ODL-M

A Mapping Language for Schema Integration in Object-Oriented Multidatabase Systems

Steinar A. Kindingstad

Master thesis submitted to the Cand. Scient degree in Informatics
at the Department of Informatics, University of Oslo.

August 1996

Preface

This is my masters thesis, submitted in partial fulfillment of the requirements for the degree Cand. Scient. in Informatics at the Department of Informatics, University of Oslo(UiO). The thesis has been a project at the Department of Informatics, SINTEF[1], Oslo, and the work has been done partly at SINTEF and partly at the Department of Informatics, UiO.

I would like to thank my advisors, Arne-Jørgen Berre and Bjørn Skjellaug for their guidance and patience with me. Our meetings kept me on a straight pace towards the goal. Also, thanks goes to Ragnar Normann for his final remarks on this work.

I would also like to thank my fellow students at the Department of Informatics for our social community, and especially Jan Bjelde whom I had useful discussions on most subjects.

Oslo, Steinar A. Kindingstad
August 1996

[1]The Foundation for Scientific & Industrial Research

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we will state the purpose and goal of this thesis and how we intend to achieve our goal. A short introduction to the problem focus, schema integration, will be given before presenting a case that will be used throughout the thesis. From the case we will identify conflict groups that we want to investigate further. First we give some background history of our problem area.

1.1 Background

More and more of our society is becoming dependent on the use of computers. The sharing and storing of information has exploded during the last decades. Over the years there has been a continuous development on the frontier of database systems. Hierarchical and network philosophies were released by the relational philosophy as the leading in this field. Later we also have seen the object-oriented databases emerge, a new way of viewing and modeling the world that has become very popular.

Computer systems are widely used in all functions of contemporary organizations. In most of these organizations, the computing environment consists of *distributed*, *heterogeneous* and *autonomous*[1] hardware and software systems. Although no provision for a possible future integration was made during the development of these systems, there is an increasing need for technology to support the cooperation of the provided services and resources for handling more complex applications.

The requirements for cooperation among distinct systems can be met at two levels – a lower level and a higher level [MHG+92]. The ability of systems to communicate and exchange information is referred to as *interconnectivity*. At a higher level, the systems are not only able to communicate, but also be able to interact and jointly execute tasks. This ability is referred to as *interoperability*.

Unfortunately, one might say, the system over all systems, that will solve all needs, has yet to be developed and it most probably never will be found for obvious reasons. Therefore our database world as it is today has several different database systems to cope with. As with a lot of other areas it is often the vendors who control, or at least influence,

[1] These terms will be closer defined in later chapters

what we are buying and thereby which systems we have in use. But different vendors have been successful in different areas and as a result companies have a lot of different systems based on different characteristics, e.g. data models. Also within companies database solutions can be diverse. This raises new needs in the companies: How can we obtain unified views over some existing database systems in the company? Applications and users would like a unified view over relevant database systems in the company and be able to manipulate them through a single logical database system level rather than to operate on them locally through multiple and heterogeneous database systems. *Multidatabases* or *federated databases* are an approach to solve this problem. Multidatabases are discussed in more detail in chapter 2. Multidatabases do however have many problems that still have to be investigated. The focus in this thesis is the problem of schema integration which will be given a short introduction in section 1.3.

1.2 Purpose of the Thesis/Problem Specification

The overall theme of this thesis is multidatabases. The thesis will give an overview of the basic concepts of multidatabases as a basis for understanding the problem areas to be discussed. The main focus of the thesis is reflected in the title of this work; "*ODL-M — A Mapping Language for Schema Integration in Object-Oriented Multidatabases*". First of all, we will concentrate on object-oriented multidatabases in this discussion. This means that the multidatabase systems considered here take advantage of the object-oriented paradigm that has become more and more popular recently. Further, within the context of object-oriented multidatabases, we will study more closely the problem area of *integrating schemas* obtaining a schema that represents the union of the concepts from the integrated schemas. The schemas in our context are *heterogeneous*. The term *heterogeneous* is understood as that the involved schemas are modeled in different data models, e.g. the network model or relational model, and also such that the schemas are designed independently and therefore have discrepancies not only in data model and structure but also in the semantics they express, e.g. the perceptions of the modeled reality can be different. Furthermore we will define and develop a mapping language, which we will call ODL-M, for the purpose of supporting schema integration in object-oriented multidatabases.

This work will give an overview of the problems within schema integration and also suggest a method for a solution of the problems in focus, namely resolving specific schema conflicts. We will come back to what kind of schema conflicts we are discussing here. It is not intended to give a full framework and detailed solution in this thesis since this would go beyond the scope and time bounds of this work. Instead we will try to extend existing work. In the field of object-oriented databases the ODMG-93 database standard [Cat94] has recently emerged from a group of database system vendors. This standard will be used as a basis for our proposals.

We believe that the problems of schema integration is not yet fully understood, although there has been done a lot of work on this area. Therefore this thesis will also try to give a summarizing status quo of the research in this field.

1.2.1 Goal

This thesis is meant to give the reader an understanding of the concepts of multidatabases. Herein lies a basic architecture of such a system, the problems that arise and suggested solutions to problems described in the literature. More specifically this work will give an understanding of the complexity of schema integration problems in multidatabase systems, what characterizes them and also what can be done to resolve the problems. To organize our effort we will define requirements where we find it necessary and as far as possible meet our defined requirements in our discussion. Further we will consider some existing systems and discuss their approach to the problem area of schema integration in multidatabase systems.

We will summarize our goals as the following:

Goal: To identify requirements for, and propose solutions to schema integration in object-oriented multidatabase systems.

I will give a suggestion to a support tool approach considered to be beneficial for handling these problems and argue why this approach is beneficial. The support tool will be a mapping language, which will be able to define mappings between classes in object-oriented schemas using the ODMG/ODL object model [Cat94] as a basis.

1.2.2 Methodology

It is assumed that the reader is familiar with database technology in general and todays commonly used standards. The standard ODMG-93 [Cat94] will be described later since it will be used as a basis for our proposed solutions.

With reference to the goals of this thesis this work will introduce the reader to multidatabase systems based on a general knowledge of database concepts. Multidatabases incorporate a large portion of knowledge drawn from several parts of computer science research and this work will in no way cover all aspects of this topic as it would be far to exhaustive. However there will be given a general overview of commonly accepted basic concepts of multidatabase systems. The focus of this work is schema integration so we will give a more detailed description of this topic and try to enlighten what has been experienced as important research areas and future goals on the subject.

To guide the reader's understanding of the problems discussed, a case will be designed and used throughout this thesis. The case will hopefully ease the understanding of some complicated descriptions of schema discrepancy problems, which we will describe closer in the following chapters, and also demonstrate how our suggested solutions will work. As an aid to the schema integration process a mapping language will be developed and defined. We will call it ODL-M, since it will be an extension to the ODL object model of ODMG [Cat94]. Using this mapping language the case will again be used to show the usability and benefits of the mapping language. To guide the path of discussion and evaluation a set of requirements will be defined that our solutions should meet. Our first set of general requirements on structural conflicts will be defined in section 1.5, and as we discuss in more detail the issues of schema integration, we will define two additional requirement

tables. One table will define the requirements for a canonical data model. The other will define requirements for the schema integration process itself, based on the complexity of the process and the understandability of its result. Finally the proposed mapping language and its use will be discussed along the intention of the requirements.

1.3 Schema Integration Problems in Multidatabase Systems

As mentioned, multidatabases try to give a unified view over multiple underlying heterogeneous databases or parts of them. Several problems arise in this approach. One of the basic problems is that incompatible data models are trying to communicate with each other. This is a difficult problem which can be divided into several subproblem areas. In short it can be solved by translating each database's schemas to a common data model, a *canonical model*[2], through which the communication is made. It is during this translation it can be necessary to integrate two or more schemas from different databases to one schema in a canonical data model. Examples of when this would be necessary are:

- When we encounter two schemas that actually are duplicates in two different databases and we need one common schema in the canonical model to map these two through, i.e. all modifications done to the one schema should also be done to the other.
- When we encounter two schemas that partly duplicate each other and we need one common schema to represent them in the canonical model.
- When we have two different schemas in two heterogeneous databases that represent the same real life object, but different aspects of it and we need a single schema in the overlaying system to represent the two as the real life entity they describe.

