# Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems

by

Joachim Hammer

---

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Science)

August 1994

# Dedication

*To my parents, Renate and Peter,*
*my brother Christian, and*
*Kathryn Stuart.*

> *Every thought which genius and piety throw into the world alters the world.*
>
> *—Ralph Waldo Emerson*

# Acknowledgements

I wish to express my gratitude to the many helpful people at the University of Southern California where this dissertation was written. I have been fortunate to be associated with so many competent and intellectually stimulating individuals who helped me grow into the person I am today. I deeply regret that I cannot acknowledge all who assisted.

First and foremost I would like to thank Dennis McLeod, my research advisor and "mentor", for his continuing and enthusiastic support, advice, and friendship during my graduate years. His insight, high standards of scholarship, and intellectual integrity contributed substantially to the results presented in this dissertation. I was very fortunate to work under his guidance during the past five years.

I am also grateful to the other members of my qualifying examination and dissertation committees—Ellis Horowitz, Richard Hull, Bob Neches and Paul Watkins for serving on these committees and for their helpful comments and discussions while this work was in progress.

Bob Neches and Brian Harp and numerous other people from the SHELTER group at ISI supported me with their resources during the testing and prototyping stages of my research. Thank you for providing such a helpful and stimulating environment.

I sincerely thank all the members, old and new, of Professor Estrin's Network and Distributed Systems Laboratory at USC which provided a home as well as network connections for one of our database machines for nearly five years. This community created a friendly and helpful research environment which greatly strengthened the outcome and quality of this work.

Last, but not least, my friends and colleagues: namely Antonio Si, Jonghyun Khang, K.J. Byeon, and Doug Fang deserve my special thanks for their endurance and their helpful input during our countless research discussions helping me prepare for and overcome the many obstacles along the way. I enjoyed the camraderie that developed while sharing an office with you.

Finally, I would also like to thank George Mullen for making all this possible by providing me with the chance to attend the University of Rochester, my undergraduate alma mater. His memory will provide the inspiration to move on to whatever challenges may follow.

# Contents

# List Of Tables

# List Of Figures

# Abstract

We present an approach to accommodating semantic heterogeneity in a loosely-coupled federation of interoperable, autonomous, heterogeneous databases. Such federations attempt to support operations for partial and controlled sharing of data with minimal effect on existing applications. A major problem that arises frequently when attempting to collaborate among heterogeneous database systems is the occurrence of representational differences that exist among related data objects in different component systems. Within the context of such a federation, components who are willing to share related, relevant information made available by other members of the collection will have to resolve all existing representational differences before they can unify related foreign data with their own local data. We will term representational differences among related data objects *semantic heterogeneity*.

In this dissertation we describe a mechanism for identifying and resolving semantic heterogeneity while at the same time honoring the autonomy of the databases that participate in the federation. By resolving semantic heterogeneity we mean two things:

1. Determine as precisely as possible the relationships between objects that model similar information in different components, and

2. Detect possible conflicts in their structural representations that pose problems during the unification with local objects when sharing information.

A minimal, common data model is introduced as the basis for describing sharable information, and a three-pronged facility for determining the relationships between objects is developed. Our approach serves as a basis for the sharing of related concepts through partial schema unification without the need for a global view of the data that is stored in the different components. The mechanism presented here can be seen in contrast with more traditional approaches such as "integrated

databases" or "distributed databases". An experimental prototype implementation has been constructed within the framework of the Remote-Exchange experimental system at USC.

# Chapter 1

# Introduction

The imminent combination of computing and telecommunications has radically changed our vision of world-wide computing. In this new vision, problems or questions posed by one or more agents (e.g., humans or computers) can be solved as automatically and transparently as possible. However, this requires that the necessary computing resources (e.g., programs, information bases, knowledge bases, etc.) are identified and caused to interact cooperatively to effectively and efficiently solve problems. Thus the dominant computer paradigm will involve large numbers of heterogeneous, information systems (ISs), such as computers (from mainframe to personal), information-intensive applications (multimedia), and data (files and databases) distributed over large computer/communication networks. Networks consisting of heterogeneous ISs are seen as the next generation of information processing systems and the goal is that any computing resource connected to the network should be able to transparently and efficiently utilize any other. Such interconnected environments supporting the sharing of resources are called intelligent and cooperative information systems (ICISs) [50, 51, 58] and the design, construction, use, and evolution of such systems within the above paradigm will require sophisticated technologies from many different areas of computer science.

As a result of rapid technological development in the computer software and hardware, and telecommunication industries, I believe a true revolution in the delivery of information, transactional, and telecommunication services may be upon us. These new technologies expand access to data requiring capabilities for assimilation and analysis that greatly exceed what we now have in hand. Without intelligent processing, these advances will provide only a minor benefit to the users, who will be swamped with ill-defined data of unknown origin. New high-speed networks, for example, could give us access to the Library of Congress, containing nearly 100 million books. However, without the appropriate tools (e.g., intelligent information retrieval systems) to access, search, and exchange this wealth of information the desired data could be hard or even impossible to obtain. Wiederhold [73] proposes the use of mechanisms that will use the context knowledge to mediate among systems to provide for meaningful data exchange. Other examples of intelligent tools that are needed to facilitate user access and processing include presenting an integrated view of multiple information systems, intentional queries, presenting functionality through graphic, visual, linguistic or other support, etc.

## 1.1 Next Generation Database Technology

One of the areas in which collaboration among heterogeneous information systems is desirable is in the area of data/knowledge bases. For example, in the scientific community, in order to keep pace with the wealth of newly discovered knowledge, researchers continue to specialize in smaller and smaller areas. Therefore, the quality and progress of scientific endeavors depends on the researcher's ability to efficiently store large quantities of mostly heterogeneous data and, even more importantly, to share and exchange this knowledge with his/her colleagues. Such environments consisting of a collection of data/knowledge bases and their supporting systems, and in which it is desired to accommodate the controlled sharing and exchange of information among the collection are extremely common in but not exclusively confined to the scientific community. Cooperative work, computer-based manufacturing, scientific databases, and traditional data processing are only a few of the environments where collaboration among autonomous, heterogeneous components is desired. It is obvious that the next generation of database systems must keep pace with this current trend in order to ensure support for large inter-disciplinary projects such as the Human Genome Project and Informatics[1] (HGI) [20].

In light of these observations, participants in the 1990 Workshop on Future Database Systems Research sponsored by the National Science Foundation (NSF) have identified the following goal as one of the key areas in future database research [68]: The creation of an environment that allows for the controlled sharing and exchange of information among autonomous, heterogeneous databases. This is a very ambitious goal and its complete realization depends on the realization of several important sub-goals. In order to support this kind of information sharing, a mechanism is needed to make selected data objects from one database "usable" by other databases of similar or different constitution. For example, the respective databases may differ in their database management system (DBMS), their data model (DM), or their conceptual schema; in this research we are not concerned with differences in hardware or operating systems. Such a collection of cooperating but heterogeneous, autonomous component database systems (DBSs) may be termed a *federated database system* (FDBS) or *federation* for short [25, 27]. A key characteristic of a federation is the cooperation among independent systems, which is reflected by controlled and sometimes limited integration of its autonomous components. This kind of cooperation is often referred to as *interoperability*. To this extent, an FDBS provides an explicit interface to its database components that facilitates sharing while at the same time allowing each component to function independently.

A key aspect of making data be "usable" across heterogeneous databases involves *(partial) schema unification*, and it is imperative to develop methods that can perform this task with as little human intervention as possible. In this dissertation, we use the term "unification" to denote the process of combining different (meta-)data objects together. While the term "(schema) integration" prevails as the predominant expression for activities of this kind, we stress here partially uniting meta-data specifications. By meta-data we mean objects that represent the structural part of a database (i.e., the conceptual schema) as opposed to "factual" data which represents the contents of the database. Rather than unifying complete schemas with each

[1] This project is a major scientific venture under the auspices of both the National Institute of Health (NIH) and the Department of Energy (DOE).

other, a process that is both costly and difficult, partial schema unification is concerned with uniting non-local objects[2] that are selected on demand by components in the federation. One of the fundamental problems that must be solved before one can successfully unify schemas (or parts thereof) is that of *semantic heterogeneity* or *semantic diversity*. In the database context, this heterogeneity refers to differences in the meaning and use of data that make it difficult to identify the various relationships that exist between similar or related objects in different components.

The goal of this dissertation is to demonstrate a new, modular facility for resolving semantic heterogeneity in a federation of autonomous, heterogeneous, object-based databases in order to achieve interoperability. By resolving semantic heterogeneity we mean two things: (1) determine the relationships between objects that model similar information, and (2) detect possible conflicts in their representations that pose problems during the unification of shared data.

## 1.2 About this Dissertation

The remainder of this dissertation is organized as follows: In Chap. 2, we review related work. In Chap. 3, we introduce the context for this research. Section 3.1 describes a typical sharing scenario in a federated database environment where individual components can share and exchange objects. Section 3.2 gives a detailed definition of semantic heterogeneity and other important terms used in this research. A spectrum for heterogeneity is provided and the most common causes for schema diversities are discussed using examples. Chapter 4 describes the different steps that are needed to achieve interoperability between individual components. We consider the use of an intelligent *sharing advisor* that will locate relevant information sources in other components. In Chap. 5, we provide the object context in which we have couched our research. In order for any sharing to take place among heterogeneous components, the components must agree on some common model for describing shared data. The model used in this work is called Common Object Data Model (CODM) which is a simplified version of the Kernel Object Data Model (KODM)

[2] In this context and throughout the remainder of this dissertation, the term "object" may refer to type objects, function objects (also termed methods or behavioral objects), and instance objects.

from *Remote-Exchange* [15]. In this chapter, we also present an overview of the different ways in which (meta-)data objects (i.e. type objects) can be related to each other depending upon the relationships of the real-world concepts they represent. In Chap. 6, we show in detail how our approach to resolving semantic heterogeneity operates. Specifically, we present a strategy for determining the relationship between two objects which is based on three separate mechanisms (parts) for gathering structural as well as semantic information about the objects in question. In Chap. 7, we investigate three scenarios for unifying foreign objects with local objects once ambiguities between the objects have been resolved. In this chapter, we limit our discussion to the unification of type and instance objects. The unification of behavioral objects requires additional work and is treated in detail in Chap. 8. Here we present a comprehensive mechanism for the transparent sharing of methods. Chapter 9 contains a description of two experimental prototypes that were used to prove the feasibility and correctness of our approach. Finally, Chapter 10 contains concluding observations with a critical evaluation of our results and their potential impact.

## 1.3 Some Guidelines for the Reader

In order to facilitate the reading, we would like to establish the following terminological conventions. The term "data" can refer to meta-data such as types, behavioral data such as methods and functions, and factual data such as the actual instances that make up the state of a database. The term "object" generally refers to type objects, function objects, and instance objects unless explicitly stated otherwise (e.g., instance object). In all pictures and tables, the **boldface** type will be used for type objects and concepts, the *italicized* type will indicate function objects. Program code as well as screen output will be typeset in `courier` font.

# Chapter 2

# Related Research

Projects and prototype development efforts to integrate heterogeneous database schemas started in the late 1970's and early 1980's, mostly focusing on providing methodologies for relational database design. Their goals are to produce an abstract, global conceptual description of a proposed database application based on different perspectives and expectations of the users. Subsequent research efforts started to focus on the integration of object-based databases or parts thereof in order to obtain a global view of related information in different domains. The term schema integration, as it is used in most of the reviewed literature, includes semantic heterogeneity resolution, merging, and restructuring. As of today, research efforts have brought about a variety of different sharing architectures utilizing different approaches and mechanisms for integrating heterogeneous databases. Depending on the degree of integration of the participating component schemas different mechanisms to resolve semantic heterogeneity are utilized ranging from manual resolution (before the sharing phase) to semi-automated resolution (during the sharing phase).

## 2.1 View Integration

Early integration work focused on algorithms and (manual) procedures to merge individual, user-oriented schemata into a single global conceptual schema. Views are typically designed for rather narrow domains and thus exhibit a great deal of redundancy in their vocabulary. View integration mainly performs structural transformations from one conceptual schema to another and the process of disambiguation (viz. solving semantic heterogeneity) is much more straightforward than in any other context. There are several different approaches to view integration that can be distinguished by the set of operators used to perform the transformation [4, 13, 26, 54, 53, 55, 65].

In their 1986 survey paper, Batini et al. [5] investigated twelve of these methodologies and compared them on the basis of five commonly accepted integration activities (pre-integration, comparison of schemas, conforming of schemas, merging, restructuring). However, most of the approaches examined in this survey do not directly address the problem of resolving semantic heterogeneity described earlier. Although the problem of achieving semantic interoperability has been studied extensively in view integration using the relational and semantic data models, (partial) integration of multiple heterogeneous object-based databases is still in its infancy.

## 2.2 Database Integration

Schema integration of heterogeneous databases poses substantially new problems. Since names of objects are used much more inconsistently in multiple heterogeneous databases, previous approaches taken from view integration can only be partially applied to the comparison and integration of heterogeneous databases. Another problem that must be addressed when integrating schemas is that of inter-database object correspondences. Work in this area has focused on inter-database object reference by description [62] and object identification via language constructs [35]. Based on these earlier results, there are now several efforts focusing on database schema integration, with varied approaches to handling the problem of semantic heterogeneity.

### 2.2.1 Database Integration Based on Structural Knowledge

One common approach to conflict resolution is to reason about the meaning and resemblance of heterogeneous objects in terms of their structural representation [40].

However, one can argue that any set of structural characteristics does not sufficiently describe the real-world meaning of an object, and thus their comparison can lead to unintended correspondences or fail to detect important ones. Other promising methodologies that were developed include heuristics to determine the similarity of objects based on the percentage of occurrences of common attributes [26, 55, 65]. More accurate techniques use classification for choosing a possible relationship between classes [61]. Whereas most of these methods primarily utilize schema knowledge, techniques utilizing semantic knowledge (based on real-world experience) have also been investigated. Siegel and Madnick [67] use a rule-based approach to resolving semantic conflicts. In their approach they present methods for comparing rules that describe the application's semantic view and the database meta-data definition. The database manager maintains the results of these comparisons for use in query processing. Sheth et al. [63] introduce the concept of semantic proximity in order to formally specify various degrees of semantic similarities among related objects in different application domains based on the real-world context in which these objects are used. Fankhauser et al. [17] present an approach that utilizes fuzzy and possibly incomplete real world knowledge for resolving semantic and representational discrepancies. In their methodology, class definitions, or more generally schemas, are disambiguated by matching unknown terms with concepts in an interconnected knowledge base. A similar methodology is suggested by Heiler and Ventrone [72] that uses so-called enterprise models to provide an application context for describing unknown terms in the schemas of the participating components.

### 2.2.2 Database Integration Using Behavior

A different approach uses behavior to solve domain and schema mismatch problems (see Kent [34]). Domain and schema mismatch are two important semantic integration problems for interoperating heterogeneous databases. The domain mismatch problem generally arises when some commonly understood concept, for example money, is represented differently in different databases (i.e., U.S. dollars vs. English pounds). Schema mismatch arises when similar concepts are expressed differently in the schema (i.e., a relationship that is being modeled as one-to-one in one schema and one-to-many in another). Kent proposes to use an object-oriented database programming language to express mappings between these common concepts that allow a user to view them in some integrated way. It remains to be seen if a language that is sophisticated enough to meet all of the requirements given by Kent in his solution can be developed in the near future.

Despite the many problems that researchers are faced when trying to resolve semantic heterogeneity in object-based databases, there is a variety of promising efforts in the makings. Kim et al. [38] provide the most up-to-date enumeration and classification of techniques for resolving representational conflicts arising within the context of object-based schema integration.

## 2.3 Heterogeneous Database Systems

Research on building complete architectures that support the sharing of information among heterogeneous database systems began only a decade ago [18, 69]. The term "heterogeneous database system" (HDBS) was originally used to distinguish work which included database model and conceptual schema heterogeneity from work in "distributed databases[1]" which addressed issues solely related to distribution [10]. Recently, there has been a resurgence in the area of designing heterogeneous database systems. This work may be characterized by the different levels of integration of the component database systems and by different levels of global (federation) services. In Mermaid [71], for example, which is considered a tightly coupled HDBS, component database schemas are integrated into one centralized global schema with the option of defining different user views on the unified schema. While this approach supports pre-existing component databases, it falls short in terms of flexible sharing patterns. Due to the tight integration of conceptual schemas, no mechanism for dynamically resolving semantic heterogeneity is needed.

The federated architecture proposed in [27], which is similar to the multidatabase architecture of [46], involves a loosely coupled collection of database systems, stressing autonomy and flexible sharing patterns through inter-component negotiation. Rather than using a single, static global schema, the loosely coupled architecture

[1]The term distributed database is used here as it has been mainly used in the literature, denoting a relatively tightly coupled, homogeneous system of logically centralized, but physically distributed component databases.

allows multiple import schemas, enabling data retrieval directly from the exporter and not indirectly through some central node as in the tightly coupled architecture. Examples of loosely coupled FDBSs are MRSDM [45], Omnibase [60], and Calida [31].

Most of the above architectures do not explicitly address the problem of resolving semantic heterogeneity. The architects rely on manual resolution techniques or the existence of integration mechanisms mentioned earlier.

## 2.4 Object Sharing

Besides work on resolving semantic heterogeneity, much effort has been spent on the problem of inter-component object sharing. This involves the placement of objects such as instances, types, and methods in the best possible location in the existing schema of the importing component. Sharing of methods, also termed behavior sharing has attracted most of the attention due to the particular difficulties that are involved. At the top level, there are two distinct aspects of behavior sharing: (1) the remote execution of behavioral objects, and (2) the location of the actual information units upon which behavioral objects operate. Research in the area of distributed programming languages has of course addressed issues of the remote execution of behavioral objects (which may also be termed operations, methods, or functions) [44, 70].

Work in the area of database systems, on the other hand, predominantly focuses on the manipulation of information units; this is related to the second main aspect of behavior sharing. Research on object-oriented database systems has approached the problem of supporting behavior in the database itself [3, 19, 39, 41, 49]. When these systems are extended to a distributed environment, it is the location of the (persistent) data that determines the location of the remote execution of the functions [19, 39, 49]. In the approach used in this research, for example, methods are implemented without explicit knowledge of where they will be executed or where the data will reside.

# Chapter 3

# The Federated Database System Context

As stated in the introduction, the trend towards decentralization of computing that has occurred over the past decade has accelerated the need for effective principles, techniques and mechanisms to support information sharing and exchange among distributed heterogeneous databases. Ideally, we would like to have a single, world-wide database system, from which not only scientists but every user can obtain information on any topic covered by data that has been made available. Assuming that global weather data has been made available, an agent in Capstat (e.g., a travel agency booking a trip for one of its customers) would be able to access the predicted air and water temperatures for the Los Angeles area between March 10 and March 14, 1994, for example (see Fig. 3.1). While such a system is still many years away, we must first start with a much smaller and more restricted version that will help drive the technology needed for world-wide interconnection of information.

## 3.1 A Sharing Scenario

Imagine the following scenario involving the travel business. Several travel agencies located in different cities of the country decide that it would be mutually beneficial to join forces and form a nation-wide Federation Of Travel Agencies (FOTA). The goal of FOTA is to share and exchange travel related information between the individual

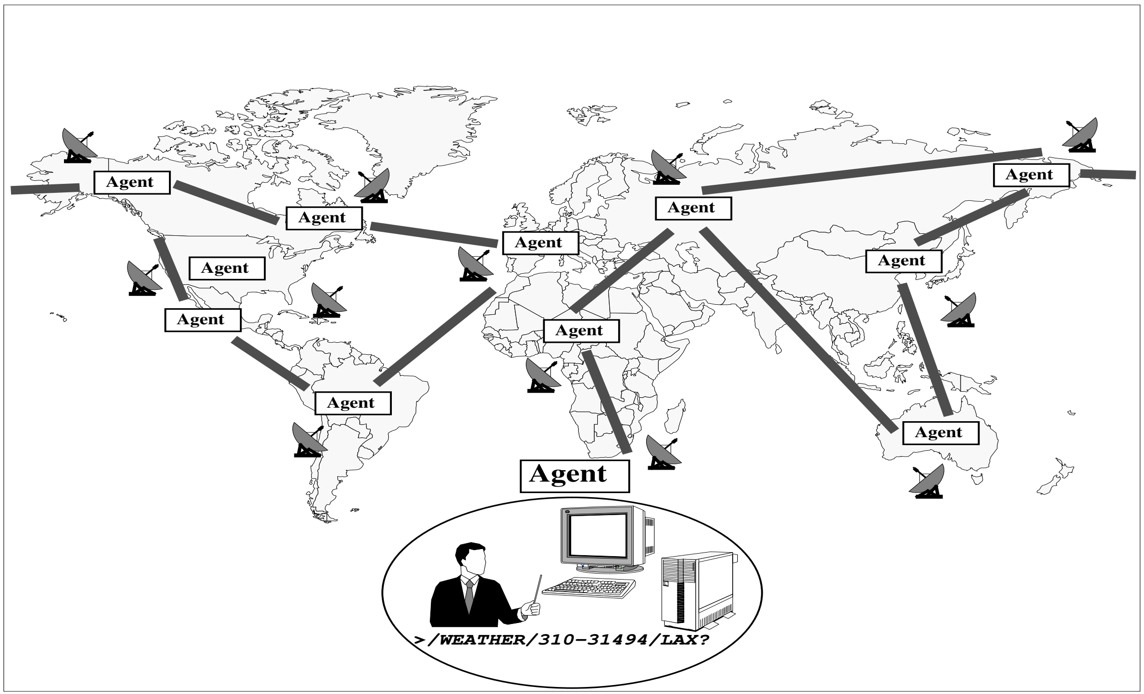


Figure 3.1: A vision of world-wide interoperability.

agencies in order to stay competitive and keep up to date with the fast pace in the world of business and pleasure travel. However, each travel agency wants to retain autonomy over its own database with respect to organization and administration. Therefore, when a component agrees to joining the federation it will keep its own local DBMS together with its original conceptual schema. The main advantage of this is that the costly and inefficient process of restructuring the component's existing data is avoided. Furthermore, there is no need for travel agencies to retrain their employees on a new system. We may assume that since the agencies are located in different parts of the country, the contents of their databases reflect the different travel habits of their customers. Thus FOTA depicted in Figure 3.2 is an example of a loosely coupled collection of heterogeneous database systems, or FDBS, as described in Section 2.

Components of FOTA keep track of the following kinds of information: rental cars, airline information, train information, pleasure cruise information, sight-seeing information such as museums and historic buildings, local entertainment such as shows and events, and hotel information. We also assume that each travel agency will agree to use an object-based database model at the federation interface. Components may update their own local schema at any time, and it is the responsibility of an importing component to obtain access to new information. Figure 3.3 shows a snapshot of the information stored in the federation at a given instance during its lifetime. As mentioned before, the content and organization of each database differs from travel agency to travel agency. We can see, for example, that travel agency $B$ in Miami is the only component with information on pleasure cruises, due to the heavy demand and the large number of cruises departing from and arriving in that area. All components contain data on hotels for the specific area to which the travel agency is primarily catering. With the exception of some large chains such as Holiday Inn, or Hilton, for example, hotel information is relatively localized and not as readily available in the remaining parts of the country. For example, travel agency $E$ which is located in Washington, has data on hotels in the Northeastern U.S., whereas agency $A$ in Los Angeles represents "places-to-stay" in California. All components contain national airline and train information.

It is important to note that an additional difficulty in finding a solution to the problem of achieving interoperability in FOTA and other similar federations stems from the conflicting nature of sharing and autonomy. On one hand, a travel agency would like to share information with other components of FOTA. On the other hand, the same component would also like to exercise some degree of control over the sharing process, e.g., control over the information it is willing to "export" to the other components. Since the focus of our work is on a solution to resolving semantic heterogeneity, several other important issues such as security, i.e., access control, and automatic update of shared data are not dealt with in their entirety here. In consequence, we assume that all the information stored in a specially "marked" section of a travel agent's database, called the *export schema*, is available to every other travel agency in the federation.

