40] S. Spaccapietra, Y. Dupont, and Y. Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *VLDB Journal*, 1(1):81–126, July 1992.

[41] S. Spaccapietra and C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):258–273, April 1994.

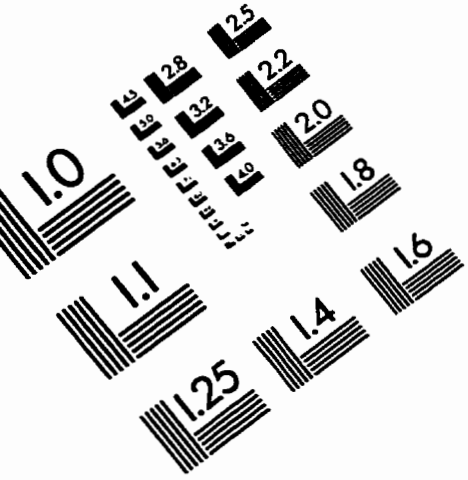
[42] W. Sull and R. Kashyap. A Self-Organizing Knowledge Representation Scheme for Extensible Heterogeneous Information Environment. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):185–192, April 1992.

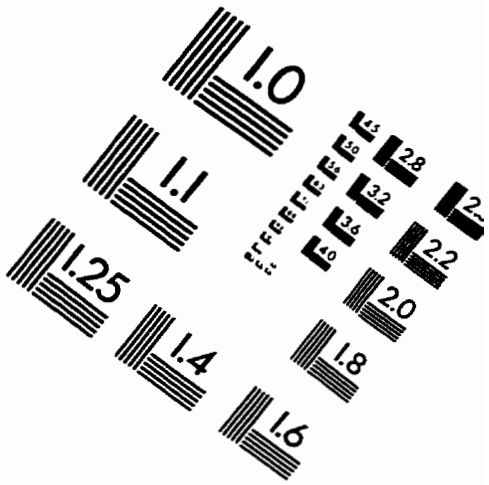
[43] C. Thieme and A. Siebes. Schema Integration in Object-Oriented Databases. Technical Report CS-R9320 1993, Department of Algorithms and Architecture, CWI., Amsterdam, The Netherlands, 1993.

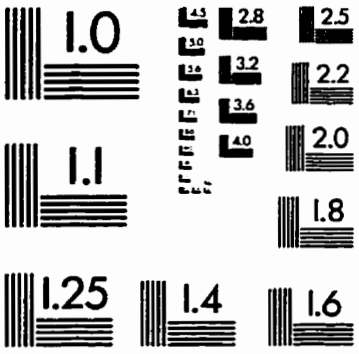
[44] M. Vermeer and P. Apers. The Role of Integrity Constraints in Database Interoperation. In *Proceedings of the 22nd VLDB Conference*, Mumbai (Bombay), India, 1996.

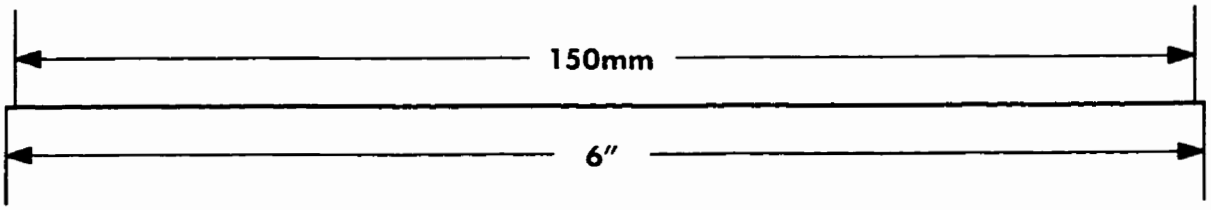
[45] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

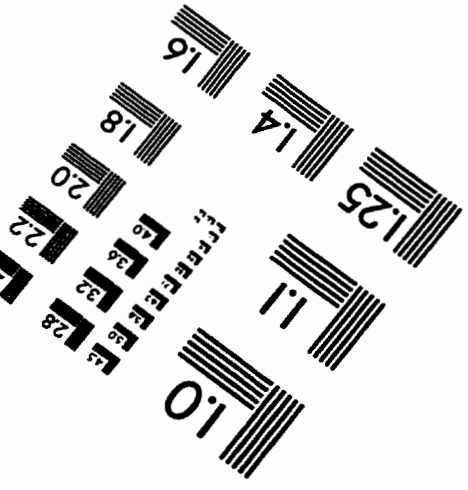
IMAGE EVALUATION
TEST TARGET (QA-3)

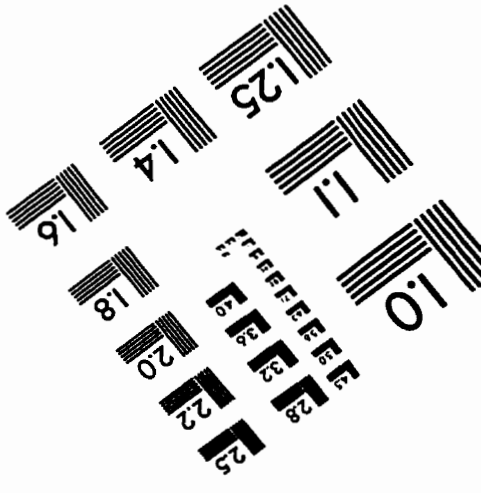












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989