1.4 Case

As mentioned above, this thesis will have an example case that will be used to demonstrate various problems and solutions in schema integration. The same case will also be used to show how some proposed solutions can be performed. The case will be modeled in an object-oriented data model. The case is outlined with each schema's classes and attributes in figure 1.1.

The case to be used will model four university databases with corresponding schemas. University A has two schemas, `Schema 1` and `Schema 2`. University B is modeled by `Schema 3`, university C is modeled by `Schema 4`, and finally university D is modeled by `Schema 5`. The information stored therein will be faculty scholastic activities(teaching courses and advising theses) and student performance.

University A has separated its student information into two offices, represented by two schemas, one for undergraduate students and one for graduate students. The graduate student office only maintains students' thesis information and grade point averages of their course work. Also the faculty information is kept at both schemas (but some faculty

[2]The canonical model is discussed in more detail in section 2.5.

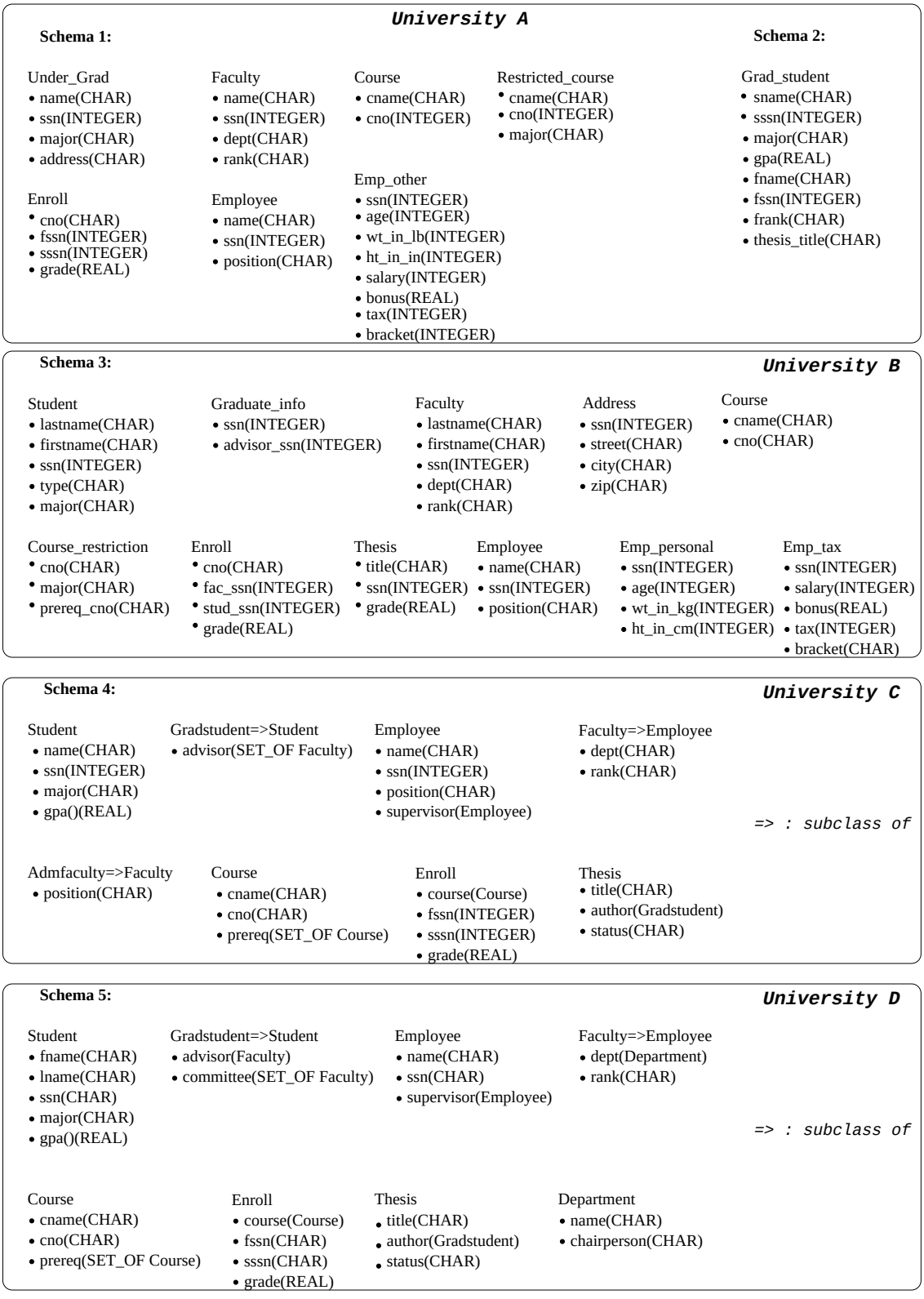

Figure 1.1: Case used in the thesis

members belong to only one schema/office DB). Universities A and B (schemas 1, 2, and 3) have relational DBMSs – their schemas reflect this fact, e.g. that no defined types are used as attribute domains and no subclasses occur. An OMT [RBP+91] representation of the schemas 1, 2 and 3 is given in figure 1.2.

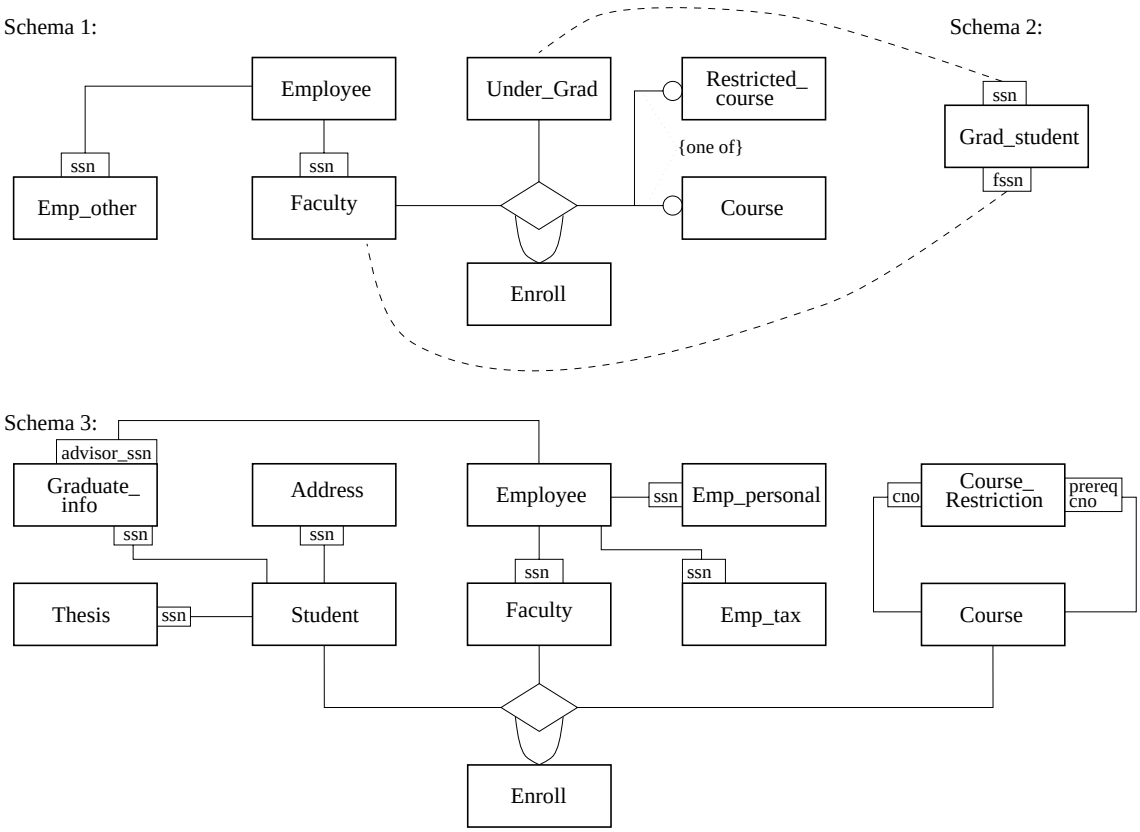

Figure 1.2: OMT model for schema 1,2 and 3 of the case

Universities C and D use object-oriented database systems to store their students' information (schema 4 and 5). The `Student` class in the OODBs of schemas 4 and 5 have the `gpa` method for computing students' grade point averages, whereas in other universities, separate queries have to be issued to compute the `gpa`. The schemas of university C and D are similar to one another, however there are some interesting differences between the class definitions of `Schema 4` and `Schema 5`. The `ssn` attribute has different domains, namely `Integer` and `String`. The `advisor` attribute of the `Gradstudent` class in `Schema 4` and `Schema 5`, respectively, has domains `SET_OF(Faculty)` and `Faculty`, while `Gradstudent` in `Schema 5` has an additional attribute `committee`. The attribute `dept` has domains `String` and `Department`.