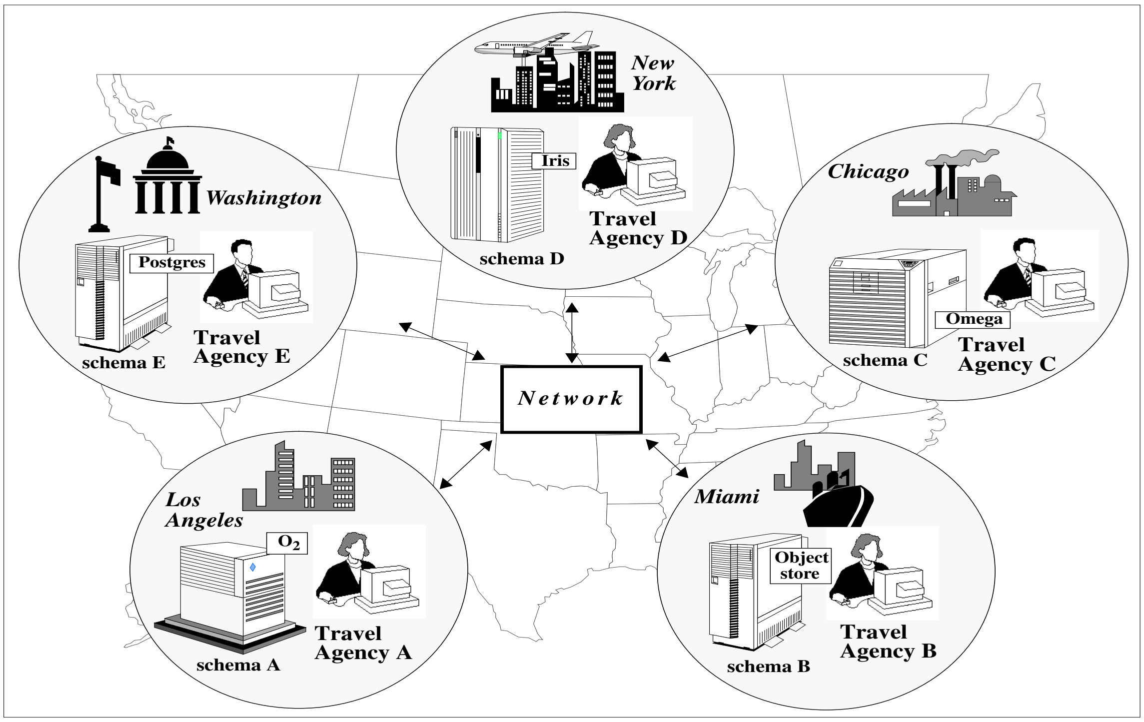


Figure 3.2: A loosely interconnected federation of travel agencies.

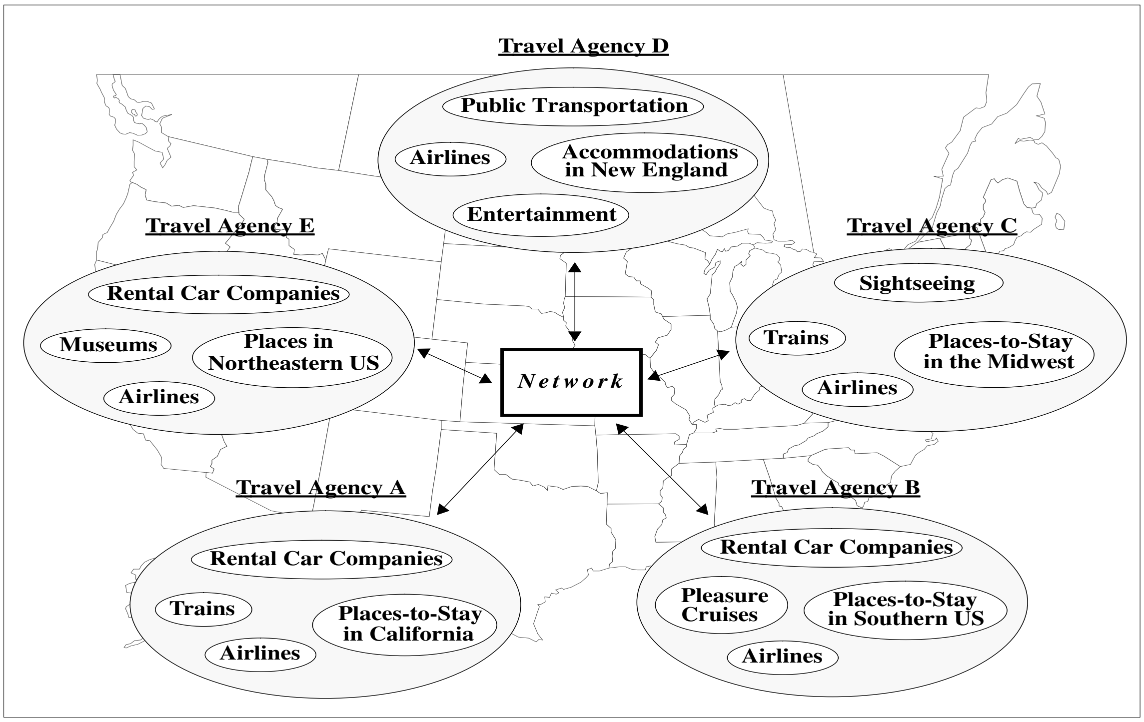


Figure 3.3: A conceptual overview of the federation and its components.

## 3.2 Semantic Heterogeneity in Federated Database Systems

One of the central problems of interoperability that must be addressed in order to support the sharing of information among a collection of autonomous, heterogeneous databases is *semantic heterogeneity*. By this we mean variations in the manner in which data is specified and structured in different components. Semantic heterogeneity is a natural consequence of the independent creation and evolution of autonomous databases which are tailored to the requirements of the application system they serve. For the remainder of this dissertation, we will term the problem of overcoming semantic heterogeneity in order to enable information sharing among autonomous, heterogeneous databases the *heterogeneity problem* in federated databases. Before we can present a solution to the heterogeneity problem, it is useful to examine the different kinds of semantic heterogeneity that may occur.

### 3.2.1 A Spectrum of Semantic Heterogeneity

Within the context of a federation of loosely coupled components, we can identify a spectrum of heterogeneity based on the following levels of abstraction:

1. Meta-data language (conceptual database model): The components may use a different collection of and techniques for combining the structures, constraints, and operations used to describe data. For example, different data models provide different structural primitives (e.g., records in the relational model versus objects in the object-based models) and operations (QUEL versus OSQL[1]) that contribute to semantic heterogeneity.

2. Meta-data specification (conceptual schema): While the components share a common meta-data language (conceptual database model), they may have independent specifications of their data (varied conceptual schemas). For example, this refers to the different schemas used by the members of FOTA.

[1] Object-based dialect of SQL, used as the DDL/DML in IRIS [19].

3. Object comparability: The components may agree upon a conceptual schema, or more generally agree upon common subparts of their schemas; however, there may be differences in the manner in which information facts are represented [33]. This variety of heterogeneity also relates how information objects are identified to the interpretation of atomic data values as denotations of information modeled in a database (e.g., naming). Looking ahead to travel agency $D$'s schema in Fig. 3.4, we can see that the type **Accommodations** represents information that is comparable to the information represented in $E$'s **Places in Northeastern U.S.**. Both types have associated with them information regarding the name and owner(s). However, schema $E$ does not contain information about ratings (missing information) and the location of an accommodation in schema is only implicitly contained in the type name, whereas it is explicitly stored in an attribute in schema $D$. Finally, a place in schema $E$ can be co-owned by several people but only one owner per accommodation is allowed in schema $D$ (mapping constraints).

4. Low-level data format: While the components agree at the model, schema, and object comparability levels, they may utilize different low-level representation techniques for atomic data values (e.g., units of measure). In terms of FOTA this refers to the problem that arises when values such as fare prices are represented in different denominations (U.S. dollars versus English pounds) or different precisions (16-bit float versus 8 bit integer) respectively (see Fig. 3.5).

5. Tool (database management system): The components may utilize different tools to manage and provide an interface to their data. This kind of heterogeneity may exist with or without the varieties described immediately above. In the context of FOTA, this kind of heterogeneity is due to the fact that components may use a different DBMS.

We assume that the components utilize a common language (see Section 5.1), thus ruling out the occurrence of heterogeneities of type (i), conceptual database model. Furthermore, tool heterogeneity, which is type (v) in the above spectrum, is treated as somewhat orthogonal to our concern in this paper. As a result, for the purpose of this research we term types (1) through (4) in the heterogeneity spectrum *semantic heterogeneity*.

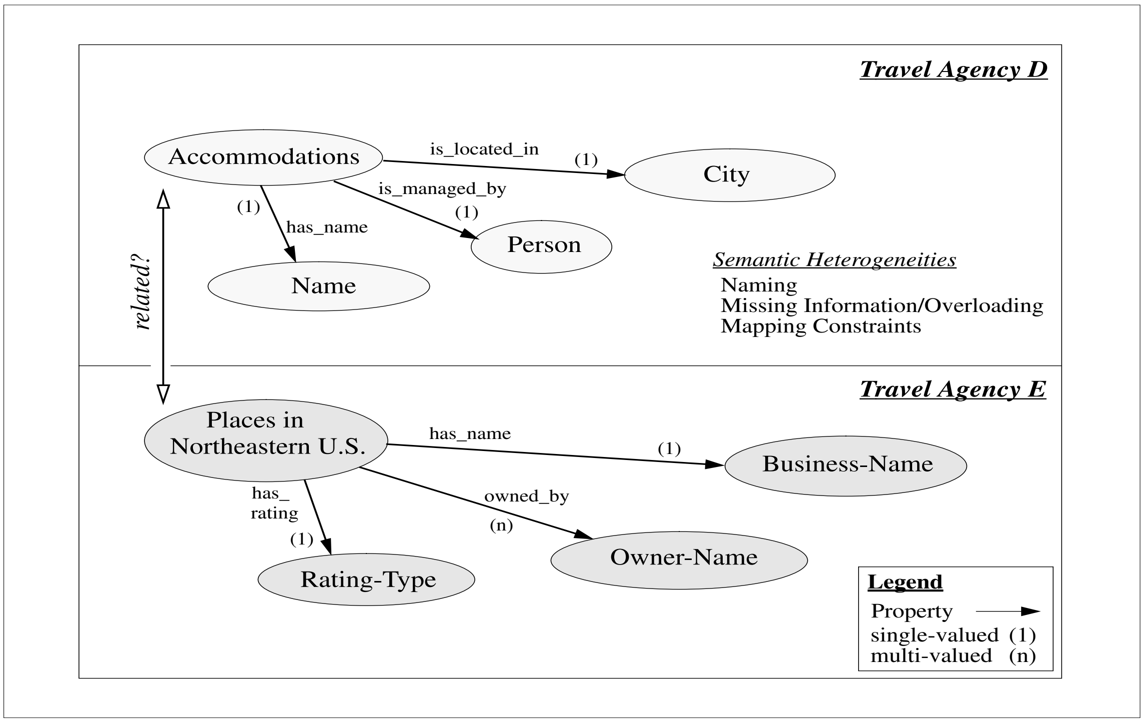


Figure 3.4: Semantic heterogeneity in FOTA: Example 1.

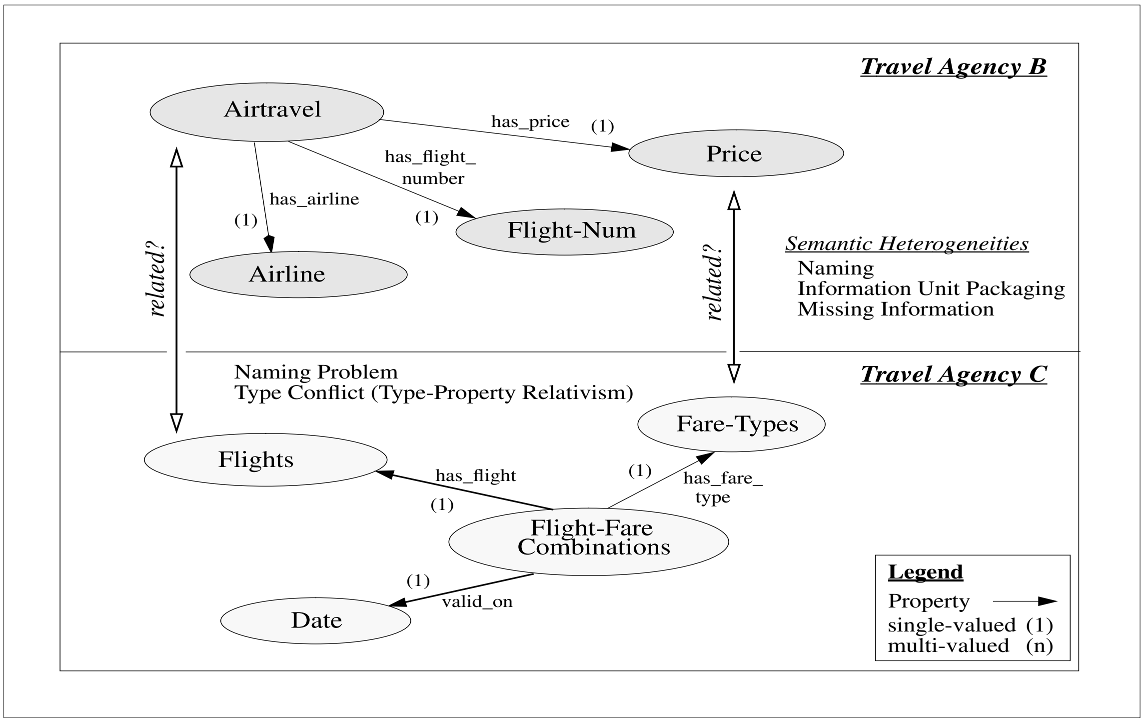


Figure 3.5: Semantic heterogeneity in FOTA: Example 2.

### 3.2.2 Causes for Schema Diversity

According to Batini et al. [5] there are three major causes for semantic heterogeneity:

1. Different perspectives:
   This is a modeling problem that finds its roots mostly during the design phase of a database schema. Different user groups or designers adopt their own viewpoints when modeling the same information. For instance, in example 1 in Fig. 3.4, different names were attached to the same concept (**Accommodations** versus **Places in the Northeastern U.S.**) in the two schemas of travel agencies $D$ and $E$, respectively.

2. Equivalent constructs:
   The rich set of constructs in data models allows for a large number of modeling possibilities, which results in variations in the conceptual database structure [33]. Typically, in conceptual models, several combinations of constructs can model the same real-world domain equivalently. In example 2 in Fig. 3.5, the association between flights and fare information was modeled as a function *has_price* in schema $A$ as opposed to a separate type **Flight-Fare-Combinations** connecting **Flights** and **Fare-Types** in schema $B$.

3. Incompatible design specifications:
   Different design specifications result in different schemas. In example 3 in Fig. 3.6, the relationship between **Travelers** and **Bookings** in schema $B$ indicates that a customer can only have one booking at a time, since the cardinality constraint $1 : n$ has been specified. The more realistic situation (that a customer may have several reservations at once) appears in schema $C$.

Thus far, we have discussed the nature of the interoperability problem and identified the causes and implications of semantic heterogeneity. In the following sections, we present the details of our mechanism for accomplishing interoperability between two components. In order to illustrate our solution we use the framework of FOTA.

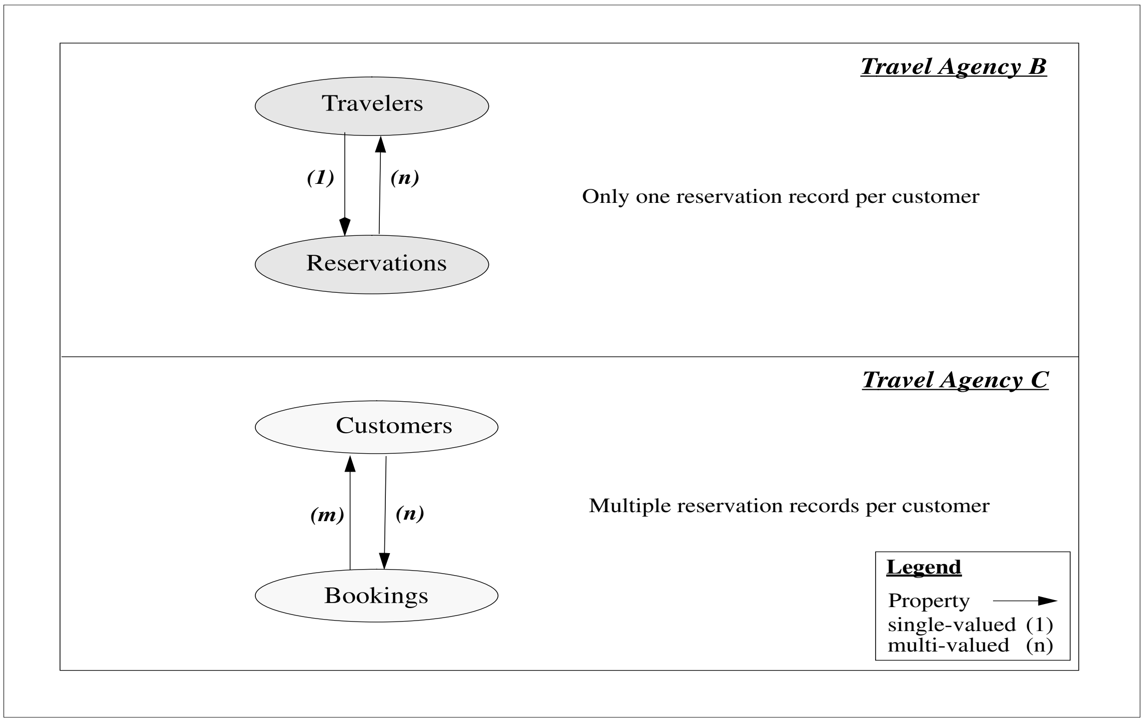


Figure 3.6: Semantic heterogeneity in FOTA: Example 3.

# Chapter 4

# A Perspective on Database Interoperability

Database interoperability refers to the ability to allow partial and controlled sharing of data among autonomous, heterogeneous database components [64]. Due to the complexity and difficulty in achieving interoperability in such systems, we have divided up this task into three subtasks which can be performed (iteratively) during different operational phases in the lifetime of a federation. The approach that we take in implementing each phase is interactive to the extent that the user (e.g., the user of a database component that is engaged in a sharing procedure) or the administrator of the federation has the ability to intervene at any time during each phase. In addition, human input is essential for resolving semantic heterogeneity.

## 4.1 Operational Phases

When a federation is initially formed or when a new component joins an existing federation, an intelligent *sharing advisor* must first locate all sharable information in each component and store the (partially) identified concepts in a separate repository called the *semantic dictionary*. In our approach, sharable information corresponds to all those data objects that a component is willing to *export* to the rest of the federation. Each component has a so-called *export schema* [27] that contains only

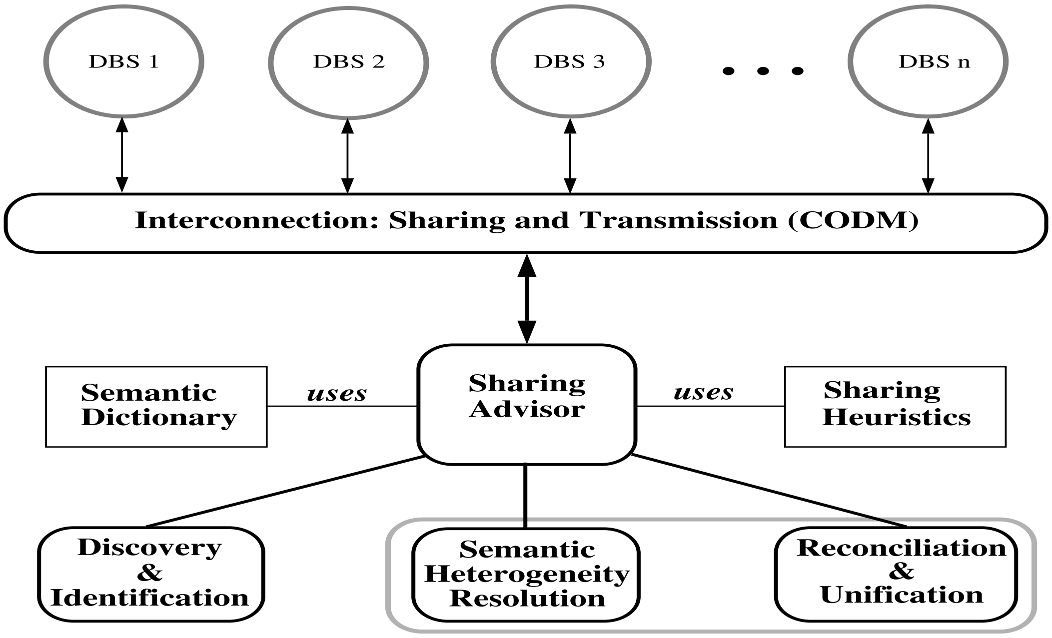


Figure 4.1: Interoperability in a federation of databases.

those objects that are non-private and thus sharable by other components. All other objects are kept in the component's *private schema*. This is a relatively simple way for each component to exercise control over its database but it suffices for our purpose since the focus here is not on security and access control mechanisms. The sharing advisor tool also responds to requests from database components to locate information that is in structure and content "relevant" to certain information in its own local schema. This first phase of locating and identifying non-local, relevant information is termed *resource discovery and identification* and currently under investigation in a related research project at USC (see [15]). During the second phase, the exact relationship between the requested non-local information and the local information in the schema of the integrating component is determined. This phase is termed *resolution of semantic heterogeneity* and is the focus of this dissertation. After the exact relationship between a foreign object and one or more local objects has been established in phase two, the third phase, called *sharing and transmission* supports efficient access to shared object(s). A picture of the operational phases is given in Fig. 4.1. In the next three sections, we will describe in more detail each of these three phases using the familiar example of FOTA.

## 4.2 Resource Discovery and Identification

A travel agency, for example, travel agency $A$ located in Los Angeles, needs information on bed & breakfast places in the New England area. $A$ only has bed & breakfast information for California and wants to incorporate this additional information into its own schema so that it can be accessed transparently like any other local information in $A$'s database. Assuming that $A$ has no prior knowledge of where to find bed & breakfast places in New England within the federation, it consults with the intelligent sharing advisor that assists components in discovering relevant, sharable information throughout the federation. The sharing advisor uses the information stored in the semantic dictionary and a set of heuristic rules (see Fig. 4.1); it returns to the component initiating the inquiry those concepts that it considers relevant to the requested information. In our case it locates two possible sources of bed & breakfast places in the New England area: the types **Lodgings**[1], **Inexpensive**, and **Expensive** of travel agency $E$ in Washington, and **Private Accommodations** of travel agency $D$ in New York. Fig. 4.2 displays the three (partial) schemas of the travel agencies involved in this identification process.

The goal of identifying relevant information is to retrieve information in other components that is *identical*, *similar*, or *related* to the requested information. Depending upon the amount and kinds of information in which a user is interested, s/he may wish for a large intersection or a small one. Large intersections correspond to concepts that are identical or similar to the concepts in his/her database (e.g., all components of FOTA are in the travel business, hence their databases contain many similar concepts such as **Accommodations** and **Hotels**, **Flights** and **Airtravel**, etc.). Small intersections correspond to related types of information (e.g., a travel agency and an automobile club share a few commonalities like **Travel** and **Trips**, hence their databases have a smaller overlap as far as data objects are concerned), or *completely disjoint* information (e.g., a travel agency and a telephone company will probably have no common concepts in their database schemas).

[1]Throughout the rest of this dissertation, the **boldface** type will be used for type objects, the *italicized* type will indicate function objects.

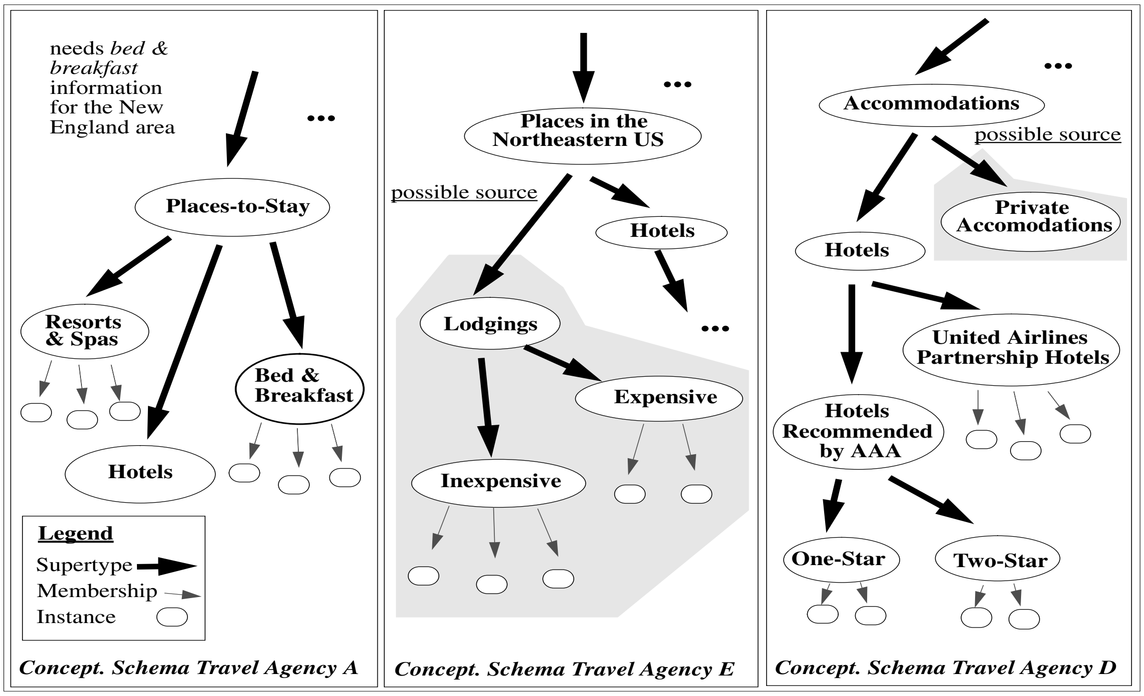


Figure 4.2: Three partial conceptual schemas for travel agencies $A$, $E$, and $D$.

## 4.3 Resolution of Semantic Heterogeneity

Having identified two possible sources of information, component $A$ has to find the exact relationship(s) between its own type **Bed & Breakfast**, the types **Lodgings**, **Inexpensive**, and **Expensive** of component $E$, and **Private Accommodations** of component $D$ in order to determine if and how they can be folded into its conceptual framework. This is the problem of resolving semantic heterogeneity as depicted in Fig. 4.1; it must be addressed before any sharing can take place. Using a *sharing language*, a *local dictionary* or *lexicon* as well as the semantic dictionary mentioned earlier, the meanings of concepts unknown to another component are "derived". Sharing language commands return structural information about an object (supertype, subtype(s), properties, etc.). The local lexicon, which is created and updated by each component separately, contains a semantic description of every sharable type object in the database. In order to make the local lexica usable throughout the entire federation, a common knowledge representation is used. Terms in question can be located and compared with each other. The semantic dictionary describes the relationships between terms in the local lexica.

## 4.4 Sharing and Transmission

In general, there is a spectrum of the kinds of inter-component sharing that may be desired. At one extreme, a *copy* of the foreign object can be created in the importing database. At the other, a surrogate (i.e., handle or place-holder) for the foreign object can exist in the local database. In the later case, objects such as instances, types, and functions are added to the importing schema at specially created places using local surrogates. Surrogates are essentially references that are used to refer to the original object in the exporting database whenever it is used by the importing database. Thus, local surrogates enable individual components to exchange and use each others' objects in a transparent way without making physical copies of the shared objects. The goal is to place the shared object into the best possible location in the existing schema of the importing component. In addition to sharing type objects with or without instances, it is also possible to share methods and individual instances by themselves. In order to share methods and instances, the importing component must already have the underlying meta-data (i.e. the type(s) that the method or instances belong to) in its schema. We will have more to say about the sharing of methods in Chapter 8.

Based upon this sharing mechanism the component that wishes to import the foreign concept will be able to add the (meta-)data representing the concept to its own local schema. Adding (meta-)data to an already existing schema is a two step process. First, conflicts (e.g., naming, modeling, scaling) between the objects of the importing database and the external objects from the other components must be resolved (conflict resolution). Second, these objects must be imported into the new schema as gracefully as possible and unified with already existing local types (unification). Fig. 4.3 shows the final schema of travel agency $A$ after conflict resolution and unification are completed. Finally, our sharing mechanism has to guarantee the transparent access to instances and functions that belong to the imported meta-data object and were imported at the same time.

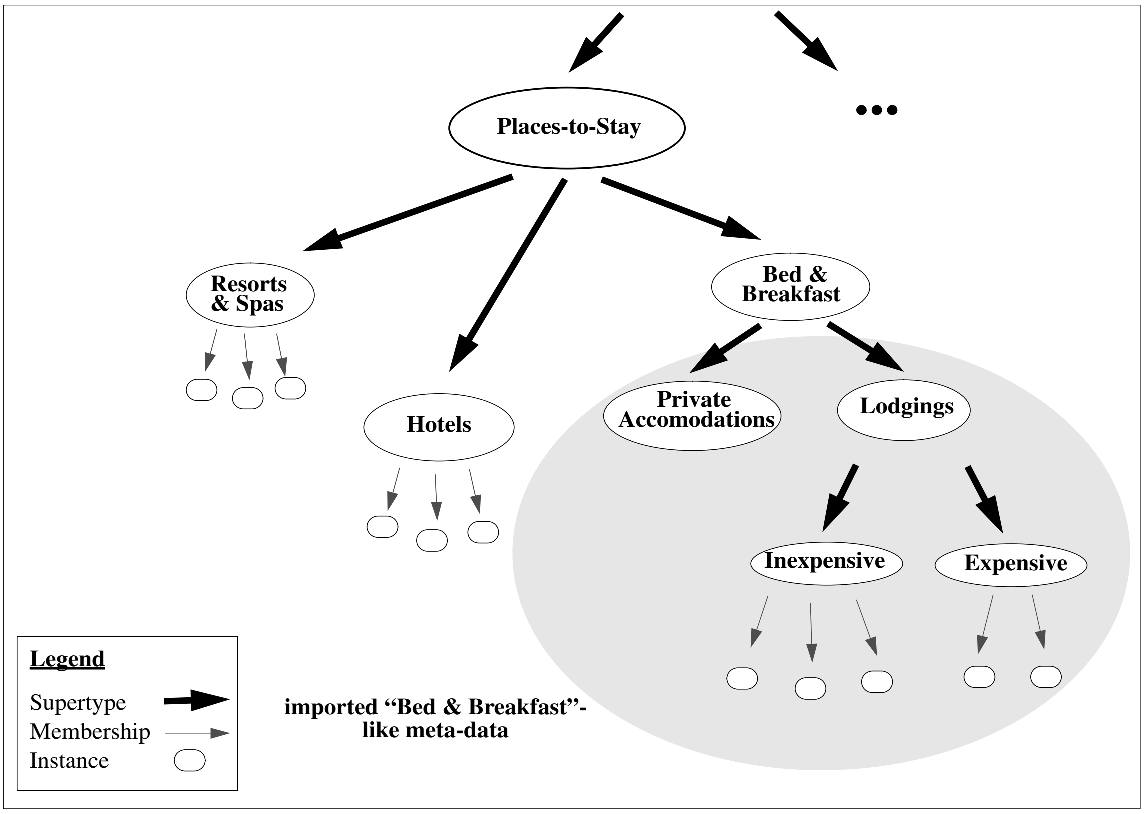


Figure 4.3: Component $A$'s conceptual schema augmented with remote information.

# Chapter 5

# The Object Database Model

In order for any collaboration to take place among the heterogeneous components of a federation, a common model for describing the sharable data must be established. This model must be semantically expressive enough to capture the intended meanings of conceptual schemas which may reflect several of the kinds of heterogeneity enumerated above. Further, this model must be simple enough so that it can be readily understood and implemented. The advantage of a simple model is that it can be implemented using a variety of already existing object-oriented database management systems saving both time and effort. To this end, we have chosen to use a Core Object Data Model (CODM) as the common data model for describing the structure, constraints, and operations for sharable data.

## 5.1 Core Object Data Model

CODM is a generic functional object data model, which supports the usual object-based constructs. In particular, it draws upon the essentials of functional database models, such as those proposed in in Daplex [66], Iris [19], and Omega [21]. CODM contains the basic features common to most semantic [1, 29] and object-oriented models [3], such as GemStone [49], $O2$ [41], and Orion [37]. The model supports complex objects (aggregation), type membership (classification), subtype to supertype relationships (generalization), inheritance of stored functions (attributes) from supertype to subtypes, and user-definable functions (methods). Not supported at this point are run-time binding of functions (method override), overloading of operations, constraints (semantic integrity rules) on types and functions, and remote transparency. We expect that CODM will eventually incorporate some of the concepts from [42, 47] for additional support in the unification process.

Some of the advantages of using an object-based common data model include the ability to encapsulate the functionality of shared objects [6], its extensible nature [43], and object uniformity [11]. This last item is especially important for the unification phase where one can ask about the equivalence of actual data-values, types, and operations.

## 5.2 Relationships Among Objects

Before we present the details of our approach to resolving semantic heterogeneity, we first present an overview of the various relationships that can exist among objects that model the same or similar real-world concepts in different components of a federation [5].

### 5.2.1 Common Concepts

As a direct result of the different causes for schema diversity described above, it may happen that the same concept of an application domain is modeled by different representations $R_1$ and $R_2$ in different schemas. Returning to the example in Fig. 4.2 of Sec. 4, we can see that the concept of "bed & breakfast" as it can be found in travel agency $A$'s schema is also represented by the type **Lodgings** in travel agency $E$ and by the type **Private Accommodations** in travel agency $D$. In addition to the obvious naming differences, both abstract objects mirror closely related real-world information but use different modeling constructs in their representations. Several types of *semantic relationships* can exist between two representations $R_1$ and $R_2$: they may be identical, equivalent, agreeable, or incompatible:

1. Identical:
   $R_1$ and $R_2$ are exactly the same. This happens when the same modeling constructs are used, the same perceptions are applied, and no extraneous information enters into the specification. For example, **Reservations** in schema $B$ and **Bookings** in schema $D$ are equal representations for the same real world concept, namely records of services rendered by the travel agency to its customers.

2. Equivalent:
   $R_1$ and $R_2$ are not exactly the same because different modeling constructs have been applied. For example, in Fig. 3.4, the types **Accommodations** in travel agency $D$, and **Places in Northeastern U.S.** in travel agency $E$ can be used to model the same information, namely hotels in the Northeastern U.S. of the U.S., but the representation is different. In $D$'s schema the location is explicitly modeled through a separate function called *is_located_in*. In $E$'s schema, the location information is implicit since it is part of the type name.

3. Compatible:
   $R_1$ and $R_2$ are neither identical nor equivalent. However, their representation is not contradictory. For example, **Private Accommodations** and **Resorts** both model the same basic hotel information (i.e., rooms to rent) but differ in the type of services offered.

4. Incompatible:
   $R_1$ and $R_2$ are contradictory because of inconsistent design specifications or fundamental differences in the underlying information. For example, in Fig. 3.6, the two partial schemas displaying **Travelers** and **Reservations** and **Customers** and **Bookings** are incompatible because of the cardinalities assigned to their respective relationships.

### 5.2.2 Related Concepts

In addition to common concepts, related concepts arise frequently; we can enumerate the following most commonly used types of interschema (binary) relationships of this kind:

1. Generalization/Specialization:
   Generalization is the result of taking the union of two or more types to produce a higher-level type. In terms of the travel agency example, **Places-to-Stay** of schema $A$ is the generalization of **Resorts & Spas**, **Hotels**, and **Bed & Breakfast**. Specialization is the opposite of generalization.

2. Positive Association:
   It is impossible to accurately classify all kinds of relationships that can exist between objects. This category includes concepts that are "synonyms" in some context (e.g., **Bed & Breakfast** and **Private Accommodations**), and those that are typically used in the same context (e.g., **Hotel** and **Reservations**).

This list is by no means exhaustive but rather indicative of useful inter-relationships vis-a-vis semantic heterogeneity resolution.

# Chapter 6

# A Mechanism for Semantic Heterogeneity Resolution

The fundamental goal of this research is to automate the resolution of semantic heterogeneity when sharing information among components in a federation. In general, sharing is possible at many different levels of abstraction and granularity, ranging from specific information units (data objects), to meta-data[1], to behavior. For this dissertation, we term the sharing of type objects *type-level* sharing. Sharing of individual instances is termed *instance-level* sharing and sharing of behavior or functions is termed *function-level* sharing. Type-level sharing forms the foundation for instance- and function-level sharing since when sharing instances or functions, the underlying type(s) has to exist in the remote component. Furthermore, instance and function-level sharing requires more work than type-level sharing in the sense that there may be problems with side-effects[2]. Information sharing is investigated in detail in Chapters 7 and 8.

[1]This includes structural schema specifications and semantic integrity constraints.

[2]By side-effects we really mean: (1) any kind of implicit input other than the input argument, and (2) any modifications to the state of the database where the function executes.

## 6.1 Earlier Approaches

As observed in Sec. 2, most of the previous work on resolving semantic heterogeneity has concentrated in the area of determining *structural equivalence* or *behavioral equivalence* at the schema level. In what follows, we first present a brief overview of these two approaches (Secs. 6.1.1 and 6.1.2). In Section 6.2 we describe our mechanism; it is based on three parts, each addressing a different aspect of the problem of establishing the relationship between type objects in different components.

### 6.1.1 Basic Structural Equivalence

A widely used approach for disambiguating two objects[3] is to determine if they are structurally equivalent (at some level of abstraction). As an example, consider a (local) object, say `:joachim`, of type **Traveler** in travel agency $B$'s database and a foreign object, say `:dennis`, of type **Customer** in travel agency $C$'s database, that are being compared. In this case, where both objects are atomic, the comparison is relatively easy. One can simply apply some sort of "eq" or "equal" semantics. Otherwise, further comparisons are needed. When comparing type objects this includes comparing the type names, the instances of that type, the subtypes of the type, and so forth. In addition, function information such as value types, function names, missing functions, and mapping constraints, for example, can prove useful on structural characteristics. Consider the two abstract objects **Accommodations** of travel agency $D$ and **Places in Northeastern U.S.** of travel agency $E$ as shown in Fig. 3.4. In this case, the functions *has_name* with value type **Name** and *is_managed_by* with value type **Person** in $D$'s schema and *has_name* with value type **Business-Name** and *owned_by* with value type **Owner-Name** in $E$'s schema suggest a certain similarity between the two concepts. In order to determine a more exact connection, one has to make further comparisons. The more commonalities there are with respect to the above criteria the higher the correlation between **Accommodations** and **Places in Northeastern U.S.**.

[3]In this section the term object can refer to a type object as well as an instance object.

### 6.1.2 Basic Behavioral Equivalence

Another approach to determining object equivalence is to look at the operations (i.e., methods) that are defined on the objects in order to establish a behavioral equivalence between them. The general idea behind this approach is to apply all operations that are associated with a given local object, say **Cities**, and compare the results with those obtained when running the same methods on a foreign object, say **Destinations**. Examples of such methods are *sights()*, which takes a city instance as input and returns its sights, or *location()*, which returns the country in which a particular city is located. Behavioral equivalence requires the ability to compare methods and their results when executed in different environments. Thus, its success largely depends on how well a particular federation environment supports the remote execution of procedures. Another problem that remains in this approach is that of deciding when the results of the procedure applications are equal.

## 6.2 A Three-Pronged Approach to Relative Object Equivalence

In order to determine the relationship between objects within a broader context, we realize that not one single method (such as the structural approach) but a combination of several different approaches taken together is highly promising. While it is nearly impossible to completely automate such a procedure, the following mechanism provides substantially useful functionality. We now describe our three-pronged approach for resolving semantic heterogeneity, viz. determining relative object equivalence.

### 6.2.1 Remote Sharing Language

The remote sharing language (*RSL*) is part of CODM and provides a standardized interface to the conceptual schemas of the participating components. *RSL* provides the capabilities to (1) query the meta-data of selected databases in order to obtain structural information about objects (resolution), and (2) augment selected database schemas with remote objects (unification).

| RSL Command | Description |
|---|---|
| *ShowMeta* | Returns a list of all sharable types |
| *HasProperties* | Returns a list of stored functions |
| *HasValueType* | Returns the value type for a stored function |
| *HasMethods* | Returns a list of computed functions |
| *HasInstances* | Returns a list of instance OID's |
| *HasValue* | Returns the value of a stored function |
| *HasDirectSubtypes* | Returns a list of all direct subtypes of a type |
| *HasDirectSuperType* | Returns the direct supertype of a type |
| *ImportMeta* | Imports meta-data |
| *ImportInstance* | Imports an instance object |
| *ImportMethod* | Imports a computed function |

Table 6.1: Remote sharing language commands.

Table 6.1 shows a list of *RSL* commands. See App. A for a more detailed description of how these commands are used. Note that we cannot depend on examining supertypes of an object, since the supertype might not be available for sharing; however, we always assume that subtypes are. Since a subtype is a specialization of a type, it makes sense to allow all subtypes of a sharable type to be sharable as well. Also remember that the term *stored function* in our data model refers to an attribute denoting an inter-type relationship. Components that wish to share and exchange information must agree on a common interface (CODM) that can provide the functionality described above.

### 6.2.2 Local Lexicon

One of the foundations for our approach is to be able to maintain semantic information about all the sharable objects in each component beyond the information that is already provided by the underlying schema. For this purpose, each component database system is augmented by a *local lexicon* where it defines all type objects it is willing to share with the other components. The common vocabulary in which shared knowledge is represented in a lexicon draws some ideas from declarative knowledge representation forms such as the *Knowledge Representation Language* (KRL) [8], semantic networks [59], and the *Cyc* knowledge base [28]. In

| R-Descriptor | Meaning |
|---|---|
| IDENTICAL | Two types are the same |
| EQUAL | Two types are equivalent |
| COMPATIBLE | Two type are transformable |
| KINDOF | Specialization of a type |
| ASSOC | Positive association between two types |
| COLLECTIONOF | Collection of related types |
| INSTANCEOF | Instance of a type |
| COMMON | Common property of a collection |
| FEATURE | Descriptive elaboration |
| HAS | Property belonging to all instances of a type |

Table 6.2: Relationship descriptors.

our approach, knowledge is represented in a local lexicon as a static collection of facts of the simple form:

`<term>` **relationship-descriptor** `<term>`

A term on the left side of a relationship descriptor represents the unknown concept which is described by the term on the right side of a relationship descriptor. The set of descriptors is extensible and specifies the relationships that exist between the two terms. Table 6.2 shows a list of conceptual relationship descriptors utilized in our mechanism.

The terms that are used to describe the unknown concept are taken from a dynamic list that characterizes commonalities in a federation. This dynamic list of commonly understood terms is called *ontology*. According to Gruber [22], "[An] ontology is a description of the concepts and relationships that can exist for a [component] or a [federation of components] for the purpose of enabling knowledge sharing and reuse." Since interoperability only makes sense among components that model similar or related information, it is reasonable to expect a common understanding of a minimal set of concepts taken from the universe of discourse. Thus, in this research, an ontology defines the vocabulary with which information and assertions are exchanged among components. For example, a possible ontology for a federation consisting of collaborating travel agencies is different from the ontology used in a federation of cooperating biologists. Ontologies are organized in "packages" depending on the application domain they describe. Providing a federation with the appropriate ontology package beforehand will speed up the initial set-up time. Part of each federation is therefore an initial ontology package that consists of a *general-purpose* ontology (GPO) and one or more *special-purpose* ontologies (SPOs). The GPO is application independent and constitutes the minimal vocabulary that forms the basis for any kind of inter-component communication. SPOs on the other hand are application dependent and contain terms taken from a specific topic. By organizing ontologies based on their contents, they can easily be reused and shared by other federations. Both the GPO and SPOs are highly dynamic, meaning that the number of terms in them grows during the life-time of the federation depending on the sharing patterns of the participating components. Both the GPO and SPO's together provide the components with a vocabulary that describes the application areas of the database systems involved. As a result, our resolution mechanism works best if all the components have a similar background, thus reducing the need for many additional SPO's or SPO's with a lot of terms.

In the case of FOTA, a subset of the ontology package could be:

$$\mathcal{GPO} = \{\textit{Person, Name, Number, Owner, Thing}, \ldots\}$$
$$\mathcal{SPO}_{Travel} = \{\textit{Arrival, Booking, Customer, Departure, Trip}, \ldots\}$$
$$\mathcal{SPO}_{Travel-Air} = \{\textit{Airline, Fare, Flight}, \ldots\}$$
$$\mathcal{SPO}_{Travel-Hotel} = \{\textit{Category, One-Star, Two-Star, Three-Star, Hotel, Location}, \ldots\}$$
$$\mathcal{SPO}_{Travel-Cruise} = \{\textit{Boat, Entertainment, Sightseeing}, \ldots\}$$

Specifically, $SPO_{Travel}$ is a special-purpose ontology containing terms used in the travel agency business. $SPO_{Travel-Air}$, $SPO_{Travel-Hotel}$, and $SPO_{Travel-Cruise}$ contain terms that are used mostly in the context of air travel, hotel reservations, and pleasure cruises respectively. Using this package in FOTA, we may have the following in a local lexicon: The underlying idea of the lexicon is to represent the real-world meaning of all shared terms in order to complement the results of the *RSL* commands. In some cases, it is not possible to "derive" the meaning of a term only by looking at its structure (for example, **Pleasure Cruises** and **Flights** both have functions containing departure and arrival information but the types themselves are not related). By the same token, the real-world meaning by itself might not be enough to correctly integrate a term into another type hierarchy (for example, two

| Local Term | Relationship Descriptor | Ontology Term | Textual Description (not part of lexicon) |
|---|---|---|---|
| **Bed & Breakfast** | KINDOF | **Hotel** | A bed & breakfast place is a specialization of hotel and one of its features is its low price. |
| ” | FEATURE | **economical** | |
| **Sightseeing** | KINDOF | **Entertainment** | Sightseeing is a special form of entertainment that is a collection of trips to places and is meant to be enjoyable. |
| ” | COLLECTIONOF | **Trips** | |
| ” | HAS | **Place-to-visit** | |
| ” | FEATURE | **Pleasure** | |
| **MexicoFlights** | COLLECTIONOF | **Flight** | MexicoFlights is a collection of flights with the common destination of Mexico. |
| ” | COMMON | **Destination** | |
| ... | ... | ... | ... |

Table 6.3: Contents of a local lexicon.

types **Fare** and **Airfare** that represent the cost of tickets but store their prices in different denominations cannot simply be merged into a new type). Thus, by using the *RSL* together with local lexica, we are able to achieve a higher degree of confidence in the correctness of our mechanism when integrating objects. An example of the partial contents of three local lexica is given in Fig. 6.1.

### 6.2.3 Semantic Dictionary

Whereas local lexica contain semantic descriptions about local, sharable objects, they do not contain any knowledge about relationships between the entries in different lexica. This information is collected in a global repository, called *semantic dictionary*. Like the local lexica, the semantic dictionary is dynamic, meaning that its content is updated whenever new or additional information becomes available (e.g., after a relationship between two similar remote types has been established).