The modeling of schemas 4 and 5 shows its difference from schemas 1, 2, and 3 in that former schemas have object-oriented constructs, subclassing and methods. This difference

will show to introduce interesting conflicts across the schemas.

An OMT representation of the schemas 4 and 5 is given in figure 1.3.

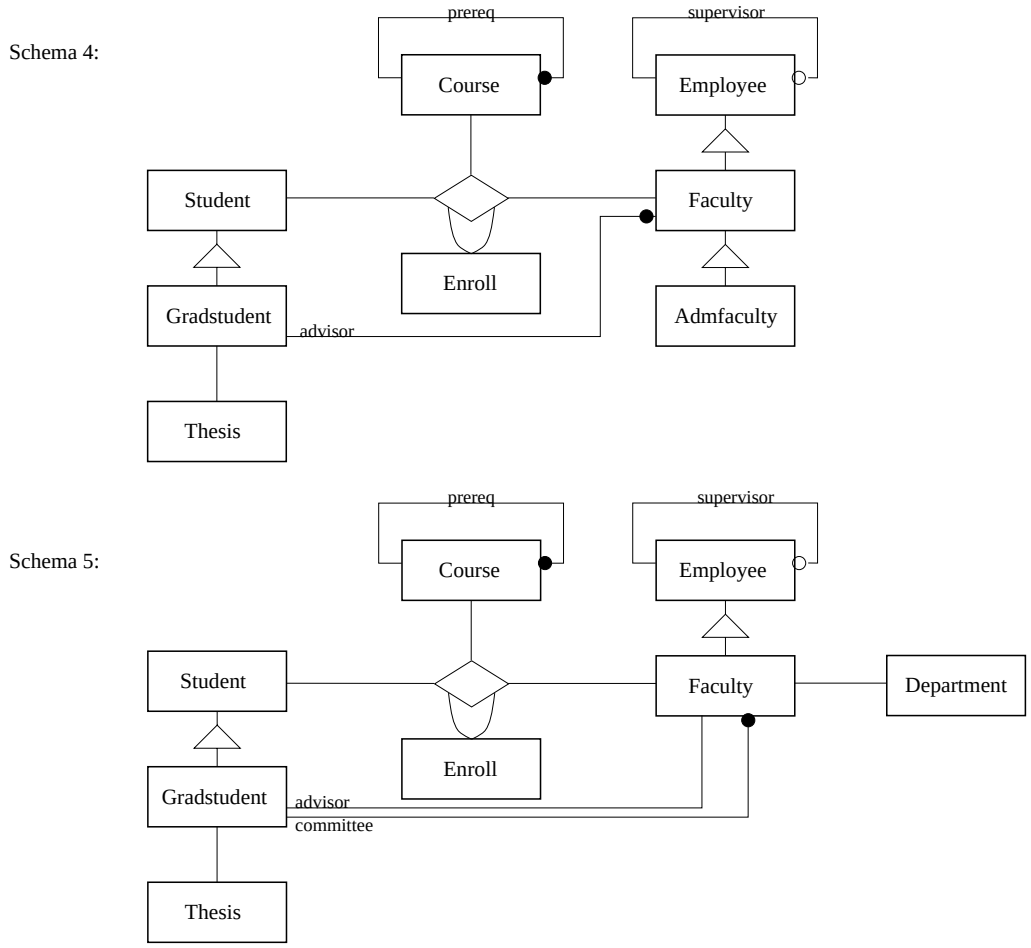

Figure 1.3: OMT model for schema 4 and 5 of the case

1.5 Requirements derived from the Case

Having presented this case, we have seen a few examples of conflicts needed to be resolved if we wish to integrate these schemas. Conflicts can arise at different levels between schemas. From the case we can encounter several conflicts by a quick overview. The classes `Grad_student` in `Schema 2` and `Gradstudent` in `Schema 4` seem to represent the same entity type with some differences, such as that `Gradstudent` in `Schema 4` is a subtype in some inheritance hierarchy while `Grad_student` in `Schema 2` is an class of its own. We can also see conflicts between corresponding attributes, such as an undergraduates name being represented by one attribute in `Schema 1` while it is split up in two attributes,

namely `lastname` and `firstname` in `Schema 3`. Another attribute conflict is between e.g. `ht_in_in` (height_in_inches) of the `Emp_other` class in `Schema 1` and `ht_in_cm` (height_in_centimeters) of the `Emp_personal` class in `Schema 3`. Further we see that in `Schema 1` we have an `address` attribute of the `Under_Grad` class, while this same information is represented as an class of its own in `Schema 3`, namely the `Address` class. Without seeing the details of the attribute values in the case we can assume that although attributes have the same name and represent the same concept, they might have different representations of the information they represent.

The examples in the previous paragraph show the type of conflicts we are interested in this thesis. The mentioned conflicts represent different levels of conflicts that can arise between schemas and we will want to classify the conflicts according to this difference in level. Thus we classify according to the detected levels of conflict in the following. We can have conflicts between two or more classes in different schemas, called class conflicts. We can have conflicts between two or more attributes between schemas, called attribute conflicts. We can have conflicts that arise between classes and attributes, as in the address example, called class vs attribute conflicts. And finally the data representation of attributes can collide with each other – we call this difference in data representation. These conflicts groups have been discussed similarly by Batini et. al [BLN86] as structural conflicts between modeling constructs and Kim et. al [KCGS95] give a classification of structural conflicts which coincides with our conflict groups.

The four groups of conflicts we have encountered will represent the main requirements in this work. For each group of conflicts we want to identify the types of conflicts that belong to this group. By identifying these individual conflict types, we hope to have covered the possibilities of structural conflicts between schemas. Since schemas not only represent pure structural representations but also inherently represent meaning of the data, or semantics as we call it, we would also like to investigate this to a degree. Semantics don't necessarily reveal themselves in the structures of the schemas, so we want to approach classifying semantic similarities and discrepancies using a separate method and see how this method can be used, if possible, in conjunction with the structural approach.

Having identified the four groups of conflicts and their characteristics, we will try to resolve the conflicts found by developing methods to support this and specifying how to use them. We would wish for the resolving methods to be as complete as possible in that they cover all the conflicts we have detected, but in cases where we are unable to resolve conflicts to a satisfying degree we will identify the cause for this.

To summarize the requirements we have encountered we present a table of the four groups of conflicts we identified in table 1.1.

The four requirement conflict groups represent four groups we wish to identify conflict types within and resolve using a developed method for the purpose.

Requirements	
RCG-1	Class Conflicts
RCG-2	Attribute Conflicts
RCG-3	Class vs Attribute Conflicts
RCG-4	Data Representation Conflicts

Table 1.1: Requirement conflict groups

1.6 Structure of the Thesis

This thesis is divided in three parts.

The first part, "An Introduction to Multidatabases and Schema Integration", covers the properties of multidatabases and the problems of schema integration within multidatabases. It also discusses several approaches to solve the problem of schema integration. In chapter 2 we present the multidatabase systems, their objectives and key issues, and a basic architecture. In chapter 3 we go into further detail of one of the key issues of multidatabases, schema integration, braking down the schema integration process into subprocesses and describing each step. In chapter 4 we dig ourselves into the complexity of schema conflicts that can arise during the comparison step of schema integration by presenting a structural classification of the conflict types identified and also presenting a semantic measure for semantic similarity. In chapter 5 we look at some real-world prototypes and project systems that approach the multidatabase system design and see how their efforts differ and also how they manage integration.