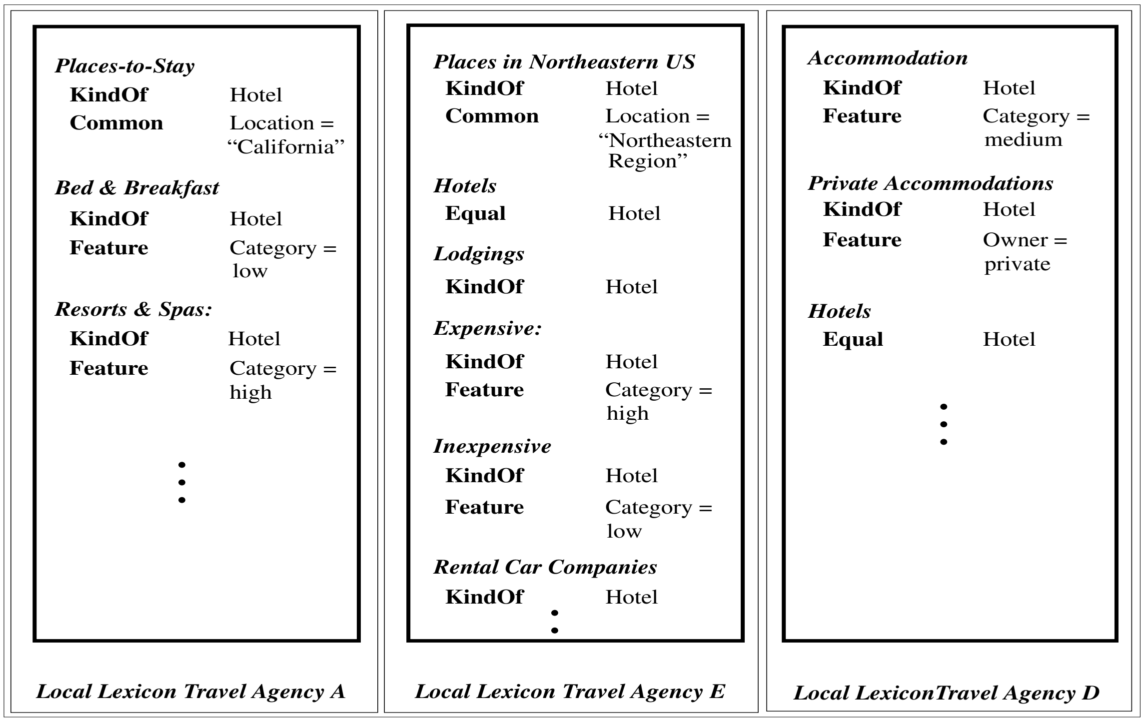


Figure 6.1: A partial view of three local lexica.

In specific, related types from different components that have been identified are grouped into a collection called a *concept*, within which subcollections called *subconcepts* can be further identified. This generates a *concept hierarchy* much like the one shown in the bottom of Fig. 6.4. A concept, shown as a rectangular box (note that ovals are reserved for type objects) represents a global view of its members (which are type objects from the component schemas) and the members of all its subconcepts. Initially, the concept hierarchy consists of key concepts from the ontology package only. During the life-time of the federation this hierarchy will grow as sharable objects and their relationships to these key concepts are identified. In a sense, the semantic dictionary with its concept hierarchy represents a dynamic federated knowledge base about the different sharing patterns in the federation. The larger the concept hierarchy, the greater the probability that a relationship between two objects has been previously resolved. In these cases, the resolution process is reduced to a look-up of previously established object relationships, similar to the problem solving approach applied in Case-Based Reasoning (CBR) techniques (see [12]).

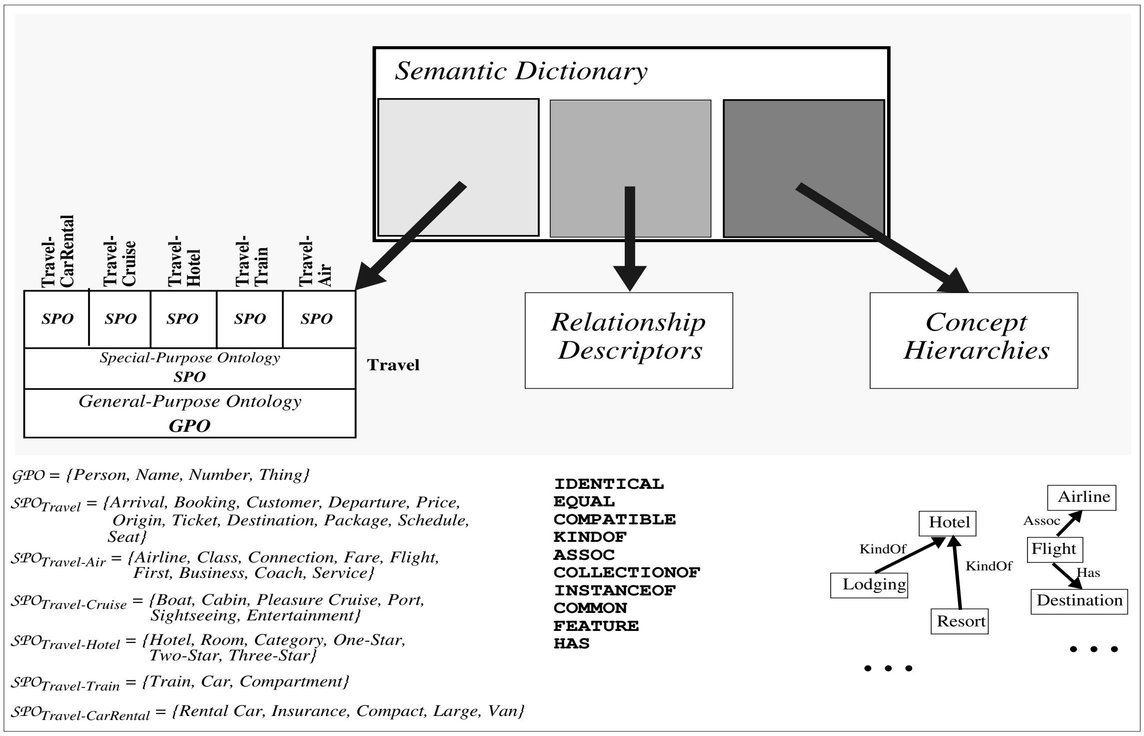


Figure 6.2: The semantic dictionary.

In addition to the concept hierarchy of sharable objects, the semantic dictionary also contains the ontology package described earlier as well as a list of the relationship descriptors that are used in the local lexica. A schematic overview of a semantic dictionary is given in Fig. 6.2.

## 6.3 Resolving Object Relationships

One reason for the relatively slow progress in the area of semantic heterogeneity resolution is that heterogeneous databases provide a wide spectrum of vocabulary and name usage which is inherently difficult to "understand" by computers. The narrower the domain, i.e., the higher the degree of redundancy in the vocabularies and the closer the relationships among objects, the higher the chances of successful resolution. Currently, the process of disambiguation (viz., solving semantic diversity) is still only partially automated and requires human interaction along the way.

In order to determine the relationship between objects within a broader context, we realize that not one single method (e.g., schema resolution based on structural

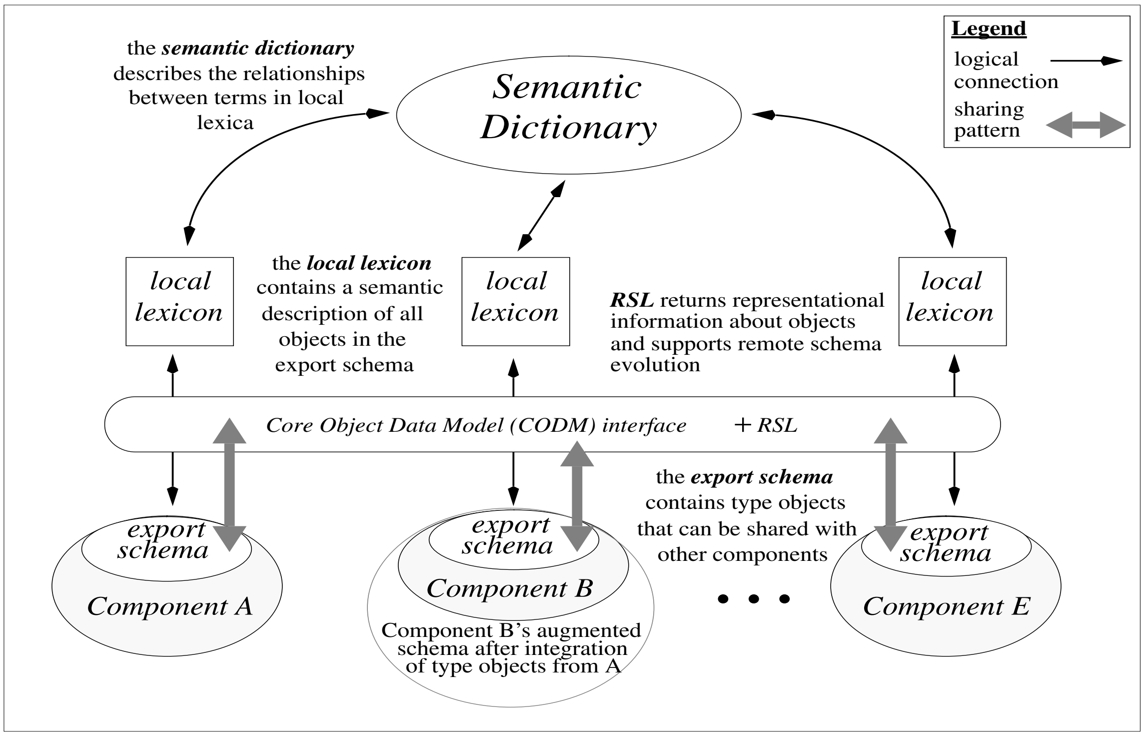


Figure 6.3: Sharing architecture and the various interactions among its components.

knowledge, see [13]) but a combination of several different approaches taken together is highly promising. This also becomes evident when studying the different types of semantic heterogeneities displayed in Figs. 3.4 through 3.6. Therefore, we decided to augment the conceptual schemas of participating components with additional semantic information describing the usage of sharable objects in their application environment. While it is nearly impossible to completely automate the resolution of inter-object relationships, the following mechanism provides substantially useful functionality. We now describe our approach for resolving semantic heterogeneity, viz., determining relative object equivalence.

Using the *RSL*, local lexica as well as the semantic dictionary mentioned earlier, the meanings of concepts unknown to another component are “derived”. *RSL* commands return structural information about an object (supertype, subtype(s), properties, etc.) and perform the unification of remote objects with local schemas. The local lexicon, which is created and updated by each component separately, contains a semantic description of every sharable type object in the database. In order to make the local lexica usable throughout the entire federation, a common knowledge representation is used. Terms in question can be located and compared with each other. The semantic dictionary describes the relationships between terms in the local lexica. See Fig. 6.3 for a pictorial description of the interactions among these components. In order to provide components with a way to protect their private data from the rest of the federation, sharable objects must be placed in a special section of the conceptual schema termed *export schema*. Everything that is not part of the export schema is invisible to the rest of the federation.

The basic problem addressed by the semantic heterogeneity resolution mechanism may be expressed, without loss of generality, as: given two objects, a local and a foreign one, return the relationship that exists between the two. Specifically, our strategy is based on structural knowledge (conceptual schema) and the (known) relationships that exist between keywords and the two objects in questions (local lexicon, semantic dictionary). One characteristic of our approach is that the majority of user input occurs before the resolution step is performed (i.e., when selecting the set of keywords and creating the local lexicon) rather than during. In Fig. 6.4 we can see a pictorial description of parts of two local lexica belonging to schema $A$ and schema $B$ respectively. In the same picture we can also see a subpart of the concept hierarchy in the semantic dictionary. Looking at $A$'s lexicon, we can see that the property *discovered_by* is equivalent to the concept **Author** from the ontology package. We also see that schema $B$'s **Researcher** and *Name* combination is equivalent to the concept **Author**. Therefore, *discovered_by* and the type property combination of **Researcher** and *Name* must be equivalent.

We can make the following observation on the use of *RSL* commands with respect to resolution of semantic heterogeneity. All information about the structure of a type object is provided through selected *RSL* commands: an approach that can be viewed as a variation to the usual paradigm of behavior encapsulation in object-oriented programming languages. Rather than encapsulating behavior, these commands "encapsulate" the structure of a type object. The advantage of this approach is that *RSL* commands that are essentially computed or foreign functions in our data model can be part of each component's schema without modifications to the underlying architecture.

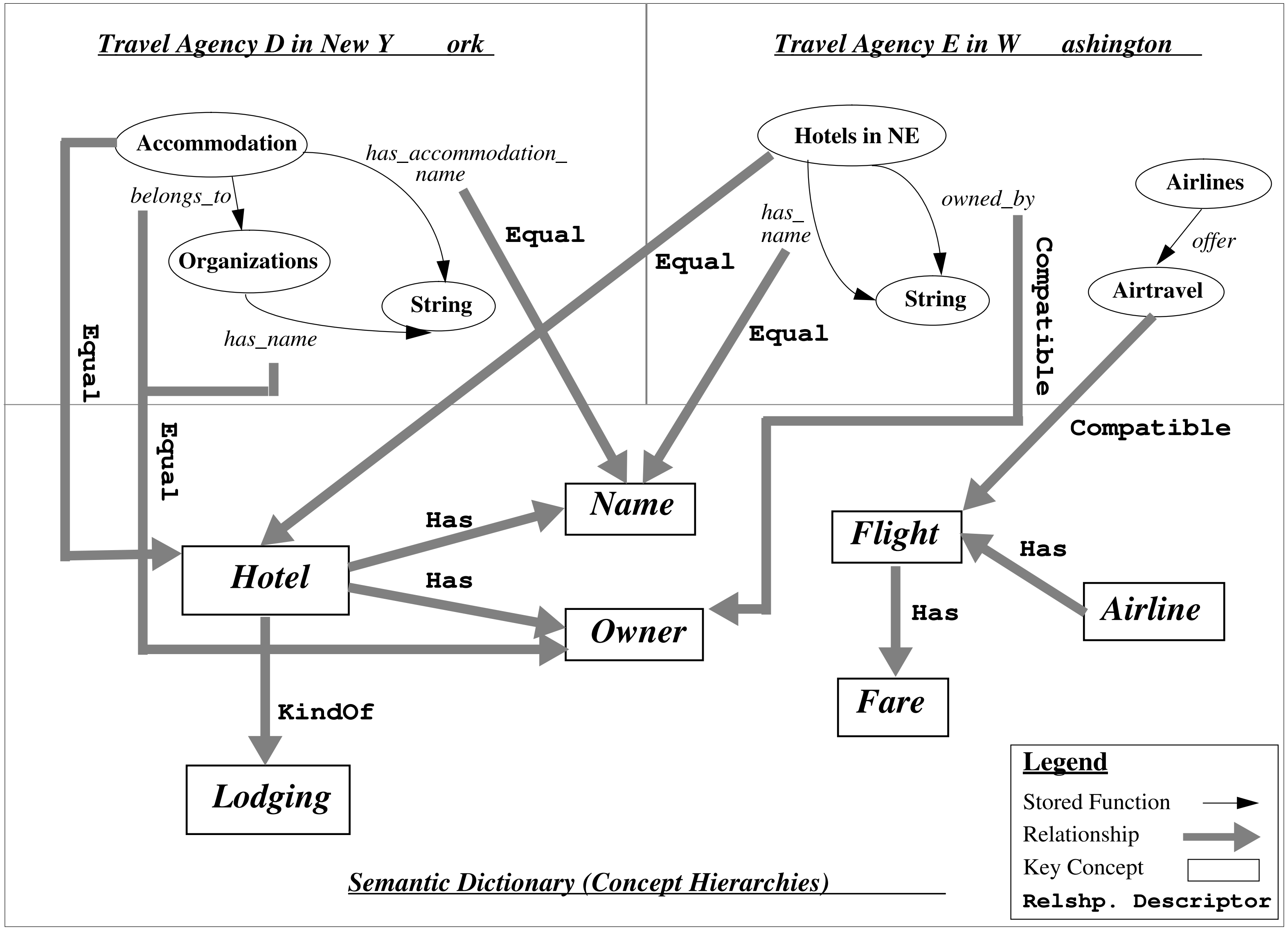


Figure 6.4: Resolution of object relationships.

# Chapter 7

# Unification of Remote and Local Information

Thus far we have discussed the nature of the schema integration problem and identified the various causes of and a solution to the heterogeneity problem. In this section we describe the individual activities that a component must go through when importing a type object. As mentioned before, there are two steps that must be performed when adding imported (meta-)data to an already existing schema. First, once conflicts (e.g., naming, modeling, scaling) between a source and a target schema are detected, these conflicts must be resolved so that the unification of the foreign type(s) with the corresponding local objects is possible (conflict resolution). Second, the foreign object(s) must be imported into the local schema as gracefully and naturally as possible (unification). In the context of this research, "schema" refers to a collection of type objects that are connected through the usual object-based constructs. As noted above, we consider here only the importation of type objects or collections thereof.

## 7.1 Eliminating Inconsistencies

The goal of this activity is to resolve inconsistencies between the imported type(s) and the target schema before the unification step. However, automatic conflict resolution is in general infeasible. Sometimes conflicts cannot be resolved because they arose as a result of some basic inconsistencies. In cases where automatic resolution is not possible, the conflicts are reported to the users who must guide the unification mechanism in the process. The following specific activities are performed during conflict resolution:

- Operations on atomic data values: This is an attempt to resolve type (iv) heterogeneities (low-level data format. For example, a conversion of fare prices that are represented in English pounds into U.S. dollars (this might also affect the respective value types of functions associated with fare prices).

- Renaming: This activity addresses the problem of *homonyms* (where the same name is used for two different concepts) and *synonyms* (in which the same concept is described by two or more names). As an example for homonyms, consider the following scenario. Two components $A$ and $B$ of FOTA want to share fare information. Both schemas include an object named **Fare-Prices**. However, **Fare-Prices** in schema $A$ includes airport and sales taxes whereas in schema $B$ it represents the "true" fare price without any taxes added. It is obvious that merging the two types would result in a problem. Therefore, one of the two types must be renamed before unification can take place in order to reflect the differences in their representations. As an example of synonyms, consider two schemas representing customer information, **Clients** and **Customers**, where both types contain the same data. In this case, keeping two distinct types in the integrated schema would result in modeling a single object by means of two different types.

Additional conflicts are resolved during the following unification phase.

## 7.2 Unification

At this point, the foreign object(s) can be unified with the corresponding local object(s). If necessary the target schema must be restructured to achieve a result that is (1) complete, (2) minimal, and (3) understandable. Complete, since the new, integrated schema must contain all concepts that were present before the unification process took place. Minimal, since concepts should only be represented once, and understandable, since the integrated schema should be easy to understand for the end user.

Upon importing the (meta-)data, structural conflicts with existing types in the component's local type hierarchy may arise. Several possibilities exist, and we demonstrate our approach with the help of several sample scenarios from FOTA. These scenarios differ in the complexity of the schemas to be integrated, starting with the most simple one: the importation of a single foreign type object. The second scenario examines the cases when a component wishes to import one or more types that are inter-related. The types to be unified are usually part of a more complex type hierarchy, hence we also need to be concerned with inheritance issues. The last scenario is a special case of the previous one and focuses on the relationship(s) between the objects to be unified. In this scenario, it is possible to construct the situation in which a component wishes to import objects that are connected in a complex manner involving one or more types not present in the local schema. In assembling this framework, we want to point out the relevance of previous work in this area described in [5, 30, 32, 53].

Below, we assume the following naming convention. Let $C_{imp}$ denote an importing component of FOTA, $C_{exp}$ denote an exporting component of FOTA, $L$ denote a local object belonging to $C_{imp}$, and $F$ denote a foreign object belonging to $C_{exp}$.

1. In the first scenario, we assume that a component $C_{imp}$ wishes to import a single type object $F$ from component $C_{exp}$. In increasing order of generality, this scenario can be decomposed into the following three cases.

   (a) *$F$ does not exist in $C_{imp}$'s schema.* For example, travel agency $D$ wishes to import the object type **Museums** from travel agency $E$. Since **Museums** does not exist in $D$'s schema, it can be added without further modifications. The designated place for the new type is either as a subtype of "root" or "system" (i.e., the highest place in the type hierarchy), or as a subtype of some other user-specified local type. It is important to note that "adding" a type requires some additional work in the sense that the value types for the associated stored and/or computed functions do not always exist in the target schema and may have to be imported as well. The exact details on how to import functions is described Chapter 8.

(b) *$F$ is (semantically) equivalent to some local object $L$* in *$C_{imp}$'s schema.* In this case we make $F$ a subtype of $L$ and add the necessary functions to both $L$ as well as $F$. For example, travel agency $A$ wishes to import the object type **Private Accommodations** from travel agency $D$ (this scenario is depicted in Fig. 4.2). An equivalent type, namely **Bed & Breakfast** already exists in $A$'s schema. Thus, the imported type **Private Accommodations** is added as a subtype of **Bed & Breakfast** and all its functions are created for both the supertype as well as the subtype. Note that "subsetting" is a unification practice that is used by most methodologies [5]. In fact, it is considered to be the basis for accommodating multiple user perspectives on comparable types.

The case where $F$ is exactly identical to $L$ (e.g., structurally as well as semantically), is merely a simplification and only requires the importation of the type instances in which $C_{imp}$ is interested (this is essentially a type merge as mentioned earlier).

(c) *$F$ is related to an object $L$ in $C_{imp}$'s schema.* In the case when $L$ and $F$ are similar in their semantic meaning but not equivalent, i.e., they are identical with respect to some functions and different with respect to others, a new supertype is created that contains only the identical functions of $L$ and $F$. The functions in which $L$ and $F$ differ are associated with two new subtypes which inherit the functions common to both $L$ and $F$. Together, the new supertype and its two subtypes contain the same information as $L$ and $F$ before the unification. This method was proposed in [13]. In terms of FOTA, the following example as shown in Fig. 7.1 best illustrates this method. Assume that travel agency $D$ is about to integrate

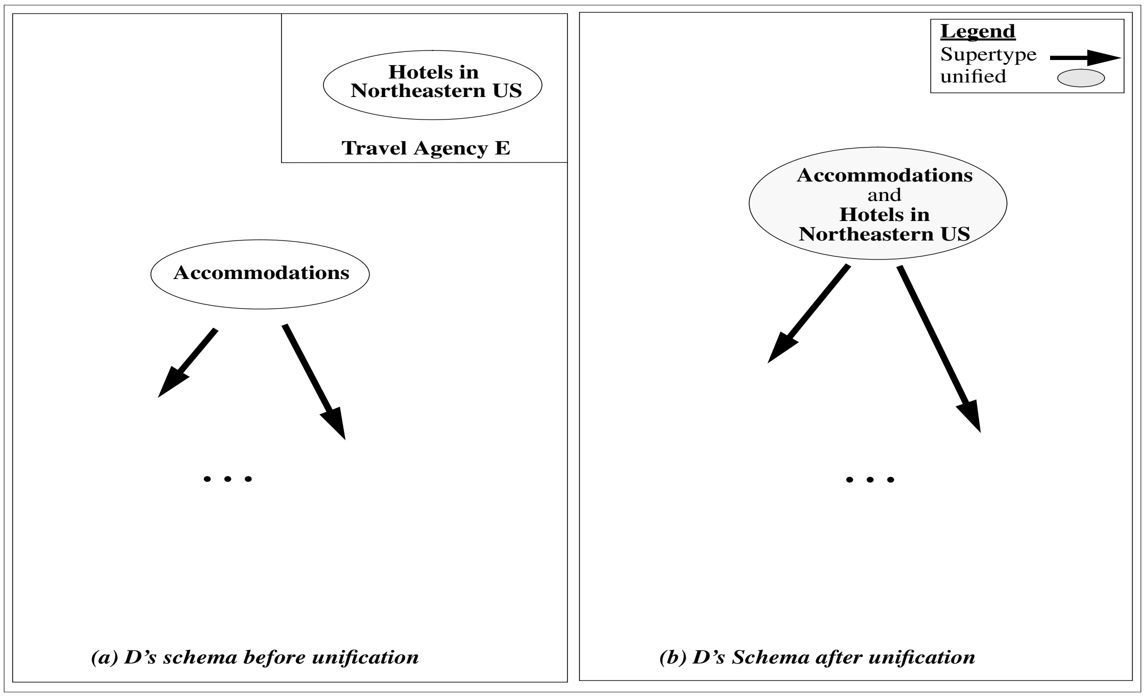


Figure 7.1: Unification of two related objects.

**Hotels in Northeastern U.S.** into its own schema. It has determined that **Hotels in the Northeastern U.S.** is related to its own **Accommodations in New England**. Although there is considerable overlap between the two types (e.g., both contain hotel information, both cover a similar geographic area), there are still enough differences to prevent $D$ from simply "merging" the two types. Using the above method, a new supertype **Accommodations** is created that has two subtype **Northeastern U.S.** and **New-England**. Together, all three types contain the same information as the two original types.

2. In the second scenario $C_{imp}$ wishes to import (meta-)data consisting of several, inter-related objects from $C_{exp}$. For simplicity, we use only two different foreign type objects, namely $F_1$ and $F_2$, but the following discussion can be adapted to situations where more than two foreign type objects are involved in the unification process. As before, we can distinguish the following cases based on their degree of generality:

(a) *$F_1$ and $F_2$ do not exist in $C_{imp}$'s schema.* Here the type objects and all instances as required by $C_{imp}$ are added. For example, travel agency $B$ that has no train information in its schema so far, wishes to import two related types called **Trains** and **Train-Fares** from travel agency $C$.

(b) *Either $F_1$ or $F_2$ is (semantically) equivalent to some object $L$ in $C_{imp}$'s schema.* If we assume that foreign type $F_1$ is equivalent to a local type, say $L_1$, several things need to be done. First a new subtype of $L_1$ is created, in order to hold the imported instances from $F_1$. Then $F_2$ is added as $L_2$ to $C_{imp}$'s local schema, including all of its instances. Finally, new functions relating $L_1$'s subtype and $L_2$ are created. For example, travel agency $D$ is importing the type **Airtravel** and the value type **Price** of function *has_price* from travel agency $C$ (see Fig. 3.5). Since **Airtravel** is related to $C$'s **Flights** it is added as a subtype of it. Then the type **Price** is added to the newly imported **Airtravel** just as in the original schema of travel agency $C$.

    In the case where both types $F_1$ and $F_2$ are present in $C_{imp}$'s schema, all foreign instances of $F_1$ and $F_2$ can be imported into the already existing local types $L_1$ and $L_2$ (type merge).

(c) *Either $F_1$ or $F_2$ is related to some object $L$ in $C_{imp}$'s schema.* Assume that $F_1$ is related to $L$. In this case, which is similar to case (c) of the first scenario, the same method applies. The functions that are common to both $F_1$ and $L$ are associated with a new supertype, which has two subtypes containing the functions that distinguished $F_1$ and $L$. Since $F_2$ was related to the original type $F_1$, it will also be related to the new supertype to which it is added as a stored function. For example, travel agency $D$ is importing the two types **Sightseeing** and **Cities** (containing the cities where the sights are) from travel agency $C$. $C$'s type **Sightseeing** is related to $D$'s local type **Entertainment**. Following the method outlined above, a new supertype is created by travel agency $D$, say **Things-to-do**, which contains all the information common to $D$'s **Entertainment**

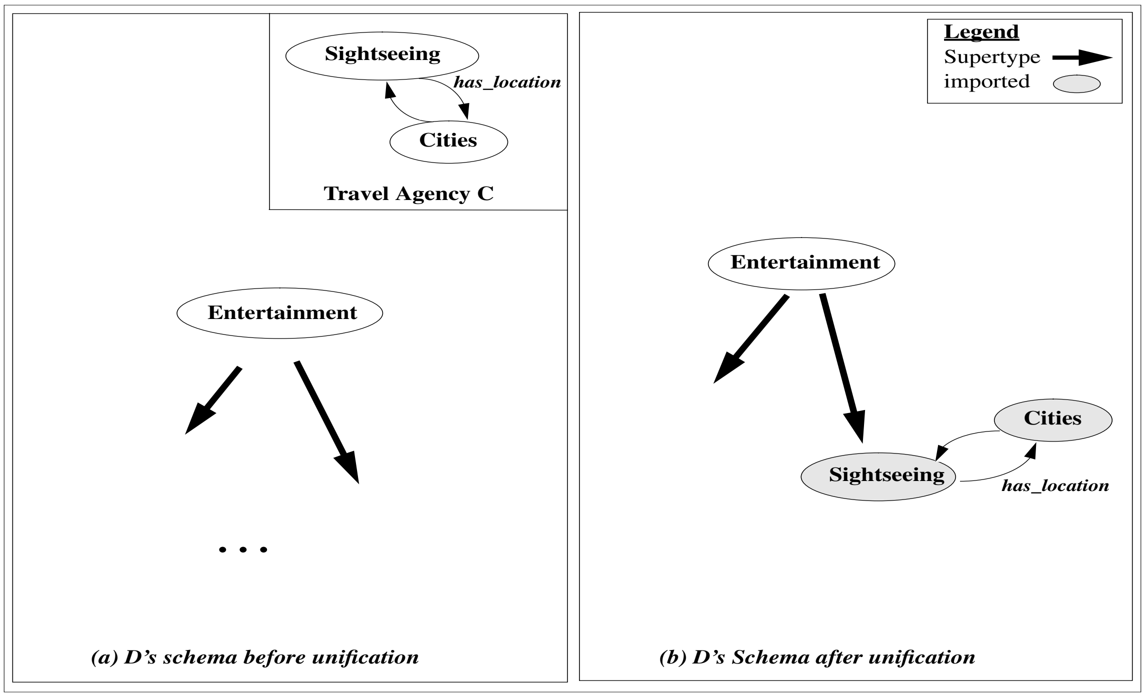


Figure 7.2: Unification of several inter-related objects: Example 1.

and $C$'s **Sightseeing**. The new supertype **Things-to-do** has two subtypes, namely **Entertainment** and **Sightseeing**, that reflect the differences between the original types. **Cities**, which is a stored function of both subtypes, is connected to **Things-to-do** via a stored function called *has_location* (see Fig. 7.2).

(d) *Both $F_1$ and $F_2$ are related to objects $L_1$ and $L_2$ in $C_{imp}$'s schema.* For example, travel agency $E$ with types **Places in Northeastern U.S.** and **Owner-Name** wants to import the types **Accommodations** and **Person** from travel agency $D$ (see Fig. 3.4). In this case, both types **Accommodations** and **Person** are imported separately. If the relationship between the two is important for the importing component, the appropriate function(s), in this case *is_managed_by* can be added afterwards.

3. In the third scenario, consider the case where component $C_{imp}$ wishes to import types from $C_{exp}$ whose relationships are modeled differently then the corresponding type relationships in $C_{imp}$'s local schema. For example, $L_1$ and $L_2$ are related through a ternary relationship that includes an additional type

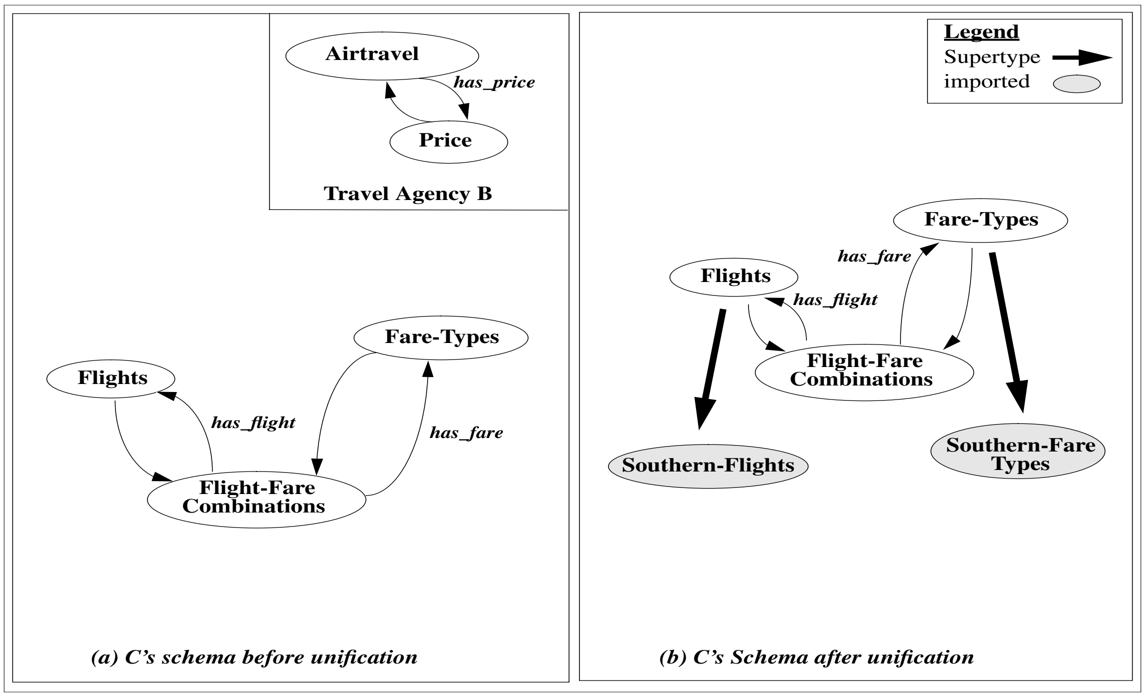


Figure 7.3: Unification of several inter-related objects: Example 2.

$L_3$ whereas the corresponding foreign types, $F_1$ and $F_2$, are related directly through functions and their inverses. Applying the method of "subsetting", two new subtypes for $F_1$ and $F_2$ are necessary. Then, for each instance pair that $C_{imp}$ imports, a new instance for $L_3$ is created, which relates the imported objects. As an example in FOTA, consider the following situation depicted in Fig. 7.3. Travel agency $C$ models its "flight-fare" information through a ternary relationship consisting of the types **Flights**, **Flight-Fare-Combinations**, and **Fare-Types**. In order to obtain fare information on charter flights from Miami to the Florida Keys, it decided to import this information from travel agency $B$. $B$ uses two types **Airtravel** and **Price** and a function *has_price* and its inverse to model the relationship between flights and fares. Using the procedure outlined above, two new subtypes for **Flights** and **Fare-Types** are created, namely **Southern-Flights** and **Southern-Fare-Types** respectively (to represent the fact that they contain information about the southern parts of the U.S.). Further, for every related pair of objects in the new subtypes, a new object is created in $C$'s **Flight-Fare-Combinations** relating each flight from **Southern-Flights** to its corresponding price object in **Southern-Fare-Types**.

These three scenarios are an attempt to examine the most important sharing situations that arise during the life-time of FOTA. We have only examined the importation and unification of single type objects or pairs of related type objects. However, the mechanisms presented in this section can be extended to accommodate the unification of three or more inter-related type objects. Some of these cases will be studied during subsequent research in this area. Although it is difficult to propose a useful measure of the "completeness" of this enumeration, we shall attempt to use the scenarios presented here as a foundation for our work, and argue that they are sufficient to prove the feasibility of this approach.

So far, we have not explicitly addressed the sharing of instances. It is important to note that this so-called instance level sharing requires the ability to compare individual instances in different components in order to prevent the unification of duplicates. This problem is addressed in more detail in [35] and will be left for further extensions to this work. Conceptually, it only makes sense to import instances that belong to a type which has been imported before. In this context, instance-level sharing can be compared to assigning the desired values to the function(s) that is (are) defined on the imported type(s) in the local schema. For example, in Fig. 7.2, component $D$ can import instances defined on $C$'s types **Sightseeing** and **Cities** by creating new objects for both types in $D$'s schema (we assume that $D$ has imported the types **Sightseeing** and **Cities** as well as the function *has_price* as can be seen in Fig. 7.2 (b)) and then assigning the desired values to the newly created objects under the *has_location* function. We will complete our discussion on schema unification in the next chapter when we present a mechanism for the sharing of behavior.

# Chapter 8

# Inter-Component Behavior Sharing

In the previous chapter we have described our theory for placing remote type objects into the best position of an existing local schema. Until this point, we have limited our discussion of partial schema unification to the sharing of meta-data objects. We have observed that in addition to the sharing of type objects sharing of instances [16] and sharing of functions, also termed behavior sharing, constitute the second and third aspect of partial schema unification respectively. In this chapter, we provide an overview of our underlying theory of behavior sharing in federated database systems [14]. We address the essential problems that are associated with each sharing situation and describe our mechanism to support sharing as implemented in our experimental prototype system.

## 8.1 Behavioral Objects in CODM

In CODM, behavior is represented by three types of functions:

- *Stored Functions*: A stored function records data as primitive facts in the database. Stored functions correspond to inter-object relationships (attributes).
- *Derived functions*: A derived function is defined by a data manipulation language (DML) expression. Derived functions correspond to queries (derived data).

- *Computed functions*: A computed function (sometimes termed a foreign function) is defined by a procedure written in some programming language. Computed functions correspond to methods (operations).

For the remainder of this dissertation, derived and computed functions are treated uniformly, and will be termed "computed functions" herein.

## 8.2 The Sharing of Behavior

Let us assume the existence of a function $\mathcal{F}$, which can be shared among components of a federation; without loss of generality, assume $\mathcal{F}$ takes as input the argument $a$[1]. The argument type can be a literal (i.e., **Integer**, **String**, ...) or a user-defined type such as **Accommodations**, for example. Sharing takes place on a component-pairwise basis, meaning that $\mathcal{F}$ is exported by a component $C1$ and imported by a component $C2$. The importing component is called the *local database*, while the exporting component is called the *remote database*. There are several ways components $C1$ and $C2$ can share the service provided by $\mathcal{F}$, depending upon the location where $\mathcal{F}$ executes and upon where its input argument $a$ resides (i.e., there are two degrees of freedom). At this level of abstraction there are four distinct *function-argument* combinations:

- local function - local argument
- local function - remote argument
- remote function - local argument
- remote function - remote argument

Upon closer analysis we can note that it is also necessary to differentiate between stored functions and computed functions. At this (finer) level of granularity, we can now distinguish between a total of eight different sharing scenarios[2], as presented in Fig. 8.1. In this table of Fig. 8.1, "Local" refers to the domain of the local database

[1] Since the argument can be a complex unit of information, this is not a limitation; multiple arguments can be handled by an obvious extension of our approach.

[2] We now have three degrees of freedom.

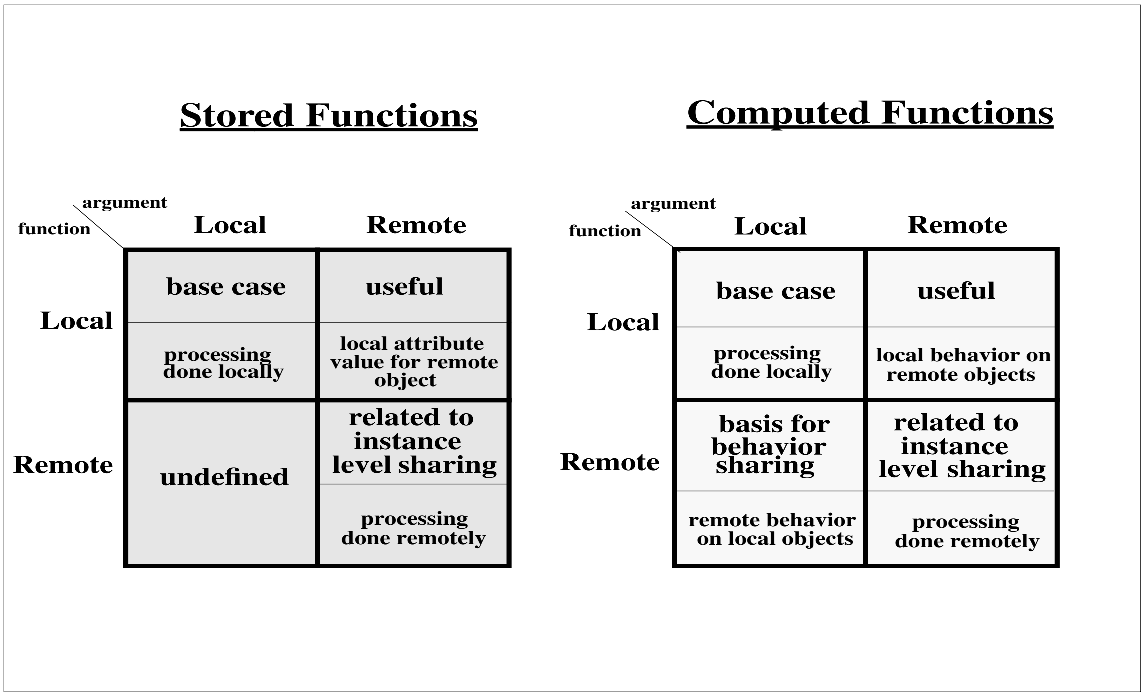


Figure 8.1: Eight different situations for sharing of behavior.

while "Remote" refers to the domain of the remote database. Local objects are those that belong to the local database, while remote objects belong to the remote database.

It is important to note that a principal goal of our approach is to provide a mechanism for behavior sharing that makes the location of a function and its argument transparent to the user. The details of how this transparency can be achieved are more fully described in Sec. 9.2, but we will briefly highlight the process here. In particular, we note that the state of a remote object (i.e., its functional values) always resides in the remote database, but when the object is imported to a local database, a surrogate is created for it in the local component. The creation of such surrogates is necessary in order to refer to remote objects using local database system tools without modification [15]. Since these surrogates are created locally, the local system is able to interpret and manipulate remote objects as usual, for example, when using them as arguments in local function calls. However, when retrieving the actual state of a remote object, the use of surrogates alone is not sufficient. Our approach exploits the extensible nature of our object-based database model by rewriting the

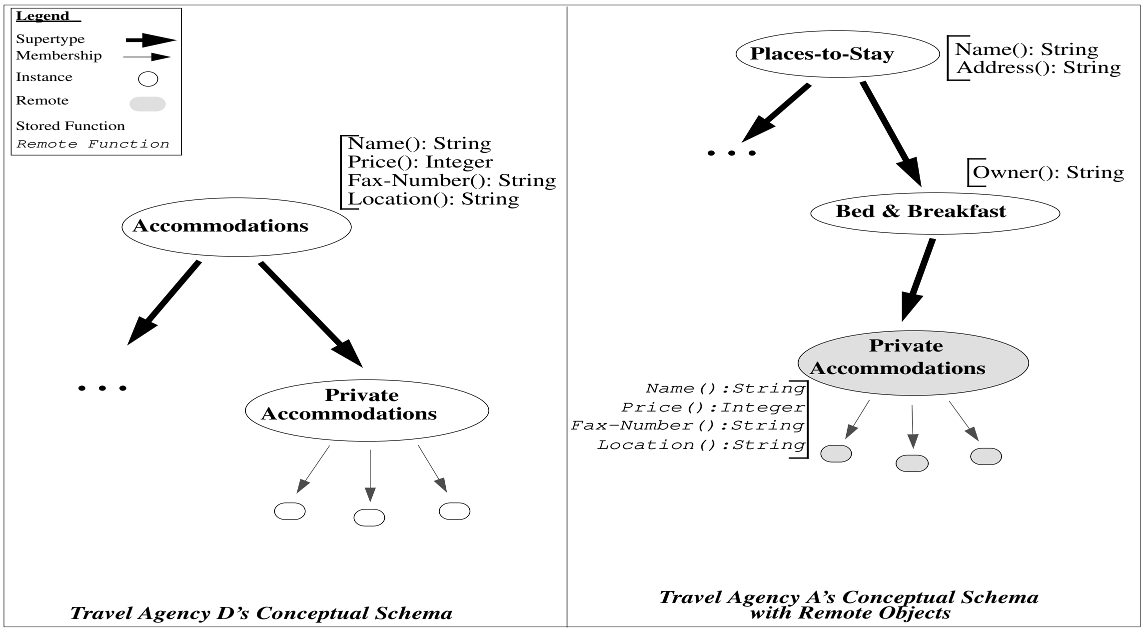


Figure 8.2: Stored functions in two component databases.

functions encapsulating surrogate objects as computed functions. These higher-level computed functions serve as place holders, and retrieve the results of applying the function on the remote component where the object is actually stored.

Given the above observations, it is now possible to consider each of the behavior sharing situations summarized in Fig. 8.1. We first focus on stored functions, and then turn our attention to computed functions.

### 8.2.1 Stored Functions

As a framework for analysis, consider again the familiar example of FOTA: Travel agency $A$ and travel agency $D$ contain related hotel information. Fig. 8.2 specifies parts of the meta-data (conceptual schemas) for these two example components[3]. Let travel agency $A$ denote the local component and travel agency $D$ denote the remote component.

The four situations for the sharing of stored functions among components can be analyzed as follows:

[3]In order to keep the signature of each function as short as possible, we have omitted the input argument whenever it is obvious; the result type, on the other hand, is always explicitly shown.

1. *Local function - Local object*

   This is what we term the *base case*. Both objects, the stored function $\mathcal{F}$ and its argument $a$, reside in the local component and can be executed as usual without any additional mechanisms.

2. *Local function - Remote object*

   In this case, $\mathcal{F}$ is a local stored function that is applied to remote argument $a$. For example, component $A$'s $Address()$ function can be applied to the **Private Accommodations** that have been imported from component $D$. As previously mentioned, surrogates are created in component $A$'s database for each remote object. These surrogates are collected in a newly created subtype of **Bed & Breakfast** called **Private Accommodations**. All local functions, for example the functions defined on **Places-to-Stay** and **Bed & Breakfast** in Fig. 8.2, operate normally on the surrogates (i.e., the instances of **Private Accommodations**). In the case of local functions that do not have counterparts in the remote component (e.g., $Address()$), new function values can be created locally for each imported instance. As a result, each imported object also has local state in component $A$'s database and can be altered by $A$ without affecting the original object in component $D$'s database. The four functions ($Name()$, $Price()$, $Fax - Number()$ and $Location()$) that are shown next to component $A$'s **Private Accommodations** in Fig. 8.2 are actually remote functions defined in component $D$'s schema; these have been included in $A$'s database for completeness. Exact details of how these functions are created and invoked are presented in Sec. 9.2. Provided that component $A$ has created new values for $Address()$ for each of the imported **Private Accommodations**, s/he might pose the following OSQL query in this context which returns the name and address of each private accommodation:

   ```
   select Name(p), Address(p)
   for each PRIVATE ACCOMMODATIONS p
   ```

   In this query, the variable $p$ is defined on all instances of type **Private Accommodations**.

3. *Remote function - Local object*

   This situation is somewhat meaningless, since stored functions only have a meaning in the local context of the component in which they were initially created. For example, suppose that component $A$, who does not have the $Fax - Number()$ function defined in his/her schema, wants to see the fax numbers for all his/her **Places-to-Stay**. Component $A$ would first need to create a new function and then populate it with the appropriate values, rather than being able to use component $D$'s $Fax - Number()$ function. However, when looking at this situation from component $D$'s point of view, one can argue that this case is merely the mirror image of the second case (Local function - Remote object). Rather than executing component $D$'s function remotely in component $A$'s database, one can integrate component $A$'s **Places-to-Stay** into component $D$'s schema (create a new subtype, **Places-to-Stay**, of **Accommodations** and populate it with component $A$'s **Places-to-Stay**) and execute the $Fax - Number()$ function locally.

4. *Remote function - Remote object*

   This case serves as the basis for *instance level sharing*. All remote objects of interest to a component, say $A$, must already be imported into the local database using surrogates. Each time a remote object is referenced, the surrogate "points" to the remote object and the desired functional value is fetched using a remote procedure call to the other component (see Sec. 9.2). In this situation (see Fig. 8.2), component $A$ might pose the following OSQL query against his/her schema:

   ```
   select Fax-Number(p)
   for each PRIVATE ACCOMMODATIONS p
   ```

   As before, $p$ is a variable defined on type **Private Accommodations**. In order to find the $Fax - Number()$ of all remote **Private Accommodations**, surrogates for each of these accommodation instances are used to locate the remote accommodations. For each remote accommodation, the fax number is

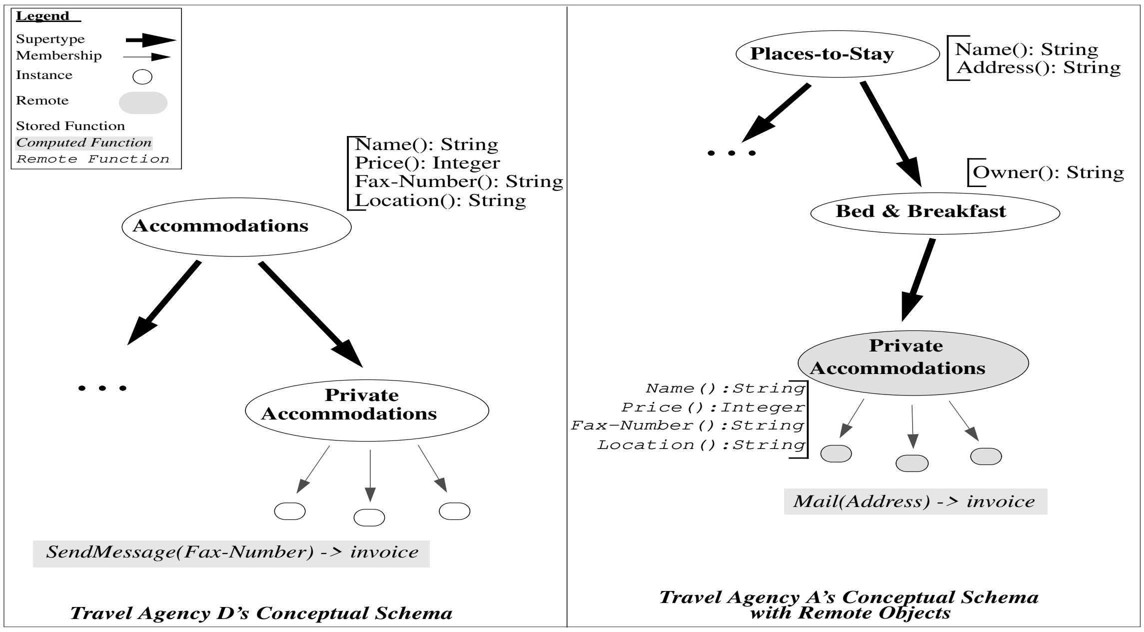


Figure 8.3: Stored and computed functions in two component databases.

determined using the remote $Fax-Number()$ function and the result of this function is returned to the local database.