The second part, "Schema Integration in ODL-M", gives an introduction to the ODL-M mapping language and then suggests a method of solving schema integration within the framework of using ODL-M as an extension to the ODMG/ODL object model [Cat94]. This part is the main contribution of this thesis where we develop and define a mapping language and use it to resolve the schema conflicts identified in chapter 4. In chapter 6 we define the ODL-M language, a mapping language for mapping object type interfaces in ODL to each other. Its definitions and use are demonstrated by various examples. In chapter 7 we develop a set of resolution techniques to resolve the conflict areas we identified in chapter 4. We also suggest how the semantic approach presented in chapter 4 can be used together with our resolution techniques to merge the schematic and semantic approach to schema integration. The case of this thesis is used as a basis for both exemplifying our conflicts and how to use ODL-M to resolve them by mapping constructs.

Following the main two parts is a third part, "Conclusion and Future Work", including the chapter of concluding remarks and suggestion to further work.

Finally the appendices are included.

Part I

An Introduction to Multidatabases and Schema Integration

Chapter 2

Multidatabases

In this chapter we will describe the properties of the multidatabase systems. Herein we will identify the objectives and key issues of such a system, and also present a general architecture. First a brief history of the evolvement of database systems will be given.

2.1 History of Database Systems

In the "early days", before any commercial database systems had been developed, data storage in computer systems was mainly done on files. The files were simple and often pure sequential. One can call this the very first primitive database system. The database managing system, if we could call it that, was made by the programmer himself who designed a data structure or format that these files should follow and therefore each database management system was uniquely different from another. This scheme soon showed to be unsufficient and better strategies needed to be developed. Independently several approaches to better data storage emerged.

2.1.1 Hierarchical Database systems

The *hierarchical model* [TL76], as the name suggests, is based on a hierarchical structure. The model consists of two main data structuring concepts: *records* and *parent-child relationships*. A record is a collection of *field values* that provide information on an entity or a relationship instance. A parent-child relationship type is a $1 : N$ relationship between two record types.

2.1.2 Network Database systems

The data of the *network model* [DBT71] are represented by collections of *records* and relationships among data are represented by *links*. Each record consists of a group of related data values. Navigation through the network is achieved by following the links.

2.1.3 Relational Database systems

In the *relational model*, the data are represented by a collection of tables. Each row in a table represents a collection of related data values. The relational database systems are based on the relational data model introduced by E. F. Codd [Cod70]. The model has become widely used because of its simplicity. Its advantages also include uniform data structures and a formal nature.

2.1.4 Object-Oriented Database Systems

The object-oriented database systems are built on the *object-oriented paradigm* (see appendix A). The data of interest is modeled as encapsulated *objects* which have a *state*, described by their attributes, and a *behavior*, described by their methods or operations.

2.2 Distributed Database Management Systems

A distributed database is a database that physically is spread over multiple sites in a network, but that logically belongs to the same system.

Reasons for distributed databases:

- Natural distribution in e.g. companies located at different sites(e.g. bank)
- Increased reliability and availability
- Controlled sharing of data throughout the distributed system
- Improved performance(locally)

2.2.1 Types of distributed database systems

The basic property of systems like this is that data and software are distributed over multiple sites connected by some form of communication network.

There are two factors in particular we will consider within these types of systems:

- Homogeneity versus heterogeneity.
- Autonomy versus non-autonomy

The first addresses the distributed systems characteristics such as operating system, database management system and modeling paradigm. The more heterogeneous the systems are, the more diverse their characteristics are. One system can use a relational model on a DOS platform while another can use an object-oriented modeling of its data on a Unix platform. Clearly, the more heterogeneous the distributed systems parts are, the more problems it introduces to manage the system.

There are two extremes of the autonomy spectrum relevant to this thesis. At the one extreme we have a distributed database management system (DDBMS) that looks

like a centralized DBMS to the user. There exists only one conceptual schema, and all access to the system goes through an application processor. No local autonomy exists. At the other extreme is a type of DBMS called a *federated* DDBMS or a *multidatabase* system, depending on its characteristics. In such a system each data processor(DP) is an independent and autonomous centralized DBMS that has its own local users with local transactions and a DBA[1] and therefore a very high degree of local autonomy. The local DP specifies an export schema to authorize access to particular portions of its database.

Note: A federated/multidatabase system is a hybrid between distributed and centralized systems; it is a centralized system for the local autonomous users and a distributed system for the global users.

Since the local DBMSs are heterogeneous it is necessary to have a canonical system language and include language translators in the AP(application process) to translate subqueries from the canonical language to the language of each data processor.

2.3 Multidatabases – A Motivation

Over the years companies and institutions have developed their computer systems according to their needs and what has been available on the market. Almost any institution that uses a data system of some kind has some sort of database system. The need for storing data has increased, likewise has the need for developing better database systems to manage the data been stored. There are numerous database systems to choose from and therefore there exists several systems in use. These systems however were not initially designed to communicate with each other. The need of accessing multiple systems at multiple sites homogeneously is increasing and it is this that forces the development of the multidatabase systems. These systems will be designed to integrate the existing systems so the the union of their shared data will be accessible as one dataset.

2.4 General Introduction – Basic Concepts and Definitions

There has been done a lot of research on interoperability of autonomous databases and architectures for such systems. However, we will mainly follow the classifications and taxonomy from Sheth and Larson [SL90] because their work gives a general overview that seems to cover the majority of the literature on the subject (e.g. [BHP92, HM85, LMR90, NSGS89]). Also, most of the literature in recent years seems to refer to Sheth and Larson [SL90] as a basis for their work.

As mentioned above, a database system (DBS) can vary in its level of distribution. At the one end a DBS can be centralized and residing on one computer system. At the other end a DBS can be distributed and residing on multiple computer systems. The multidatabases are characterized at the one extreme of the autonomy spectrum, as described above. A general description of a multidatabase system can be the following:

[1] Data Base Administrator

A Multidatabase System (MDBS) is a database system that manages multiple component DBSs residing at different sites, i.e. the MDBS can do operations across its participating DBSs simultaneously. A simultaneous operation across the component DBSs means that it's not only submitting separate requests to each DBMS, but that the operation is a cooperation of multiple requests to the component DBMSs.

A MDBS can be *homogeneous* or *heterogeneous*. A homogeneous MDBS means that each component DBMS is the same, i.e. is based on the same data model and system level support(concurrency control, commit, recovery etc). In a heterogeneous MDBS the component DBMSs are different, i.e. are based on different data models or the underlying DBMSs are different. We normally think of the latter type when discussing MDBSs in general

2.4.1 Objectives and Key Issues of Multidatabase Systems

Here are the most important general objectives of a multidatabase system [Kim95]:

Objective 1: It must obviate the need for a batch conversion and migration of data from one data source to another.

Objective 2: It must require absolutely no changes on the local database system(LDBS) software. This preserves the design autonomy. In other words, an MDBS must appear to any of the LDBSs as just another application or user.

Objective 3: It must not prevent any of the LDBSs from being used in its native mode. In other words, users of an LDBS may continue to work with the system for transactions that require access only to data managed by the system, while users will use the MDBS to issue transactions that require access to more than one data source. In this way, applications written in any of the LDBSs are preserved, and new applications that require access to more than one data source may be developed using the MDBS.

Objective 4: It must make it possible for users and applications to interact with it in one database language. In other words, the users and applications should not have to work with the different interface languages of the LDBSs.

Objective 5: It must shield the users and applications from the heterogeneity of the operating environments of the LDBSs, including the computer, operating system and network protocol.

Objective 6: Unlike most previous attempts at allowing the interoperability of heterogeneous database systems, it must support distributed transactions involving both reads and updates against different databases.

Objective 7: It must be a full-blown database system – that is, it must make available to users all the facilities provided by standard database systems, including schema definition, non-procedural queries, automatic query optimization, updates, transaction

management, concurrency control and recovery, integrity control, access authorization, both interactive and host-language application support, graphics application development tools, and so forth.

Objective 8: It must introduce virtually no changes to the operation and administration of any of the LDBSs.

Objective 9: It must provide run-time performance that approaches that of a homogeneous distributed database system.

We summarize the objectives in table 2.1.

Multidatabase System Objectives	
1.	Obviate need for batch conversion and migration
2.	No change to local database system
3.	Allow LDBSs to be used in its native mode
4.	Users and applications interact with *one* language
5.	Shield users from heterogeneity of operating environment
6.	Support for distributed transactions(reads and updates)
7.	Full-blown database system
8.	No changes to operation and administration of LDBSs
9.	Run-time performance that approaches "ordinary" distributed system

Table 2.1: Multidatabase System Objectives

The three key issues for a full-fledged multidatabase system are [Kim95]:

1. **Constructing a global schema across independently designed heterogeneous databases.** The basis for achieving this is having a comprehensive taxonomy of schema differences *and* a schema integration technique for homogenizing, i.e. resolving, each type of difference.

2. **Processing of queries that the users will issue against the global database.** This is achieved by translating each global query into a set of subqueries to be carried out by the LDBSs.

3. **Management of transactions, issued against the global database as an atomic unit, across heterogeneous databases.** Sub-issues in this context are how to obtain concurrency control and recovery routines as the scheme here obviously is more complicated than the legacy database systems. Deadlock detection and resolution is also a consideration in this regard.

We will come back to schema integration in section 2.6, query processing in section 2.7, and transaction management in section 2.8.

2.4.2 Three Dimensions of Multidatabases

Multidatabase systems can be viewed from three orthogonal dimensions [SL90]:

Distribution Databases may be stored on one machine or distributed among multiple data-machines in different ways. In the MDBS case there usually exist component DBSs that are distributed physically from the beginning across some communication lines.

Heterogeneity There are two sides of heterogeneity; technical and semantic. The technical side involves differences in DBMSs such as data models, differences in operating systems and hardware systems. The semantic heterogeneity deals with the problems of the understanding or intention of the data in the databases. This is an area not yet fully understood and introduces difficult problems in building multidatabases.

Autonomy The dimension of autonomy measures the strength of a component DBSs independent control. This can be in terms of how the component DBS is designed, how willingly it will communicate with others and also to what extent it will share its data.

In the following the two latter dimensions will be focused on. The next section will present a taxonomy which focuses on the autonomy dimension.

2.4.3 Taxonomy of Multidatabase Systems

In this section we present a taxonomy of multidatabase systems as described in [SL90]. It will show the classifications such a system can have and what characterizes each classification. It will also serve as a pointer to which classification we assume to be addressing in the later chapters of this thesis.

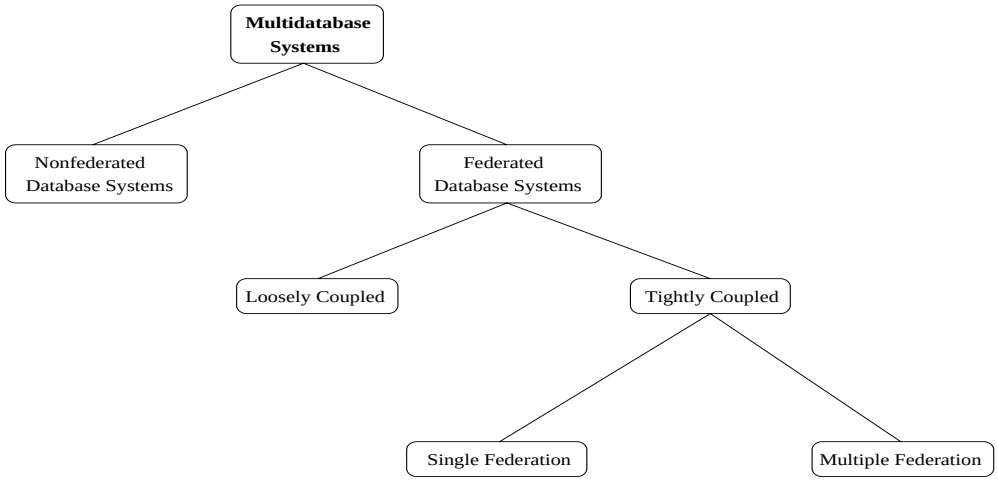

Figure 2.1: Taxonomy

A *multidatabase system* is a database system designed to operate on multiple component databases. Based on the level of autonomy of the component systems, an MDBS can be classified into two types(fig.2.1):

- Non-federated database system – A non-federated system is an integration of local systems which have no local autonomy. The overlaying MDBMS controls all the participating DBMSs which are only slaves to the system.

- Federated database system – A federated system has a set of autonomous DBSs participating in the federation. There is an agreement of how the local systems share their data with the federation so they still can control their data as an independent DBS. Sharing data means that the component DBS has to somehow respond to extern requests and therefore it must give up some of its autonomy to support the cooperation.

The autonomy factor of the component systems affects the way they are integrated into the system. Another factor to consider is the responsibility of management of the system. We can categorize a federated system by these two factors; A federated database system(FDBS) is *loosely coupled* if the user is responsible for the management of the federation. On the other hand we have a *tightly coupled* FDBS if the administrators of the federation have the responsibility of managing the system. This means creating a federated schema and controlling access to the component DBSs. To the user, the federation will be transparent in the sense that he will not see at which underlying system the data of the federated schema originates.

Finally, in this taxonomy, we can categorize tightly coupled federated systems as *single* federations and *multiple* federations. The difference lies in how many federated schemas are allowed created in the system. A multiple federation allows several schemas, a single federation allows only one. This one schema will then be a union of the shared data of the component DBSs. In the rest of this thesis we will be referring to a single federation when talking about a multidatabase in general.

2.4.4 The Five-Level Schema Architecture

In the traditional ANSI/SPARC architecture [TK78] for database systems, the *three-schema architecture*, we have three (of course) levels; the *internal* level, the *conceptual* level and the *external* level. This architecture was proposed for single database systems to achieve data independence, both logical and physical. The three-schema architecture is adequate for describing the architecture of centralized DBMSs, however it is not adequate for MDBMSs because of the three dimensions; distribution, heterogeneity and autonomy. To support the three dimensions, Sheth and Larson [SL90] have proposed an extended architecture: The five-level schema architecture (fig. 2.2).

The five-level architecture has the following schemas:

Local Schema The local schema is the conceptual schema of one of the component DBSs. It is therefore expressed in this local DBMS's native data model.