### 8.2.2 Computed Functions

In order to consider sharing for computed functions, consider again the example of two collaborating travel agents in FOTAs. Fig. 8.3 shows the two example component databases as before, but with additional (computed) functions: **SendMessage()** in component $D$'s schema takes a fax number as input parameter and then opens a text editor which lets the user create a message text. At the end of the editing session, the composed text is faxed to the specified number. **Mail()** in component $A$'s schema uses an address (throughout this chapter we assume that address refers to an e-mail address) and sends a precomposed text to that e-mail address. The four situations for the sharing of computed functions among components can be analyzed as follows:

1. *Local function - Local object*

   As in the case of stored functions, this is the base case. Computed function $\mathcal{F}$

as well as its argument $a$ resides in the local component and the execution is local (e.g., in component $D$: **SendMessage(Fax-Number(**$a$**)))**.

2. *Local function - Remote object*

   This situation can be reduced to the base case described in case 1. For example, if component $A$ wants to mail a note to one of component $D$'s hotels, s/he will run *Mail* on the *Address*() of the surrogate for that hotel, say *inst* (e.g., **Mail(Address(***inst***)))**.

3. *Remote function - Local object*

   This is the reverse of the previous case: the function executes remotely and the input argument is supplied from the local database. For example, component $A$ may desire to send a fax message but does not have a fax program in his/her own local database[4]. In this case, s/he will invoke component $D$'s *SendMessage*() function remotely through a previously created handle in his/her own database and supply it with a local argument. In effect, the remote database is providing a non-local "service". Intuitively, from component $A$'s perspective, this is what "sharing of behavior" corresponds to.

4. *Remote function - Remote object*

   This situation is similar to the first case (Local function - Local object) in that both the state of the object and execution of the function are in the same component. For example, component $A$ sends a fax message to one of $D$'s **Private Accommodations** using $D$'s original *SendMessage*() function. Compared to the first case (Local function - Local object) where no sharing takes place and execution occurs locally, in this case all the processing is done on the remote site. In order to invoke a remote computed function using remote arguments from within the local component, surrogates for the remote objects must be created locally. These surrogates enable the local component to access the actual state of desired objects which reside with the remote component. This procedure is similar to instance level sharing (see Sec. 9.2.2.1) where surrogates for shared instances are created locally in order to provide access to the actual state of each instance in the remote component.

[4]For simplicity, we assume that $A$'s hardware include a fax modem.

In the examples above, the functions being shared have returned a literal type (e.g., the *Address*() function returns a **String**). However, functions with signatures involving abstract (user-defined) types can also be shared. In this case, both the input and output argument types must be defined locally; if they are not, their meta-data must be imported beforehand. The location of the result argument is determined by the location where the function executes.

## 8.3 Observations on the Practical Use of Behavior Sharing

With the above analysis and framework in place, it is now possible to make some observations on the practical utility of the behavior sharing capabilities supported by our mechanism. In the above analysis of behavior sharing, we stressed the separation of the location where the function executes from the location where the data resides. However, from a user's perspective, this separation of function execution and argument location is completely transparent.

Our analysis of eight different sharing patterns can be reduced to two "most interesting" cases: (1) executing an imported function on a local argument, and (2) executing a local function on an imported (shared) argument. The first case involves the reuse of a previously defined function in a different environment. This "reuse of function" is a principal reason why components would want to share behavior. The second case can be described as extending the "characteristics" of a remote object while at the same time respecting the autonomy of the originating site. That is, the importer can customize the remote object according to his local conceptual schema. These local attributes are managed entirely by the local component avoiding any unnecessary modification to the originating (remote) component.

In both cases above, the user is not aware of the environment in which a shared function executes, and need not worry about where the state of an imported object actually resides. Instead, components are able to freely browse the meta-data and behavior that have been made available (i.e., exported) by others in the federation in order to select the services that they would like to share[5] (i.e., import).

[5]Our discussion has in a sense assumed that there are no access restrictions in place that would further complicate the sharing process. An investigation of behavior-based authorization in object-based databases is the subject of a related research project at USC.

# Chapter 9

# Experimental Prototype Implementations

We have implemented two separate prototypes in order to verify the underlying theories of this research. The first prototype is intended to prove the feasibility of our mechanism to resolve representational discrepancies among related objects in different federation components. Specifically, we show the use of ontologies and relationship descriptors as a means of adding semantic information to conceptual database schemas. By implementing this prototype, we show that this additional semantic information is sufficient to resolve semantic heterogeneity in a loosely coupled federation environment such as FOTA. The details of this prototype are described immediately below. In our second prototype we focus on unifying remote objects with existing schemas. Besides the sharing type objects, we paid particular attention to our implementation of instance level sharing as well as the implementation of the behavioral sharing patterns described in Chapter 8.

## 9.1 Semantic Heterogeneity Resolution in the SHELTER Testbed

We have investigated the resolution of semantic heterogeneity in the SHELTER knowledge base environment [56]. In essence, SHELTER offers an environment for collaborative, team development of knowledge based systems. Knowledge bases created with SHELTER are translated into LOOM [48] objects. LOOM is a knowledge representation system that supports a description language for modeling objects and relationships. Furthermore, its production-based and classification-based inference capabilities support deductive reasoning. LOOM knowledge bases can be permanently stored using the ITASCA storage manager [36]. By using the SHELTER testbed, we were able to make use of LOOM's knowledge representation and inference mechanism without having to implement our own from scratch. For these reasons, SHELTER became the ideal test vehicle for our approach to resolving object relationships. Note that since we are dealing with knowledge bases instead of databases, we are using the term concept in place of object for the remainder of this section.

The first step in building this prototype was to create several knowledge bases that resembled components of FOTA. In order to correctly simulate our resolution mechanism, we also implemented a semantic dictionary (in a separate knowledge base) and local lexica (part of the component knowledge bases). Recall that the semantic dictionary contains, among other things, the ontology for the application domain, in our case the collaborating travel agencies. The ontology is used to describe unknown terms which appear in the export schema of each component. The semantic dictionary is the first knowledge base that is created and defined in SHELTER. In this way, components can use the already defined ontology concepts in their local lexica. Once the semantic dictionary is defined, the order in which the other knowledge bases are created and loaded into SHELTER has no effect on the resolution mechanism. Fig 9.1 depicts a sample session with SHELTER during the creation of the travel agency knowledge bases. Refer to Appendix B for a complete listing of the LOOM script files that are used to create three FOTA components and the semantic dictionary as LOOM knowledge bases.

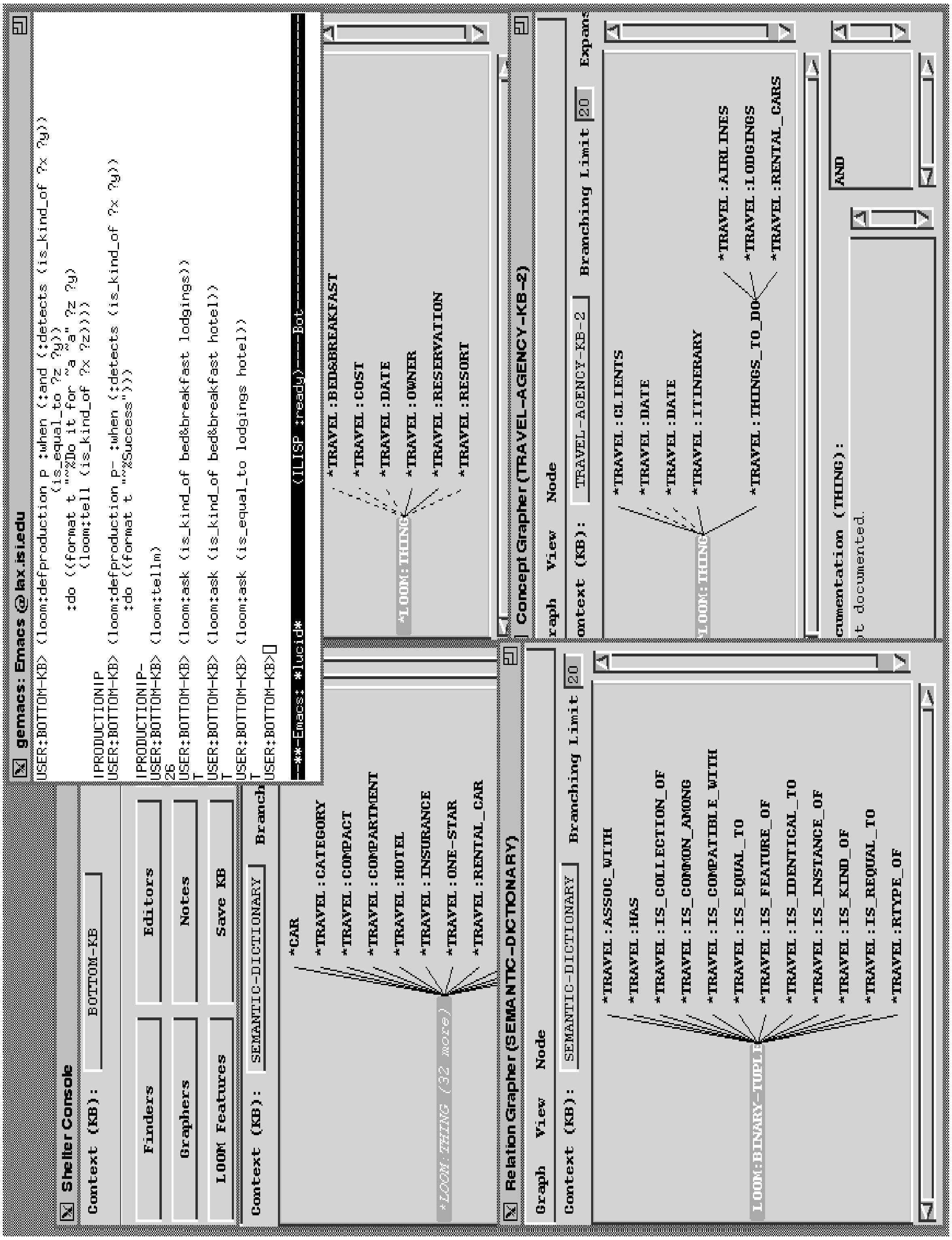


Figure 9.1: The SHELTER development environment.

The second part of this prototype consists of the implementation of so-called *resolution rules* that LOOM's inference engine can use in order to disambiguate related objects in different knowledge bases. Resolution rules are formulated to capture object relationships which are described by relationship descriptors. Resolution rules are of the form:

**If** {list of facts connected by "and"} **then** {newly inferred fact}

Based on the set of relationship descriptors introduced earlier a sample resolution rule using LOOM syntax is as follows:

```
(defproduction p :when (:and (:detects (is-kind-of ?x ?y))
                                       (is-equal-to ?z ?y))
  :do ((format t "~%Do it for ~A ~A" ?x ?z)
       (tell (is-kind-of ?x ?z))))
(defproduction p- :when (:detects (is-kind-of ?x ?y))
  :do ((format t "~%Success")))
(tellm)
```

In plain English, this resolution rule can be described as follows:

**If** $X$ KINDOF $Y$ and **if** $Z$ EQUAL $Y$ **then** $X$ KINDOF $Z$

In this rule, $Y$ is a concept from the ontology, and $X$ and $Z$ are concepts whose relationship is being investigated. Using this resolution rule for example, we could "inquire" about the relationship between two concepts from different schemas, for example **Bed&Breakfast** from component $A$ and **Lodgings** from component $E$:

```
> (ask (is-kind-of Bed&Breakfast Lodgings))
```

Based on the knowledge already defined in $A$'s and $E$'s knowledge bases, namely the fact that **Bed&Breakfast** is KINDOF a **Hotel** ($A$'s local lexicon) and the fact that **Lodgings** is EQUAL to **Hotel** ($E$'s local lexicon, see also App. B), LOOM's inference engine will return:

```
T
```

We have implemented resolution rules that cover most of the relationships that can occur in object-based data models. For example, we have transitive rules such as

$$\textbf{If } X \text{ Rel } Y \text{ and } \textbf{if } Z \text{ Equal } Y \textbf{ then } X \text{ Rel } Z \qquad (1)$$

where $X$, $Y$, and $Z$ are as above and Rel is a relationship descriptor in the set: {Identical, Equal, KindOf, Assoc, CollectionOf, InstanceOf}. Furthermore, we have rules comparing characteristic properties of concepts. The general structure of so-called property rules is:

$$\textbf{If } X \text{ Rel } Y \text{ and } \textbf{if } Z \text{ Rel } Y \textbf{ then } X \text{ Assoc } Z \qquad (2)$$

Again, $X$, $Y$, and $Z$ are as above and Rel stands for either Feature or Common. All in all, we estimate the total number of resolution rules to be around 50; so far, in this prototype we have implemented 15 of these rules, enough to demonstrate the correctness of our resolution mechanism.

For each pair of related objects, there may be more than one resolution rule that is applicable, resulting in more than one possible relationship. In order to arrive at a solution that describes the relationship in the most accurate way, we have assigned a value to each relationship descriptor indicating its expressive power. In this prototype, Identical has the highest expressive value, followed by Equal, KindOf, Compatible, and Assoc, which has the lowest. CollectionOf and InstanceOf are on the same level as KindOf. Common, Feature, and Has are not rated since they will not be used to describe results of resolution rules.

Going back to our example of involving $A$'s **Bed&Breakfast** and $E$'s **Lodgings**, we end up with two relationship descriptors describing their relationship, namely KindOf and Assoc. The positive association is due to the fact that both **Bed&Breakfast** and **Lodgings** are of the "One-Star" category (rule (2) above). Since KindOf has a higher descriptive value than Assoc, it will be used as the final result of the resolution process. In those cases where no exact relationship between two concepts can be established (except for Assoc), the resolution process depends on the input from the user. So, for example, whenever the resolution rules return positive associations (Assoc) the user is asked to intervene and decide on the final outcome. Note, the more positive associations there are, the more likely it is that the objects are at least compatible or even equal.

We have defined a lisp function *relationship?* which applies the algorithm described above to a pair of concepts which are under investigation. Thus, rather than manually applying every resolution rule that could apply, we can simply ask

```
> (ask (relationship? Bed&Breakfast Accommodation))
```

and LOOM will return

```
('Is-Kind-Of)
```

Based on the assumption that our set of relationship descriptors is capable of describing most of the relationships that can be modeled in CODM like data models, the above resolution rules proved sufficient for resolving all the semantic heterogeneities occurring in FOTA. After completing this prototype, we are satisfied with the performance of our mechanism and feel comfortable that more complicated relationships can be resolved by extending the set of relationship descriptors. For example, the current set lacks the ability to describe exactly how two objects are related if they are not equal, identical, or subtype/supertype of one another. In this case, Assoc is the only other alternative and simply indicates that there is some kind of positive relationship between these two objects. Further prototype implementations will test the use of additional relationship descriptors and corresponding resolution rules, moving us even closer ahead to a design that can be implemented and used in real-world applications.

## 9.2 Schema Unification in the Remote-Exchange Testbed

An experimental implementation of our sharing and transmission mechanism has been designed and built using our Remote-Exchange testbed consisting of a federation of Omega database components[1].

### 9.2.1 The Prototype

In the current Remote-Exchange testbed, we have implemented the seamless (transparent) unification of objects from remote databases with local schemas. Each federation component consists of an Omega DBMS for maintaining local information

[1]The current Remote-Exchange environment also supports IRIS database components to some degree, and we plan to incorporate other database systems in the future.

and an *importer/exporter* for handling communication requests to and from remote components. Communication among individual Omega components is supported using the RPC message passing paradigm. In what follows, we describe the essential aspects of this testbed, and examine some of the critical implementation issues we faced in our experiments.

As noted, our unification mechanism requires the ability to extract meta-data information from a database component. Since the original functional database model underlying the Omega DBMS does not support this functionality[2], we implemented our remote sharing language *RSL* as part of each exporter/importer[3]. In effect, *RSL* together with the exporter/importer serves as our inter-component communication protocol.

Sharing in Remote-Exchange is made possible through the implementation of *RSL* commands in form of an X11-based graphical user interface. This user interface called *discovery* and *unification tool* hides the syntax of the pure *RSL* commands from the user and provides a more flexible and intuitive basis for communicating with component schemas. At this point, automatic conflict resolution has not been implemented in our Remote-Exchange prototype yet, and the user must perform this task manually before unification can occur. Fig. 9.2 shows a sample session with the unification tool. In this session, travel-agency $A$ is the local component and we see parts of its schema displayed in the large window titled *****| Omega DBMS |*****. Travel-agency $A$ has opened a connection to another remote component travel-agency $D$ on host **aludra**. Information about this remote component is displayed in the smaller window to the right, titled *****| Discovery Tool |*****. At any point, $A$ can select among the operations displayed in the bottom row of each window. In fact these choices correspond roughly to the *RSL* commands that are implemented in our prototype. At this point, $A$ is browsing meta-data in $D$'s export schema. Since $A$ has not selected a type in $D$'s schema, neither the **Show Values** nor the **Import...** buttons are activated. Locally in $A$'s schema, the **Places-to-Stay** type

[2]Besides the lack of an *RSL* equivalent, there is a close resemblance between Omega's functional data model and CODM.

[3]Another alternative would have been to implement this functionality inside each Omega DBMS; however, it would require modification of the kernel of the database system which did not seem acceptable. Furthermore, additional modifications would be necessary every time the *RSL* is extended with new functionality.

is selected and its functions are displayed in the **Functions On Type** window. If $A$ selects one of the three functions (i.e., *Name*, *Address*, or *Owner*) the *Show Values* button on the bottom of the large window becomes active and allows the user to display the value of the selected function for all instance objects belonging to the underlying type. The ability of our graphical interface to allow the selection of types and functions on the screen eliminates the need for set operations involving OID values that would otherwise be necessary in a character based interface (e.g., store the set of OID's returned by *HasInstances()*, select each OID individually from the set, call *HasValue()*).

In the next sections, we describe our approach to implementing the sharing of instance and function objects in the Remote-Exchange testbed, i.e., the *RSL* commands *ImportInstance()* and *ImportMethod()*.

### 9.2.2 Sharing and Transmission

Conceptually, in instance level sharing, a remote instance object is imported directly into a local type. This remote instance behaves in the same manner as a local instance object from the user's perspective. However, the actual state of the remote instance exists in the remote component database; retrieval of any state of the remote object is done by accessing the remote database transparently. Hence, access to remote instance objects corresponds to the Remote function - Remote object situations described above[4].

The importation of a function object can be seen as follows. Intuitively, when an instance object is imported, only data is being shared. On the other hand, importing a function object gives the importer access to services not provided by his/her local system. This corresponds to the Remote function - Local object situations described in Chapter 8.

#### 9.2.2.1 Instance Level Sharing Implementation

Our mechanism for importing instance level objects follows three steps:

1. Create local surrogates for remote objects.

[4]Note that this does not depend upon whether the function is stored or computed.

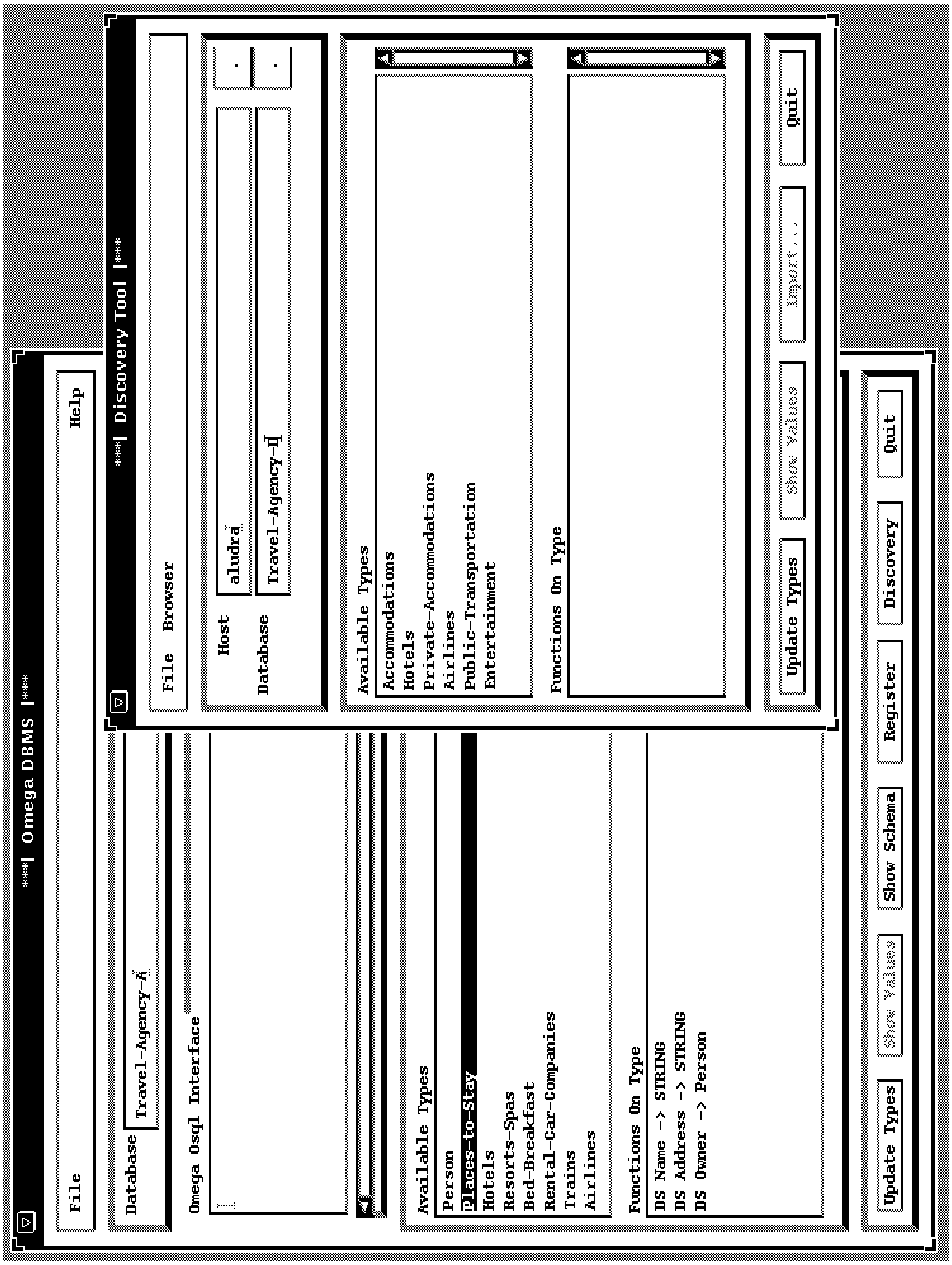


Figure 9.2: The Remote-Exchange sharing interface.