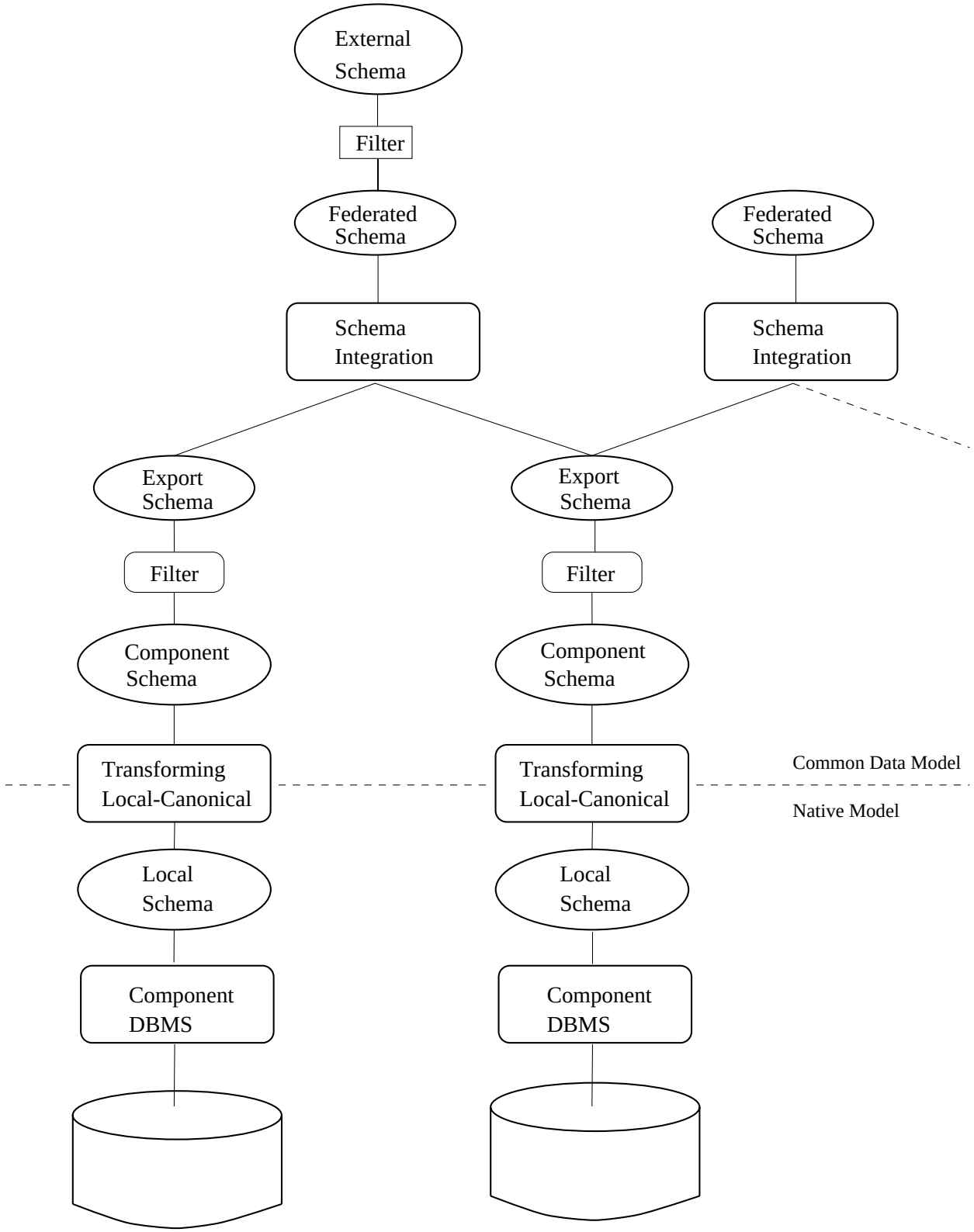

Figure 2.2: Five-level schema/system architecture of an FDBS

Component Schema To avoid the MDBMS having to understand several data models, the local schemas are translated to a uniform model, called the *canonical model*(see. 2.5). It is at this level the heterogeneity feature of MDBMSs is supported. This process generates the mappings between the component schema objects and the local schema objects. The process of translation might also add semantics to the original schema to provide better understandability in the component schema.

Export Schema The autonomy feature of a MDBMS allows the component DBSs, among other things, to restrict access to its data. The export schema is only a subset of the component schema, where the non-shared data is filtered out. This is the schema that is available to the federation.

Federated Schema The federated schema is an integration of multiple export schemas from the component DBMSs. The federated schemas also need information on how these integrated schemas were constructed, i.e. which parts of it came from which export schemas.

External Schema In a MDBMS, as in a traditional centralized DBMS, the user does not always need access to all the data of the available schema. The external schema is defined for a user or application and a filtering process is applied to the federated schema to filter out unnecessary or non-accessible data.

Saltor et. al [SCG94] have suggested an extension of this architecture. They argue that three additional schemas be added:

Negotiable Schema: This schema is located between the component schema and the export schema. It serves as the data from the CDB that different federations can negotiate from to include in their export schemas. Different federations may therefore have different export schemas derived from the same negotiable schema.

Translated Schema: The user of the federated system might not be educated in the canonical model used. Therefore in those cases where the user requirements demand it, the external schema is divided in two: the *translated* schema which filters out data from the federated schema, and the *user schema* which is modeled in the user familiar model. Between the translated and the user schema is a transforming processor to translate to the users model.

Application Schema: The federated schema is integrated from different source component schemas. It expresses the underlying semantics by compromising the different semantics expressed. However the user might want to differentiate between separate semantics in the federated schema, thus an *application schema* should express which semantics to use. It is located between the federated schema and the external/translated schema to support multiple semantics. The user might have a requirement of knowing which local database the data came from, so the data is source tagged for this purpose.

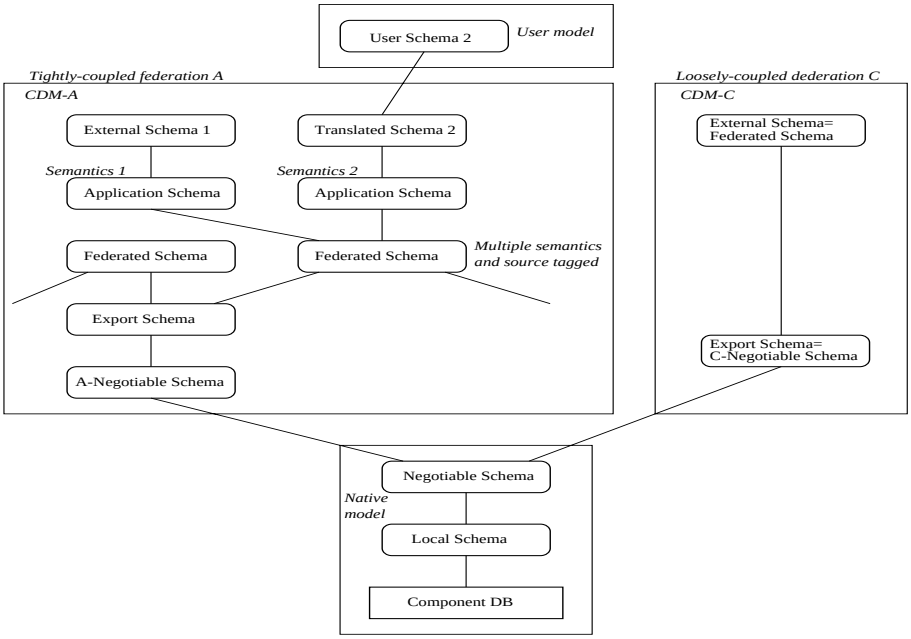

Figure 2.3: Complete 8-level Architecture

The full 8-level architecture is described in figure 2.3.

The substance of these additional schemas are not applicable to the focus of this thesis so we will not mention them in the following. However we feel it is interesting to include them as they may show to be necessary in future research on full-fledged multidatabase systems.

2.5 Canonical Data Model

Suppose we were to develop a multidatabase system with n heterogeneous component systems. Initially we are dealing with potentially n different data models that we are trying to communicate across. At first sight we might need $n \times n$ translators to cope with the $n : n$ possible connections. The factor $n \times n$ grows by a square factor with the number of component systems connected and this factor is naturally undesirable. Parallel to e.g. networking systems we therefore define one data model as a standard that all models can be translated/mapped to. This data model we call the *canonical data model*(CDM). Having one common data model at the component level ensures that one only needs n translators in the MDBMS. The CDM can be the same as one of the component data models, but can also be a model separate from the existing ones. We can say that this translation solves the problem of syntactic heterogeneity, consequence of the use of different native data models. The CDM must be chosen according to requirements for suitability in such a system. This is discussed next.

2.5.1 Requirements for a Canonical Data Model

For a data model to fit as a CDM, Saltor et. al [SCG91] suggest that it should have two properties: *Expressiveness*, and *Semantic Relativism*. Especially for schema integration they argue that data models should support *views*. Views are also discussed as an enhancement to to models by Pitoura et. al [PBE95] and we therefore add it as an additional requirement. We summarize the three requirements in table 2.2 and describe them in the following.

Requirements for a canonical data model	
RCDM-1	Expressiveness
RCDM-2	Semantic Relativism
RCDM-3	Support for Views

Table 2.2: Requirements for a canonical data model

2.5.1.1 Expressiveness

The canonical model is the model that all underlying models translate to. This means that this model should be powerful enough to express all concepts of all potential component data models. Otherwise we would lose information in the translating process. Moreover it should support additional semantics made explicit through a semantic enrichment process in case that such a process is applied.

Expressiveness may be seen as composed of a structural part and a behavioral part. *Structural expressiveness* is the ability of the structures of the model to represent concepts. *Behavioral expressiveness* reflects the ability of the model to represent behaviors of concepts.

2.5.1.1.1 Semantic enrichment Assuming that the component local schemas are less expressive than the CDM leads to the fact that the local schemas were intended to express more information than they explicitly show by their syntax. In fact, one can say that all schemas are intended to express more than their syntax expresses. This is especially true of the semantics of the schema where the semantics are not explicitly shown. Since the CDM has greater expressiveness, it would be beneficial to extract knowledge from the local schemas that is intended, but not explicitly expressed. This will be referred to as *semantic enrichment* and it is done through the process of knowledge acquisition in cooperation with the schema designers. Thereby the transformation from local schemas into component schemas is not just a syntactic translation from one model to another, it includes a structural and behavioral semantic enrichment in order to upgrade the semantic level of the local schemas. The schema integration process will then be less difficult, by making use of these additional semantics to detect and solve semantic conflicts.

2.5.1.2 Semantic Relativism

A DB should not just support one conceptualization, but many. A conceptualization is not absolute, it is relative to the point of view of a user or group of people because different persons perceive and conceive reality in different ways. A DB should store a single conceptualization, encompassing all those conceptualizations, and avoiding redundancies. For the DB to support all users' conceptualizations, it must support one *external schema* for each conceptualization, and their *derivation* from the single database schema.

The *semantic relativism* of a DB is the degree to which it can accommodate all these different conceptualizations(of the same real world).

We call *semantic relativism of a data model* the ability of its operations to derive external schemas.

2.5.1.3 Support for Views

A *view* is a way of defining a "virtual" database on top of one or more existing databases. Views are not stored, but are recomputed for each query that refers to them. The definition of a view is dependent upon the data model and the facilities of the language used to specify the view. Object-oriented views are in general defined by a set of virtual classes that are populated by existing objects or by imaginary objects constructed from existing objects ([PBE95]). In general, the object-oriented models lacks some necessary mechanisms for grouping already-existing objects and we need therefore to define a suitable way to define and express views.

2.6 Schema Integration

As we mentioned in section 2.4.1 the three key issues were; constructing a global schema by means of *schema integration*, processing of queries by means of *query processing*, and management of transactions by means of *transaction management*. In the following we describe these three issues and how they are characterized in multidatabase systems. First, we present schema integration.

The process of schema integration takes place between the export schemas and the federated schemas in the five-schema architecture. Its purpose is to integrate the constructs of the export schemas. This integration is the basis for the communication between the global and local system and provides the MDBS with a global schema representing all of the underlying systems shared data to the global management system. When integrating schemas from local export schemas there are two considerations to make:

Structural: The structural constructs of the local schemas must be investigated for similarities and discrepancies. These include aliases between named entities and related inheritance hierarchies. The structural conflicts must be resolved by proper schema integration techniques to form the integrated schema.

Semantical: Pure data has no meaning without an interpretation of it. Since the schemas of the local databases usually are designed by different designers, their modeling may

vary by which viewpoint they have. In the schema integration process it is important to capture the intention of the data represented. The semantical meaning can vary in level from a meaning of a relationship down to the symbolism of attribute values.

We will cover the process of schema integration and the conflicts that can arise therein in chapter 3 and 4.

2.7 Query Processing

A multidatabase system must support a query processing system in order to extract the information within the underlying databases. The application programmer is provided with a global query language to specify queries against the global schema. We call these queries *global queries* to distinguish them from the queries taking place at the local DBs. Conceptually a global query can be processed in three steps[MY95]:

- Query decomposition
- Query translation
- Query combining

2.7.1 Query Decomposition

When a global query is submitted, it is first decomposed into two types of queries by the *query composer*. One is queries against individual export schemas. These type of queries are called *export schema subqueries* or simply *subqueries*. Another is queries that combine the results returned by subqueries to form a global answer. These types of queries are called *post-processing queries*. Post-processing queries may not always be needed.

The decomposition itself is usually accomplished in two steps. At the first step, the global query using global names is modified to queries using only names in the export schemas. At the second step, these queries are decomposed such that the data needed by each subquery are available from one local database.

2.7.2 Query Translation

After the decomposition the subqueries are still in the global query language. If the global query language is different from the language of the local database system, the corresponding export schema subquery must be translated to the local subquery language by the *query translator*. For example, if the global query language is an OODB query language and a subquery is for a relational system using SQL, then this subquery needs to be translated to an SQL query. The system will need as many query translators as it has heterogeneous local database systems attached.

2.7.3 Query Combining

Finally the local query processors will return their answers and it is up to the global query processor to now combine all the results to a global answer which addresses the query that was submitted in the first place.

2.7.4 Optimizing Global Queries

Clearly, the complexity of this scheme is at a high level. Query optimization in single database systems is a research field of its own. Now the field has been broadened to optimize over distributed data in different data models. When optimizing the process of global query decomposition and translation the system must consider the whole chain of events from the global query modification, through the translation to the actual data transfer that takes place when the local data is returned. This area will show to be of great importance for providing good enough performance in future multidatabase systems.

2.8 Transaction Management

Access to data located in one or more local data sources is accomplished through *transactions*. A transaction results from the execution of a user written program written in a high level programming language. In a single database system the transaction manager system can work on the basis of an autonomous, cooperating system. In a multidatabase system a major problem is introduced; *local autonomy* in the local database systems.

There are two types of transactions executed at local systems[BGMS95]: transactions that the MDBS is not aware of, i.e. local transactions, and transactions that the MDBS has submitted to the local DBMS as a part of the execution of a global transaction(global sub-transactions). Each local DBMS has its own concurrency control mechanism that ensures serializable and deadlock-free execution of local transactions and global sub-transactions. The objective of MDBS transaction management is to ensure multidatabase consistency in the presence of local transactions.

The key issue to how the transaction manager will be able to perform is *autonomy* – that is, how willingly are the individual DBMSs to share their control information with the MDBS or to restrict access of local transactions to local data. Defining local autonomy too broadly may lead to considerable difficulties in retaining global database consistency. On the other hand, defining local autonomy too narrowly would not satisfy the basic requirement that a local DBMS be largely independent from the centralized coordinator and thereby would make the multidatabase system unacceptable to the users.

As the case is for the stand-alone database systems, the MDBMS's global transactions must be *atomic* for correctness – that is, either all their actions commit or they all abort. In a homogeneous distributed database system, atomicity of transactions is ensured by an *atomic commit protocol* such as the two-phase commit(2PC) protocol. The 2PC protocol requires that the participating local sites provide a *prepare-to-commit* command in their interface and thereby promising the global manager that it will commit its work in the

future if so asked by. It loses some of its autonomy by this commitment since it is no longer free to make decisions regarding some of its own resources.

Multidatabase transaction management research is still at an early stage, and considerable work needs to be done. It seems clear that full data consistency and serializability can only be achieved in a multidatabase system by imposing restrictions that many consider severe. Thus, there may be a need to identify alternative forms of consistency and ways of restricting standard notions of consistency so that positive results can be stated, rather than impossible results.

2.9 Basic Problems in Multidatabase Systems

The main keywords for what complicates the idea of multidatabases are *heterogeneity* and *autonomy*. The heterogeneity exists at three basic levels. The first is the platform level. Database systems reside on different hardware, use different operating systems and communicate with other systems using different communication protocols. The second level of heterogeneity is the database management system level. Data is managed by a variety of database management systems based on different data models and languages(e.g. file systems, relational database systems, object-oriented database systems etc.). Finally the third level of heterogeneity is that of semantics. Since different databases have been designed independently, semantic conflicts are likely to be present. This includes schema conflicts and data conflicts. Semantic conflicts might be considered the hardest level of heterogeneity to resolve due to the complexity it can have and the fact that it seems to not be fully understood.

The autonomy of the underlying systems is a crucial factor to how smooth the federation will cooperate. The spectrum can vary from a CDB being a slave to the MDBMS to operating as a stand-alone DB where the MDBMS is treated like any other application or user with no special privileges. From the federations viewpoint it is desirable that the underlying systems "obey on command", but there can be several reasons for a CDB to maintain its autonomy(e.g. efficiency demands it must meet locally or security aspects). So the global and local systems may have to negotiate on a policy to follow regarding the autonomy of the local system in order to create a system that all participants can accept.

With reference to MDBS objective no. 6: "Support for distributed transactions(reads and updates)", it has become clear that this objective might be difficult, if not impossible, to completely satisfy. Transactions regarding read-only operations seem to be achievable, but when it comes to update-transactions the complexity of performing this will increase vastly compared to single database systems. In short, the reason for this is that we may define mappings from the global schema to the local ones, but their nature is often that they are irreversible such that the data flow cannot be defined the opposite way – at least not without undefined or undesirable results. Also the control of this operation introduces new problem dimensions that might demand that we give up other objectives(e.g. autonomy).

2.10 Summary

In this chapter we introduced the multidatabase systems. We gave an overview where we identified the objectives such a system should strive for, the three key issues sought solutions for, and suggestions to alternative architectures. The canonical model was argued to be an important build-stone and we listed table 2.2 as the requirements it should meet. We will come back to how our thesis proposal meets these requirements in chapter 7. From the three key issues we chose to focus on schema integration and will take a closer look at it in the next chapter. This chapter has served as a basis for our further work as we now go into further detail of schema integration.