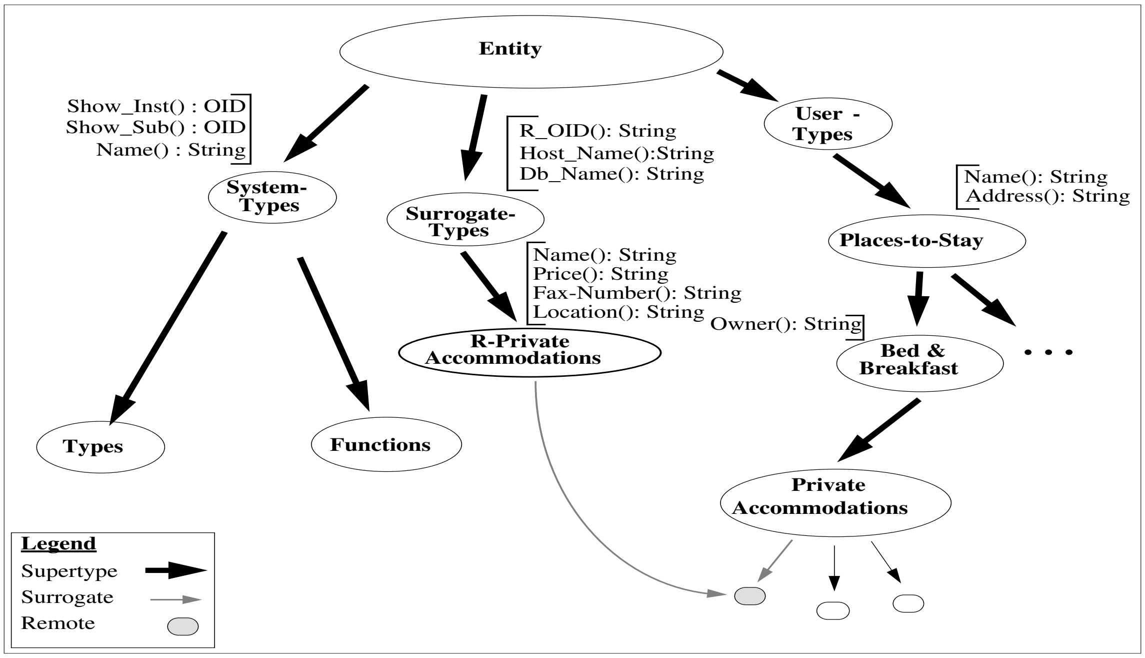


Figure 9.3: Sharing instance objects.

2. Create computed functions for retrieving data from remote components.

3. Overwrite functions defined on surrogates to use (or refer to) the newly created computed functions in step 2.

The local surrogate serves as a local handle for accessing a remote instance. By using the surrogate, differences between remote representations of objects, e.g., object identifiers (OIDs) can be masked out (made transparent). Since the state of the remote object exists externally, computed functions for accessing that state must be created. These computed functions use the Remote Procedure Call (RPC) paradigm [7] for accessing the remote component. Finally, in order to have the surrogates use the remote functions, any existing functions defined on the surrogate must be overridden to use the RPC defined functions. This is implemented by dynamically binding functions to objects.

In terms of the collaborating travel agents, we can envision a scenario where travel agent $A$ wants to import instances of another remote type **Private Accommodations** into his/her local schema using the above approach. This situation may occur after $A$ has already imported the meta-data, in this case **Private Accommodations**, and is interested in remote instances belonging to this type. This situation is depicted in Fig. 9.3. Surrogates are created as instances of both a local type (i.e., component $A$'s **Private Accommodations**) and a remote type called **R-Private Accommodations** in Fig. 9.3. The purpose of the new surrogate type is to override the local functions that surrogates inherit from the local type to which they belong (e.g., $A$'s *Address()* function defined on **Places-to-Stay**). By additionally creating the surrogate as a member of this surrogate type, the functions that the surrogate instance originally inherited are overridden. The two thin dotted arrows from **R-Private Accommodations** ("R" for remote) and **Private Accommodations** to the surrogate instance serve to indicate that surrogate instances (i.e., remote instances) of **Private Accommodations** are created as members of both **Private Accommodations** and **R-Private Accommodations**. Thus these (remote) instances inherit functions multiply from both **R-Private Accommodations** and **Private Accommodations**; any duplicately named function from the two types is overridden by the function defined for **R-Private Accommodations**. The functions defined on **R-Private Accommodations** are computed functions, which make RPC requests to the remote component database and retrieve the values of functions on remote instances.

#### 9.2.2.2 Function Level Sharing Implementation

As in instance level sharing, meta-information containing the location (e.g., remote OID and remote component name) of the remote function object being imported must be stored locally. However, by contrast with the case for instance level sharing, this meta-information is associated directly with the function being imported. In instance level sharing, the meta-information is indirectly kept for remote functions such as *Name()* and *Location()* via the surrogate instance object (see Fig. 9.3). Thus, in our implementation we distinguish between two kinds of "remote" functions: those implicitly defined through instance level importation and those directly imported through function level importation. Fig. 9.4 shows our mechanism for incorporating meta-information for function level importation. We exploit the fact that meta-information is also represented using our functional object-based model. Imported

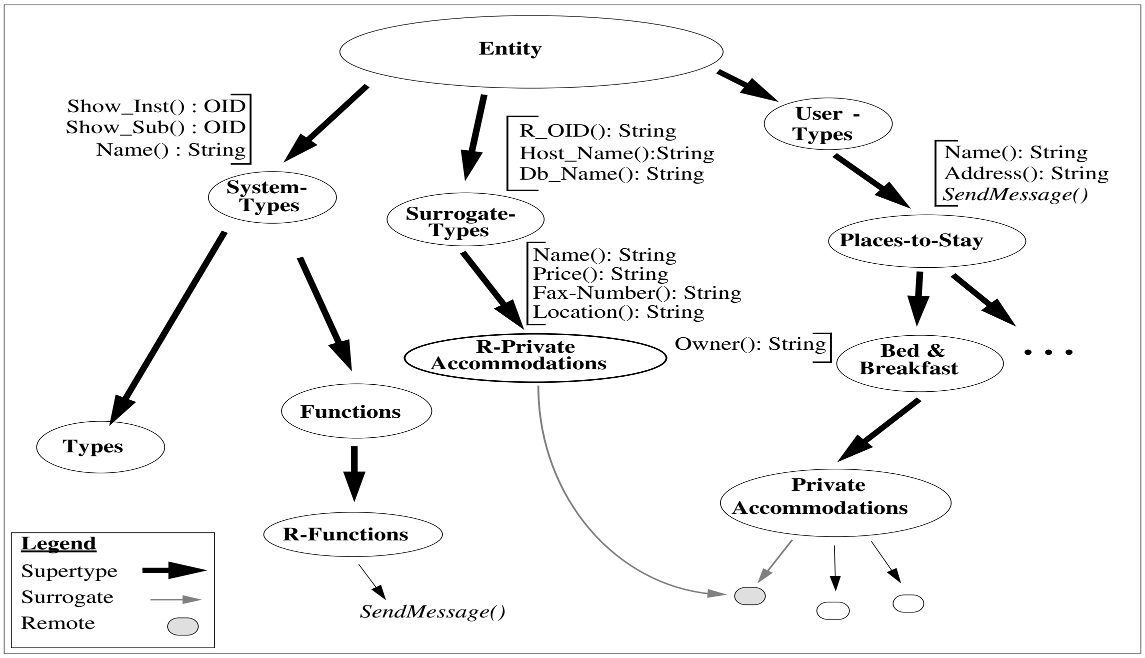


Figure 9.4: Sharing function objects.

functions are created as instances of the type **Remote-Functions** and can thus store and access the additional location meta-information required to execute the imported function.

Fig. 9.4 depicts a slightly different sharing pattern from Fig. 9.3. In this scenario, the *SendMessage()* function[5] is imported from some remote component (i.e., component $D$). Note that both the remote and local instances of **Private Accommodations** use the same imported *SendMessage()* function for retrieving fax numbers. This is evident in Fig. 9.4 by the absence of the *SendMessage()* function from the type **R-Private Accommodations** and the addition of a new (italicized) *SendMessage()* function defined on **Places-to-Stay**[6].

In order to explain how the *SendMessage()* function works, we must first explain how our implementation addresses the issue of side-effects. By side-effect we really mean two things: (1) any kind of implicit input other than the input argument that is necessary to compute the result, and (2) any modifications to the state of

[5]Assume for the remainder of this example that component $A$ does not have an equivalent function such as the *Mail()* function described earlier.

[6]The italicized font is used to indicate that the *SendMessage()* function is imported.

the database where the function executes other than to the input argument. For functions whose arguments are literals, this simply requires that the function being imported computes its result value solely based on its input argument without modifying any database state. Functions whose input argument is non-literal pose additional difficulties. In this case, the input argument is the OID of an instance. The problem then lies in determining what information a computed function accesses in order to compute its results. Strictly applying our definition of side-effects would restrict computed functions on non-literals to solely accessing and then manipulating the input OID. But realistically, a computed function must be able to access some state of the instance corresponding to the OID when computing its result. In our implementation, we take the position that the only state that a computed function can access are those functions that serve to encapsulate that object. In other words, the only state the computed function will possibly access are those functions that are defined on the types of which the instance is a member. In the case of the *SendMessage()* computed function, these functions are *Name()*, *Price()*, *Fax-Number()*, and *Location()*.

Having determined what information a computed function on a non-literal type can access, a problem arises when trying to execute such a function remotely on a local object. The problem occurs when supplying local arguments to a remote computed function. Although we know the computed function is limited to only accessing the functions that encapsulate the instance, we do not know exactly which ones it does need. Even if we pass all the possible values the computed function can access, the computed function must be written in such a way as to retrieve these arguments from the remote component and not the local one. This would be undesirable and contrary to our goal of relieving the computed function writer of needing to know where the data on which it operates is located.

The best approach to this problem in our autonomous environment is to allow computed function writers to define functions without concern as to whether the function is to be exported. Thus, in our implementation, whenever a remotely executing computed function needs state from the local component, it performs a callback to the local component to retrieve that state. For the instance level sharing described in Sec. 9.2.2.1, the local component simply runs as a client and makes RPC requests to the exporter running as a server. However, for function level sharing, when the callback mechanism is used, the local component must in addition run as a server to accept the callback requests. Computed functions can be written using any programming language that can be compiled to re-entrant object code. This object code is then dynamically linked into the database management system kernel when the computed function is accessed. In our prototype using the Omega database management system, a computed function accesses the local component through *Omega_eval()*, which has two parameters: an argument and the function that is to be applied to the argument.

We can now consider how the callback mechanism works transparently and how it allows a computed function to be written uniformly without regard as to whether that function is to be exported. Consider again the example using the *SendMessage()* computed function. Suppose that the imported *SendMessage()* function retrieves the *Fax-Number()* of an instance and faxes a precomposed message to the retrieved number. When the user of the local component invokes the *SendMessage()* function on a local object, the *SendMessage()* function is passed the local OID of a **Places-to-Stay** instance. The *SendMessage()* on the remote server makes a call to *Omega_eval()* to retrieve the *Fax-Number()*, *Omega_eval()* recognizes that the OID passed in as its argument is not local and performs a callback to the server of the local component that invoked the *SendMessage()* function. Since the local server recognizes the OID as a local OID, it performs the request and passes back the *Fax-Number()* to the remote server which can than complete its task and display an acknowledgment on the local database monitor.

# Chapter 10

# Conclusion

We have presented an approach and mechanism for resolving semantic heterogeneity in the context of a federation of autonomous database system components. While our mechanism is general enough to apply to schemas in most data models, we claim successful resolution only within the context of object-based multidatabase systems. For this mechanism to operate effectively, each participating component must agree to meet two principal conditions: First, the CODM data model must be supported at the federation interface, including the RSL functionality, as presented in sections 5.1 and 6.2.1. Second, a local lexicon must be provided, wherein a component describes the meaning of the (type) objects it is willing to share with other components in federation (see Sec. 6.2.2). Sharable objects must be described using the conceptual relationship descriptors supported by our mechanism.

## 10.1 Summary

The approach to semantic heterogeneity presented in this dissertation is couched in the framework of Remote-Exchange, a research project currently underway at USC [24]. The Remote-Exchange architecture provides three "services" to its federation components: an intelligent sharing advisor, semantic heterogeneity resolution, and unification [23]. When a new component initially joins the federation it must first register with the sharing advisor. The sharing advisor enters the data that the component is willing to share into the semantic dictionary so that it can be used by other components in the federation. Assuming that a component has met the two basic conditions above, it is able to participate in the exchange of information. Sharing takes place on a component-pairwise basis. The importing component selects relevant foreign objects and invokes the sharing advisor to resolve semantic heterogeneities that might exist between the foreign objects and local objects in its own schema [52]. Given a foreign object, a related local object and the relationship between the two, the unification tool places the foreign object into the appropriate place in the local schema. At this point, the unification is complete and the newly imported object can be used by the remote component.

We note that at present it is not possible to completely automate the process of resolving semantic heterogeneity. In consequence, humans will likely be required to assist. In order to facilitate the establishment of new federations, we employ so-called ontologies, which contain an initial set of terms that can be used to describe unknown concepts in the local lexicon of each component. A complete ontology package describes general as well as specific information from the application domain, and will evolve and grow with time to accommodate additional, more complex concepts within a given federation. Such packages can be provided for particular domains, e.g., the travel industry or genetic information.

In this dissertation, we have also presented an approach and mechanism to support the sharing of behavior among the component database systems in a federation. We considered the various situations for supporting the sharing of behavior. In our approach, we have considered the importance of decoupling the location of (persistent) data and the location of the execution of methods that operate on it. Traditional approaches inextricably link the location of the data and the execution of the operation. Other research efforts specifically address the issues in this area [9]. However, our approach allows the local component to create local (e.g., stored) functions on remote objects. This has the overall benefit of allowing the local component to create local state for remote objects that completely conform to the local system's mechanism for updates.

Experimental prototypes of our resolution and sharing mechanisms have been developed. We have simulated a federation similar to FOTA using the SHELTER development environment in order to test our resolution mechanism. Furthermore, we have built upon the Remote-Exchange prototype to demonstrate, evaluate, and refine our unification mechanism. The Remote-Exchange prototype is based upon a testbed consisting of Omega DBMS components. Omega was chosen for two reasons. First, by using an exiting DBMS we were able to focus our attention on implementing the resolution and sharing mechanism. Second, the Omega database model contains most of the modeling constructs specified in CODM. The focus of our experimental prototype implementation has been mainly on instance- and function-level sharing. We have found these sharing patterns to be very natural and easy to use in our environment. In light of these achievements, we are currently further investigating type-level sharing (involving complex abstract types) and are proceeding to quantify the performance efficiency of our mechanism.

## 10.2 Results

The result of this research have both direct and practical impact on information sharing among heterogeneous databases, specifically in the following areas:

- *Framework:* We have established a framework for accommodating semantic heterogeneity in interoperable object-based database systems. We specifically use an functional object-based data model (extended with a sharing language called *RSL*) for describing the sharable data as well as their relationships to the real-world concepts they represent.

- *Architecture:* We have introduced an architecture and experimental system for resolving semantic heterogeneity. Our system is based on the interaction between *RSL* that provide structural information, local lexica that provide semantic information, and the semantic dictionary that provides partial or incomplete knowledge about the relationships between the concepts in the local lexica. Relationships are described using a set of fundamental descriptors, which is extensible.

- *Existing Components and Autonomy:* Throughout this research we have paid careful attention to limiting required modifications to existing DBMS software and conceptual schemas. As a result, our approach requires no modification to the query processor or any other aspect of the local architecture. Other than the basic requirements of a CODM-like interface and support of a local lexicon, each component retains autonomy over its database. Furthermore, through the export schema it can specify at any given time which objects are sharable and which objects should remain private.

Our approach to resolving semantic heterogeneity can be utilized by a large variety of intelligent and cooperative information systems, such as *Services and Information Management* (SIMS) [2], GTE's *Distributed Object Management* project [50], and OMG's *Object Request Broker* [57] to name a few.

## 10.3 Future Work

We are planning to extend our current *discovery* and *unification tool* to include the semantic heterogeneity resolution mechanism, thus getting closer to our final goal of creating an intelligent sharing advisor to oversee all of the operations in the federation. Specifically, we plan on extending our set of relationship descriptors to include some variations to the relatively coarse relationship types currently described. Part of the sharing advisor's functionality will be an inference mechanism much like the one provided by LOOM. In addition to resolving object relationships, the advisor will maintain the semantic dictionary (i.e., the ontology package and the concept hierarchy), two activities that have not been implemented in our prototypes here.

Another area of research that has not been covered in this dissertation involves local updates of shared objects. In our approach, all imported objects are copies of their remote counterparts and no physical connection between the copy(ies) and the original(s) exist. Although this approach is useful in the sense that it protects the owner of shared information from accidental or unwanted updates to his/her data (e.g., updates to customer reservation records), we can envision scenarios where the propagation of updates is useful (e.g., in FOTA, travel agencies might want to be informed whenever hotel prices have changed).

Finally, we would like to investigate access control mechanisms that go beyond the protection provided by the export schemas (in our current approach, each federation component shares his/her complete export schema with every other component).

# Reference List


[1] H. Afsarmanesh and D. McLeod. The 3DIS: An Extensible, Object-Oriented Information Management Environment. *ACM Transactions on Office Information Systems*, 7:339–377, October 1989.

[2] Y. Arens, C. Y. Chee, C. Hsu, and C. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. Technical Report ISI-RR-93-308, USC/Information Sciences Institute, March 1993.

[3] M. Atkinson, et al. The Object-Oriented Database System Manifesto. In *Proceedings of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*. Kyoto, Japan, December 1989.

[4] C. Batini and M. Lenzerini. A Methodology for Data Schema Integration in the Entity Relationship Model. *IEEE Transactions on Software Engineering*, 10(6):650–664, 1984.

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

[6] E. Bertino, G. Pelagatti, and L. Sbattella. An Object-Oriented Approach to the Interconnection of Heterogenous Databases. In *Proceedings of the Workshop on Heterogenous Databases*. NSF, December 1989.

[7] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] D. G. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):10–29, 1977.

[9] Y. Breibart, A. Silberschatz, and G. Thompson. Reliable Transaction Management in a Multidatabase System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224. ACM SIGMOD, May 1990.

[10] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw Hill, 1984.

[11] T. Connors and P. Lyngbaek. Providing Uniform Access to Heterogeneous Information Bases. In *Proceedings of 2nd International Conference on Object Oriented Database Systems*. Bad Münster am Stein Ebernburg, Germany, September 1988.

[12] DARPA. Case-based Reasoning from DARPA: Machine Learning Program Plan. In *Proceedings of the 1989 DARPA Case-Based Reasoning Workshop*. DARPA, Washington, D.C., 1989.

[13] U. Dayal and H. Hwang. View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases. *IEEE Transactions on Software Engineering*, 10(6):628–644, 1984.

[14] D. Fang, J. Hammer, and D. McLeod. An Approach to Behavior Sharing in Federated Database Systems. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 334–346. Morgan Kaufman, 1993.

[15] D. Fang, J. Hammer, D. McLeod, and A. Si. Remote-Exchange: An Approach to Controlled Sharing among Autonomous, Heterogenous Database Systems. In *Proceedings of the IEEE Spring Compcon*. IEEE, San Francisco, February 1991.

[16] D. Fang and D. McLeod. A Testbed and Mechanism for Object-Based Sharing in Federated Database Systems. Technical Report USC-CS-92-507, Computer Science Department, University of Southern California, Los Angeles CA 90089-0781, February 1992.

[17] P. Fankhauser and E. Neuhold. Knowledge Based Integration of Heterogeneous Databases. Technical report, Technische Hochschule Darmstadt, 1992.

[18] A. Ferrier and C. Stangret. Heterogeneity in the Distributed Database Mangagement System SIRIUS-DELTA. In *Proceedings of the International Conference on Very Large Databases*. VLDB Endowment, 1983.

[19] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, T. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[20] K. Frenkel. The Human Genome Project and Informatics. *Communications of the ACM*, 34(11):41–51, 1991.

[21] S. Ghandeharizadeh, V. Choi, C. Ker, and K. Lin. Design and Implementation of the Omega Object-Based System. In *Proceedings of the Fourth Australian Database Conference*. Australian Computer Society, February 1993.

[22] T. R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

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

[24] J. Hammer, D. McLeod, and A. Si. An Intelligent System for Identifying and Integrating Non-Local Objects in Federated Database Systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 398–407. Computer Society of the IEEE, Hawaii, USA, January 1994.

[25] M. Hammer and D. McLeod. On Database Management System Architecture. In *Infotech State of the Art Report: Data Design*, volume 8 of *Infotech State of the Art Reports*, pages 177–202. Pergamon Infotech Limited, Maidenhead, United Kingdom, 1980.

[26] S. Hayne and S. Ram. Multi-User View Integration System (MUVIS): An Expert System for View Integration. In *Proceedings of the 6th International Conference on Data Engineering*. IEEE, February 1990.

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

[28] M. Huhns, N. Jacobs, T. Ksiezyk, W. Shen, M. Singh, and P. Cannata. Enterprise Information Modeling and Model Integration in Carnot. Technical Report Carnot-128-92, MCC, 1992.

[29] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[30] R. Hull and S. Widijojo. ILOG: Specificational Language for Generating OIDs. Technical Report RJ3325, ISI, February 1990.

[31] G. Jacobsen, 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, June 1988.

[32] R. Katz and N. Goodman. View Processing in Multibase – A Heterogeneous Database System. In *An Entity-Relationship Approach to Information Modelling and Analysis*, pages 259–280. ER Institute, 1981.

[33] W. Kent. The Many Forms of a Single Fact. In *Proceedings of the IEEE Spring Compcon*. IEEE, February 1989.

[34] W. Kent. Solving Domain Mismatch Problems with an Object-Oriented Database Programming Language. In *Proceedings of the International Conference on Very Large Databases*, pages 147–160. IEEE, September 1991.

[35] W. Kent, R. Ahmed, J. Albert, M. Ketabchi, and M. Shan. Object Identification in Multidatabase Systems. Technical report, Hewlett-Packard Laboratories, 1992.

[36] W. Kim, N. Ballou, H.-T. Chou, J.F. Garza, and D. Woelk. Integrating an Object-Oriented Programming System with a Database System. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–152. ACM, September 1988.

[37] W. Kim, J. Banerjee, H. T. Chou, J. F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 118–125, 1987.

[38] W. Kim, I. Choi, S. Gala, and M. Scheevel. On Resolving Schematic Heterogeneity in Multidatabase Systems. *Distributed and Parallel Databases*, 1(3):251–279, July 1993.

[39] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge Data Engineering*, 2(1):109–117, March 1990.

[40] J. Larson, S.B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence and its Applications to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, April 1989.

[41] C. Lecluse, P. Richard, and F. Velez. $O_2$, an Object-Oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, Ill., June 1988. ACM SIGMOD.

[42] Q. Li and D. McLeod. Object Flavor Evolution in an Object-Oriented Database System. In *Proceedings of the Conference on Office Information System*. ACM, March 1988.

[43] V. Linnemann, et al. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *Proceedings of the International Conference on Very Large Databases*, pages 294–305, Los Angeles, Ca., 1988.

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

[45] W. Litwin. An Overview of the Multidatabase System MRSDM. In *Proc. of the ACM National Conference*, pages 495–504. ACM, October 1985.

[46] W. Litwin and A. Abdellatif. Multidatabase Interoperability. *IEEE Computer*, 19(12):10–18, December 1986.

[47] P. Lyngbaek and D. McLeod. Object Management in Distributed Information Systems. *ACM Transactions on Office Information Systems*, 2(2):96–122, April 1984.

[48] R. MacGregor. A Deductive Pattern Matcher. In *Proceedings of the AAAI-89 National Conference on Artifical Intelligence*. AAAI, St. Paul, Minn., August 1988.

[49] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482. ACM, 1986.

[50] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed Object Management. *International Journal of Intelligent & Cooperative Information Systems*, 1(1):5–42, 1992.

[51] D. McLeod. Beyond Object Databases. In W. Stucky and A. Oberweis, editors, *Datenbanksysteme in Büro, Technik, und Wissenschaft*. Springer Verlag, 1993.

[52] D. McLeod, D. Fang, and J. Hammer. The Identification and Resolution of Semantic Heterogeneity. In *Proceedings of International Workshop on Interoperability in Multidatabase Systems*. Kyoto, Japan, April 1991.

[53] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7), July 1987.

[54] A. Motro and P. Buneman. Constructing Superviews. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Ann Arbor, Mich., April 1981. ACM SIGMOD.

[55] S. B. Navathe, R. ElMasri, and J. Larson. Integrating User Views in Database Design. *IEEE Computer*, 19(1):50–62, 1986.

[56] R. Neches. Approaches to Cognitive Issues in Knowledge Acquisition within the SHELTER Knowledge Base Development Environment. In *Proceedings of the AAAI Spring Symposium on Cognitive Issues in Knowledge Acquisition*. AAAI, Stanford, California, March 1992.

[57] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report Number 91.12.1, Draft 10, The Object Management Group, 1992.

[58] M. Papazoglou, S. Laufmann, and T. Sellis. An Organizational Framework for Cooperating Intelligent Information Systems. *International Journal of Intelligent & Cooperative Information Systems*, 1(1):169–202, 1992.

[59] B. Raphael. A Computer Program for Semantic Information Retrieval. In M. Minsky, editor, *Semantic Information Processing*. MIT Press, Cambridge, Mass., 1968.

[60] M. Rusinkiewitz, R. Elmasri, B. Czejdo, D. Georakopoulous, G. Karabatis, A. Jamoussi, K. 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. IEEE, May 1989.

[61] A. Savasere, A. Sheth, S. Gala, S. Navathe, and H. Marcus. On Applying Classification to Schema Integration. In *Proceedings of IEEE 1st International Workshop on Interoperability in Multidatabase Systems*, pages 258–261. Kyoto, Japan, April 1991.

[62] M.H. Scholl, C. Laasch, and M. Tresch. Update Views in Object-Oriented Databases. In *Proceedings of the 2nd International Conference on Distributed Object-Oriented Databases*, December 1991.

[63] A. Sheth and V. Kashyap. So Far (Schematically) yet So Near (Semantically). In D.K. Hsiao, E.J. Neuhold, and R. Sacks-Davis, editors, *Interoperable Database Systems (DS-5) (A-25)*, pages 283–312. Elsevier Science Publisher B.V. (North-Holland), 1993.

[64] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[65] A. Sheth, J. Larson, A. Cornelio, and S. B. Navathe. A Tool for Integrating Conceptual Schemata and User Views. In *Proceedings of the 4th International Conference on Data Engineering*, pages 176–183. IEEE, February 1988.

[66] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 2(3):140–173, March 1981.

[67] M. Siegel and S. E. Madnick. A Metadata Approach to Resolving Semantic Conflicts. In *Proc. of the 17th International Conference on Very Large Databases*, pages 133–145. IEEE, September 1991. Barcelona.

[68] A. Silberschatz, M. Stonebraker, and J. Ullman. Database Systems: Achievements and Opportunities. *ACM Sigmod Record*, 19(4):6–23, December 1990.

[69] J. Smith, P. Bernstein, U. Dayal, N. Goodman, T. Landers, K. Lin, and E. Wong. Multibase: Integrating Heterogeneous Distributed Database Systems. In *Proceedings of the National Computer Conference*, pages 487–499. AFIPS, June 1981.

[70] R. Strom and S. Yemini. The NIL Distributed Systems Programming Language: A Status Report. *ACM Sigplan Notices*, 20(5):36–43, May 1985.

[71] T. Templeton, et al. Mermaid: A Front–End to Distributed Heterogenous Databases. In *Proceedings Intl' Conf. on Data Engineering*, pages 695–708. IEEE, 1987.

[72] V. Ventrone and S. Heiler. A Practical Approach for Dealing with Semantic Heterogeneity in Federated Database Systems. Technical report, The MITRE Corporation, October 1993.

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

# Appendix A

# Remote-Sharing Language *RSL*

The following is a detailed description of the functionality that is provided by the remote sharing language commands used in CODM. Components may use their own data models as long as they adhere to the specifications set forth by CODM and provide a sharing functionality that is compatible with that of *RSL* described below. We realize that some of the *RSL* commands are not very "user-friendly." *HasInstances*, for example, returns a set of OID's and thus additional set operators are needed before one can access and manipulate individual instances. Therefore, we encourage that most federation components will hide the *RSL* communication protocol layer from users by adding their own interface, similar to the graphical interface provide by our unification tool (see Fig. 9.2).

---

**ShowMeta** type-level command

**Syntax:** *ShowMeta(d:DB)*

**Description:** Lists all the types that belong to the export schema of database $d$.

**Example:** `ShowMeta(Travel-Agency-`$D$`)` $\longrightarrow$ `Accommodations`
`Airline`
`Entertainment`
`Flight`
`Rental-Car`

---

**HasProperties** function-level command

**Syntax:** *HasProperties(d:DB, t:Type)*

**Description:** Lists all stored functions that are defined on type $t$ in database $d$.

**Example:** `HasProperties(Travel-Agency-`$D$`, Accommodations)`
$\longrightarrow$ `Name`
`Fax-Number`
`Price`

---

**HasValueType** type-level command

**Syntax:** *HasValueType(d:DB, f:Func)*

**Description:** Returns the value type for stored function $f$ in database $d$.

**Example:** `HasValueType(Travel-Agency-`$D$`, Name)` $\longrightarrow$ `STRING`

## HasMethods

function-level command

**Syntax:** *HasMethods(d:DB, t:Type)*

**Description:** Lists all computed functions that are defined on type $t$ in database $d$.

**Example:** `HasMethods(Travel-Agency-`$D$`, Accommodations)`

$\longrightarrow$ `SendMessage`
`MakeReservation`

---

## HasInstances

instance-level command

**Syntax:** *HasInstances(d:DB, t:Type)*

**Description:** Returns a list of all instance OID's of user-defined type $t$ in database $d$.

**Example:** `HasInstances(Travel-Agency-`$D$`, Accommodations)`

$\longrightarrow$ `10001`
`10002`
`10003`
`10004`

---

## HasValue

instance-level command

**Syntax:** *HasValue(d:DB, i:InstOID, f:Func)*

**Description:** Returns the value of the stored function $f$ on instance $i$ in database $d$.

**Example:** `HasValue(Travel-Agency-`$A$`, 10003, Name)`

$\longrightarrow$ `Holiday Inn`

**HasDirectSubtypes** type-level command

**Syntax:** *HasDirectSubTypes(d:DB, t:Type)*

**Description:** Returns a list of all DIRECT subtypes of type $t$ in database $d$.

**Example:** `HasDirectSubtypes(Travel-Agency-`$D$`, Accommodations)`
$\longrightarrow$ `Hotels`
`Private Accommodations`

---

**HasDirectSupertype** type-level command

**Syntax:** *HasDirectSuperType(d:DB, t:Type)*

**Description:** Returns the DIRECT supertype of type $t$ in database $d$.

**Example:** `HasDirectSupertype(Travel-Agency-`$D$`, Accommodations)`
$\longrightarrow$ `Travel-Info`

---

**ImportMeta** type-level command

**Syntax:** *ImportMeta(d1:DB, t1:Type, d2:DB, t2:type, r:Rel)*

**Description:** Imports type $t1$ from database $d1$ into $d2$ and relates remote type $t1$ to local type $t2$ using relationship $r$.

**Example:** `ImportMeta(Travel-Agency-`$D$`, Private Accommodations,`
`Travel-Agency-`$A$`, Bed&Breakfast, SUBTYPE)`

---

**ImportInstance** instance-level command

**Syntax:** *ImportInstance(d1:DB, t1:Type, d2:DB, t2:type, i:InstOID)*

**Description:** Imports instance $i$ of type $t1$ from remote database $d1$ into type $t2$ of local database $d2$

**Example:** `ImportInstance(Travel-Agency-`$D$`, Private Accommodations, Travel-Agency-`$A$`, Bed&Breakfast, 10003)`

---

### ImportMethod function-level command

**Syntax:** *ImportComputedFunction(d1:DB, f:Func, d2:DB)*

**Description:** Imports computed function $f$ from remote database $d1$ into local database $d2$.

**Example:** `ImportMethod(Travel-Agency-`$D$`, SendMessage, Travel-Agency-`$A$`)`

---

# Appendix B

# SHELTER Prototype

The following is a listing of the LOOM commands used to create a sample federation of three travel agencies in the SHELTER development environment. The three travel agencies are named $A$, $B$, and $E$ to express the similarities with the example used in earlier chapters of this dissertation. Each component knowledge base contains the schema, the data, as well as the local lexicon for the underlying travel agency. The last part of this appendix contains a listing of the semantic dictionary with the ontology and relationship descriptors that were used in this prototype implementation.

## B.1 Semantic Dictionary

The following is a listing of the SHELTER script file that was used to create the semantic dictionary.

```
;;;-*- Mode: LISP; Syntax: Common-Lisp; Package: TRAVEL; Base: 10.-*-

(in-package "TRAVEL")

;;; Knowledge Base:   SEMANTIC DICTIONARY
;;; Last Saved On:    02/02/94 10:50:42

(defparameter savedKnowledgeBase loom:*knowledge-base*)
(defparameter savedWorld loom:*world*)

(loom:change-kb 'TRAVEL:SEMANTIC-DICTIONARY)

(eval-when #+:CLTL2(:execute :load-toplevel :compile-toplevel)
          #-:CLTL2(load eval compile)

;;; Relationship Descriptors

;;; Relationships Between Concepts

(loom:defrelation Is_Identical_To
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept
  :characteristics (:symmetric))

(loom:defrelation Is_Equal_To
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept
  :characteristics (:symmetric))

(loom:defrelation Is_Compatible_With
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept
  :characteristics (:symmetric))
```

```
(loom:defrelation Is_Kind_Of
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Assoc_With
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Is_Collection_Of
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Is_Instance_Of
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Is_Common_Among
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Is_Feature_Of
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)

(loom:defrelation Has
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Concept)


;;; Relationships Between a Relation and a Concept

(loom:defrelation Is_REqual_To
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Concept
  :domain Loom:Relation)
```

```
(loom:defrelation RType_Of
  :is (:and Loom:Binary-Tuple :primitive)
  :range Loom:Relation
  :domain Loom:Concept)

;;; GPO

(loom:defconcept Person
  :is (:and :primitive))
(loom:defconcept Thing
  :is (:and :primitive))
(loom:defconcept Name
  :is (:and :primitive))
(loom:defconcept Number
  :is (:and :primitive))



;;; SPO-Travel

(loom:defconcept Customer
  :is (:and :primitive))
(loom:defconcept Package
  :is (:and :primitive))
(loom:defconcept Schedule
  :is (:and :primitive))
(loom:defconcept Booking
  :is (:and :primitive))
(loom:defconcept Arrival
  :is (:and :primitive))
(loom:defconcept Departure
  :is (:and :primitive))
(loom:defconcept Price
  :is (:and :primitive))
(loom:defconcept Ticket
  :is (:and :primitive))
(loom:defconcept Destination
  :is (:and :primitive))
(loom:defconcept Seat
  :is (:and :primitive))
(loom:defconcept Activity
```

```
  :is (:and :primitive))
(loom:defconcept Restriction
  :is (:and :primitive))


;;; SPO-Travel-Air

(loom:defconcept Flight
  :is (:and :primitive))
(loom:defconcept Airline
  :is (:and :primitive))
(loom:defconcept Fare
  :is (:and :primitive))
(loom:defconcept First
  :is (:and :primitive))
(loom:defconcept Business
  :is (:and :primitive))
(loom:defconcept Coach
  :is (:and :primitive))
(loom:defconcept Service
  :is (:and :primitive))
(loom:defconcept Connection
  :is (:and :primitive))
(loom:defconcept Class
  :is (:and :primitive))


;;; SPO-Travel-Hotel

(loom:defconcept Hotel
  :is (:and :primitive))
(loom:defconcept Room
  :is (:and :primitive))
(loom:defconcept Category
  :is (:and :primitive))
(loom:defconcept One-Star
  :is (:and :primitive))
(loom:defconcept Two-Star
  :is (:and :primitive))
(loom:defconcept Three-Star
  :is (:and :primitive))
```

```
;;; SPO-Travel-Train

(loom:defconcept Train
  :is (:and :primitive))
(loom:defconcept Car
  :is (:and :primitive))
(loom:defconcept Compartment
  :is (:and :primitive))


;;; SPO-Travel-CarRental

(loom:defconcept Rental_Car
  :is (:and :primitive))
(loom:defconcept Insurance
  :is (:and :primitive))
(loom:defconcept Compact
  :is (:and :primitive))
(loom:defconcept Large
  :is (:and :primitive))
(loom:defconcept Van
  :is (:and :primitive))


;;; SPO-Travel-Cruise

(loom:defconcept Pleasure-Cruise
  :is (:and :primitive))
(loom:defconcept Boat
  :is (:and :primitive))
(loom:defconcept Port
  :is (:and :primitive))
(loom:defconcept Cabin
  :is (:and :primitive))
(loom:defconcept Sightseeing
  :is (:and :primitive))
(loom:defconcept Entertainment
  :is (:and :primitive))


(loom:finalize-definitions)
```

```
)    ; END EVAL-WHEN


(loom:change-kb savedKnowledgeBase)
(when savedWorld (loom:change-world savedWorld))
```

## B.2 Travel Agency *A*

The following is a listing of the SHELTER script file that was used to create a conceptual schema for travel agency $A$.

```
;;;-*- Mode: LISP; Syntax: Common-Lisp; Package: TRAVEL; Base: 10.-*-

(in-package "TRAVEL")

;;; Knowledge Base:   TRAVEL:TRAVEL-AGENCY-KB-1
;;; Last Saved On:    09/01/93 10:03:48

(defparameter savedKnowledgeBase loom:*knowledge-base*)
(defparameter savedWorld loom:*world*)

(loom:change-kb 'TRAVEL:TRAVEL-AGENCY-KB-1)

(eval-when (LOAD EVAL COMPILE)

;;; Local Schema for Travel Agency A

(loom:defrelation Airline
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation End
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Has-Cost
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Has-Total-Cost
  :is :Primitive
  :kb Travel-Agency-Kb-1)
```

```
(loom:defrelation Has-Train-Ride
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Is-Owned-By
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Lodging
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Makes
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Owned-By
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Start
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defrelation Train-Ride
  :is :Primitive
  :kb Travel-Agency-Kb-1)

(loom:defconcept Hotel
  :is (:and :primitive
            (:and (:at-least 1 (:and Owned-By (:range Owner)))
                  (:at-most 999 (:and Owned-By (:range Owner))))
            (:and (:at-least 1 (:and Has-Cost (:range Cost)))
                  (:at-most 999 (:and Has-Cost (:range Cost))))))
(loom:defconcept Bed&Breakfast
  :is (:and :primitive Hotel))
(loom:defconcept Cost
  :is (:and :primitive Integer))
(loom:defconcept Customer)
(loom:defconcept Date
  :is (:and :primitive String))
(loom:defconcept Flight
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defconcept Owner
  :is :Primitive
  :kb Travel-Agency-Kb-1)
(loom:defconcept Reservation
```

```
  :is (:and :primitive
            (:all Is-Owned-By Customer)
            (:exactly 1 Is-Owned-By)
            (:at-most 999 (:and Train-Ride (:range Train)))
            (:at-most 999 (:and Airline (:range Flight)))
            (:at-most 999 (:and Lodging (:range Hotel)))
            (:all Has-Total-Cost Cost)
            (:exactly 1 Has-Total-Cost)
            (:the Start Date)
            (:the End Date)))
(loom:defconcept Resort
  :is (:and :primitive Hotel))
(loom:defconcept Train
  :is :Primitive
  :kb Travel-Agency-Kb-1)


;;; Local Lexicon

(loom:tell (semantic-dictionary^Is_Equal_To
                 semantic-dictionary^Booking Reservation))
(loom:tell (semantic-dictionary^Is_Equal_To
                 semantic-dictionary^Price Cost))


(loom:tell (semantic-dictionary^Is_Kind_Of
                 Resort semantic-dictionary^Hotel))
(loom:tell (semantic-dictionary^Is_Kind_Of
                 Bed&Breakfast semantic-dictionary^Hotel))
(loom:tell (semantic-dictionary^Is_Kind_Of
                 Owner semantic-dictionary^Person))

(loom:tell (semantic-dictionary^CollectionOf
                 Lodgings semantic-dictionary^Room))

(loom:tell (semantic-dictionary^Feature
              Bed&Breakfast semantic-dictionary^One-Star))

(loom:tell (semantic-dictionary^Has Hotel Owner))

(loom:finalize-definitions)
```

```
)  ; END EVAL-WHEN


(loom:change-kb savedKnowledgeBase)
(when savedWorld (loom:change-world savedWorld))
```

## B.3 Travel Agency *B*

The following is a listing of the SHELTER script file that was used to create a conceptual schema for travel agency *B*.

```
;;;-*- Mode: LISP; Syntax: Common-Lisp; Package: TRAVEL; Base: 10.-*-

(in-package "TRAVEL")

;;; Knowledge Base:   TRAVEL:TRAVEL-AGENCY-KB-3
;;; Last Saved On:    09/08/93 10:40:46

(defparameter savedKnowledgeBase loom:*knowledge-base*)
(defparameter savedWorld loom:*world*)

(loom:change-kb 'TRAVEL:TRAVEL-AGENCY-KB-3)

(eval-when (LOAD EVAL COMPILE)

;;; Local Schema for Travel Agency B

(loom:defrelation Arrival_Date
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Arrives_In
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Carrier
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Departs_From
  :is :Primitive
```

```
  :kb Travel-Agency-Kb-3)
(loom:defrelation Departure_Date
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Destination
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Duration
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Ends_On
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation End_Date
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Has-Last_Name
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Has_Address
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Has_Contact
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Has_First_Name
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation In_Charge_Of
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Located_In
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Package-Id
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Ss#
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defrelation Starts_On
  :is :Primitive
```

```
  :kb Travel-Agency-Kb-3)
(loom:defrelation Start_Date
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defconcept Date
  :is (:and :primitive  String))
(loom:defconcept Air-Trips
  :is (:and :primitive
          (:the Arrival_Date Date)
          (:the Departure_Date Date)))
(loom:defconcept Contact
  :is (:and :primitive Person
          (:and (:at-least 0 (:and In_Charge_Of (:range Package)))
              (:at-most 999 (:and In_Charge_Of (:range Package))))))
(loom:defconcept Cruises
  :is (:and :primitive Package
          (:and (:at-least 1 (:and Arrives_In (:range Port)))
                (:at-most 999 (:and Arrives_In (:range Port))))
          (:and (:at-least 1 (:and Departs_From (:range Port)))
                (:at-most 9999 (:and Departs_From (:range Port))))))
(loom:defconcept Exotic
  :is (:and :primitive Package
          (:the Start_Date Date)
          (:the End_Date Date)))
(loom:defconcept Hotel-Air
  :is (:and :primitive Package
          (:and (:at-least 1 (:and Destination (:range Hotels)))
                (:at-most 999 (:and Destination (:range Hotels))))
          (:the Carrier Air-Trips)))
(loom:defconcept Hotels
  :is (:and :primitive
          (:the Has_Address String)))
(loom:defconcept Package
  :is (:and :primitive
          (:all Duration Time)
          (:exactly 1 Duration)
          (:all Package-Id Integer)
          (:exactly 1 Package-Id)))
(loom:defconcept Person
  :is (:and :primitive
          (:the Has_Address String)
          (:the Ss# String)
```

```
          (:the Has_First_Name String)
          (:the Has-Last_Name String)))
(loom:defconcept Port
  :is (:and :primitive
          (:the Located_In String)))
(loom:defconcept Schedule
  :is :Primitive
  :kb Travel-Agency-Kb-3)
(loom:defconcept Time
  :is (:and :primitive String))

;;; Local Lexicon

(loom:tell (semantic-dictionary^Is_Equal_To
                              semantic-dictionary^Customer Contact))

(loom:tell (semantic-dictionary^Is_Compatible_With
                              semantic-dictionary^Booking Package))
(loom:tell (semantic-dictionary^Has Package
                              semantic-dictionary^Restriction))

(loom:tell (semantic-dictionary^Is_Compatible_With
                    semantic-dictionary^Booking Hotel-Air-Package))

(loom:tell (semantic-dictionary^Is_Compatible_With
                      semantic-dictionary^Pleasure-Cruise Cruises))

(loom:tell (semantic-dictionary^Is_Compatible_With
                                 semantic-dictionary^Hotel Exotic))
(loom:tell (semantic-dictionary^Is_Feature_Of
                            semantic-dictionary^Three-Star Exotic))


(loom:finalize-definitions)


)  ; END EVAL-WHEN

(loom:change-kb savedKnowledgeBase)
(when savedWorld (loom:change-world savedWorld))
```

## B.4 Travel Agency $E$

The following is a listing of the SHELTER script file that was used to create a conceptual schema for travel agency $E$.

```
;;;-*- Mode: LISP; Syntax: Common-Lisp; Package: TRAVEL; Base: 10.-*-

(in-package "TRAVEL")

;;; Knowledge Base:   TRAVEL:TRAVEL-AGENCY-KB-2
;;; Last Saved On:    09/03/93 12:15:57

(defparameter savedKnowledgeBase loom:*knowledge-base*)
(defparameter savedWorld loom:*world*)

(loom:change-kb 'TRAVEL:TRAVEL-AGENCY-KB-2)

(eval-when (LOAD EVAL COMPILE)

;;; Local Schema for Travel Agency E

(loom:defrelation Belongs_To
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Client_Id
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation End_Date
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Flight
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Has_Itinerary
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Has_Name
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Itinerary_Id
  :is :Primitive
  :kb Travel-Agency-Kb-2)
```

```
(loom:defrelation Rental_Car
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Start_Date
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defrelation Stays_In
  :is :Primitive
  :kb Travel-Agency-Kb-2)
(loom:defconcept Airlines
  :is (:and :primitive Things_To_Do))
(loom:defconcept Clients
  :is (:and :primitive
         (:the Has_Name String)
         (:the Client_Id Integer)))
(loom:defconcept Date
  :is (:and :primitive String))
(loom:defconcept Itinerary
  :is (:and :primitive
         (:all Itinerary_Id Integer)
         (:exactly 1 Itinerary_Id)
         (:all Belongs_To Clients)
         (:exactly 1 Belongs_To)
         (:and (:at-least 0 (:and Flight (:range Airlines)))
               (:at-most 999 (:and Flight (:range Airlines))))
         (:and (:at-least 0 (:and Rental_Car (:range Rental_Cars)))
               (:at-most 999 (:and Rental_Car (:range Rental_Cars))))
         (:and (:at-least 0 (:and Stays_In (:range Lodgings)))
               (:at-most 999 (:and Stays_In (:range Lodgings))))))
(loom:defconcept Lodgings
  :is (:and :primitive Things_To_Do))
(loom:defconcept Rental_Cars
  :is (:and :primitive Things_To_Do))
(loom:defconcept Things_To_Do
  :is (:and :primitive
         (:the Start_Date Date)
         (:the End_Date Date)))

;;; Local Lexicon

(loom:tell (semantic-dictionary^Is_Equal_To
                             semantic-dictionary^Flight Airlines))
```

```
(loom:tell (semantic-dictionary^Is_Equal_To
                            semantic-dictionary^Customer Clients))
(loom:tell (semantic-dictionary^Is_Equal_To
                       semantic-dictionary^Activity Things_To_Do))
(loom:tell (semantic-dictionary^Is_Equal_To
                      semantic-dictionary^Rental_Car Rental_Cars))
(loom:tell (semantic-dictionary^Is_Equal_To
                              semantic-dictionary^Hotel Lodgings))

(loom:tell (semantic-dictionary^Is_Compatible_With
                          semantic-dictionary^Schedule Itinerary))

(loom:tell (semantic-dictionary^CollectionOf Lodgings
                                        semantic-dictionary^Room))

(loom:tell (semantic-dictionary^Feature
                           Lodgings semantic-dictionary^One-Star))

(loom:tell (semantic-dictionary^Has Itinerary Airlines))
(loom:tell (semantic-dictionary^Has Itinerary Lodgings))
(loom:tell (semantic-dictionary^Has Itinerary Rental_Cars))


(loom:finalize-definitions)


)  ; END EVAL-WHEN


(loom:change-kb savedKnowledgeBase)
(when savedWorld (loom:change-world savedWorld))
```