Chapter 3

Schema Integration

In this chapter we will describe the process of schema integration from section 2.6 in more detail. We will explain why schema integration is needed and why schema diversity arises. The problems in this area will also be discussed. Finally we will define a set of requirements that the schema integration process and its resulting integrated schemas should meet.

3.1 Definition

Batini et. al [BLN86] define *schema integration* as the following:

Schema Integration: the activity of integrating the schemas of existing or proposed databases into a global unified schema.

We can divide the occurrence of schema integration in two contexts [BLN86]:

View Integration (in database design) produces a global conceptual description of a proposed database. This would be in the context of "top-down" methodology from [SL90].

Database Integration (in distributed database management) produces the global schema of a collection of databases. This global schema is a virtual view of all databases taken together in a distributed database environment. This would be in the context of "bottom-up" methodology from [SL90].

The focus in this thesis will be on *database integration*. The database integration activity is described in a general way in figure 3.1. It shows that this activity has as input the local schemas and the local queries and transactions. There has not been done much work that explicitly takes into account the latter process-oriented information in developing the integrated schema. It is strictly used in mapping the queries between the global and the local levels. Hence the figure shows the global schema as well as the data and query-mapping specifications to be the outputs of the database integration activity. However, this thesis will focus on the structural and semantical merging of

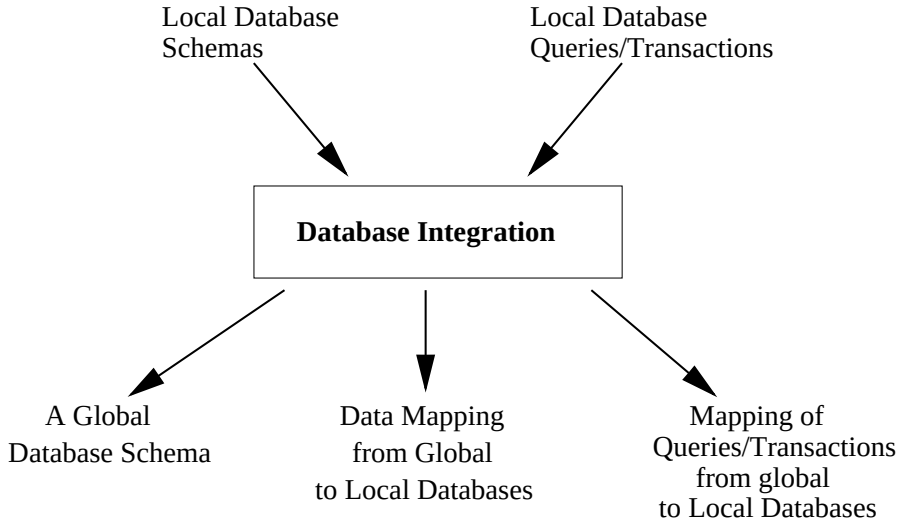

Figure 3.1: Database Integration

local schemas into a global schema. Although query and transaction mapping is partly inherently supported by the schema merging, it will not explicitly be discussed further[1].

More specifically, schema integration in a multidatabase system takes place between the level of export schemas and the federated schema(s) of the system (see section 2.4.4 on page 19). Selected export schemas are integrated into federated schemas. If it is a single federation, all the export schemas are integrated into one global schema. If a multiple federation is being designed, several federated schemas are integrated from the export schemas where the export schemas can participate in one or more global schemas. Both source and target schemas in this process are of the same common data model so there is no translation involved. However there must be mappings between them.

The schema integration process can be thought of as deriving a single schema (in the case of single federation) from a set of schemas through a sequence of simpler functions each of which address (resolve) a schematic discrepancy [KCGS95]:

$$schema\_int\_process : schema_1 \times schema_2 \times \ldots \times schema_n \rightarrow int\_schema$$

This is just to get a preliminary overview of this process. We will look closer at it in section 3.4.

3.2 Integration Motivation

There is a growing trend to regard data as an autonomous resource of the organization, independent of the functions currently in use in the organization. There is a need to capture the meaning of data for the whole organization in order to manage it effectively.

[1] An introduction to query processing and transaction management in MDBSs was presented in chapter 2

Because of this awareness, integration of data has become an area of growing interest in recent years.

From a view integration point of view, the structure of the database for large applications (organizations) is too complex to be modeled by a single designer in a single view. Several designers can break down the complexity and perform a view integration to merge the parts. From a database integration point of view, different user groups typically operate independently in organizations and have their own requirements and expectations of data, which may conflict with other user groups. A multidatabase system builds a common view over the user groups' local database systems(LDBSs) that are of interest, and the schema integration process is an effort to solve and homogenize their conflicts.

3.3 Causes for Schema Diversity

The basic problems to be dealt with during integration come from structural and semantic diversities of schemas to be merged. The main reasons for this diversity is discussed in the following.

3.3.1 Different Perspectives

Different user groups adopt their own viewpoints in modeling the same objects in the application domain. A real life object modeled as some construct in one application, might be found to be modeled as a totally different construct in another application.

3.3.2 Equivalence among Constructs of the Model

In conceptual models, several combinations of constructs can model the same application domain equivalently. As a consequence, semantically "richer" models have a larger variety of possibilities to model the same situation. As an example, figure 3.2 shows two equivalent constructs, where `Man` and `Woman` are distinguished by a generalization hierarchy in the first schema, whereas in the second schema they are distinguished by the different values of the attribute `Sex`.

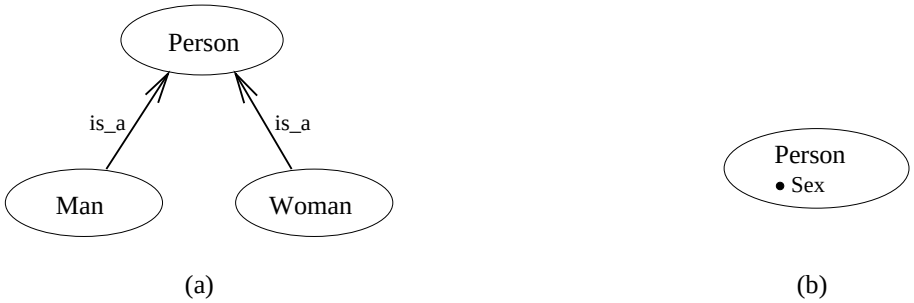

Figure 3.2: Equivalent constructs: (a) Generalization hierarchy. (b) A single class

3.3.3 Incompatible Design Specifications

A good schema integration methodology should, as far as possible, analyze the component schemas for erroneous choices regarding names, types, integrity constraints, etc. Such errors, if not detected and corrected, may result in erroneous inputs to the schema integration process, propagating the errors upwards to the global schema.

3.3.4 Common Concepts

The three aspects above are concerned with what we call the common part of the various schemas, i.e. the set of concepts of the application domain that are represented in two or more schemas. In other words, the above aspects represent the reasons why the common part may be modeled in different ways in different schemas. In order to perform schema integration is is important to single out not only the set of common concepts, but also the set of different concepts in schemas that are mutually related by some semantic properties. In general we refer to these as *interschema properties* and the conflicts among such properties as *interschema conflicts*.

3.4 The Process of Integrating Schemas

In their survey, Batini et al.[BLN86] suggest the process of schema integration be divided into the following activities:

- Preintegration
- Comparison
- Conforming
- Merging and restructuring

These activities will be described in the following sections.

3.4.1 Preintegration

This action is carried out as a method of deciding on policy for the rest of the process. Typically this stage consists of choosing which schemas should be integrated with each other and in which order. Global strategies for integration, namely the amount of designer interaction and the number of schemas to be integrated at one time, are also decided at this phase. Finally one tries to get an overview of any additional information needed for the integration that isn't implicitly known such as any assertions or constraints.

The choice of schemas also involves processing component schemas in some sequence. In general, the number of schemas considered for integration can be $n \geq 2$. Figure 3.3 shows four possible variations termed *integration processing strategies* [BLN86] . Each strategy is shown in the form of a tree. The leaf nodes of the tree correspond to the component schemas, and the non-leaf nodes correspond to intermediate results of integration.

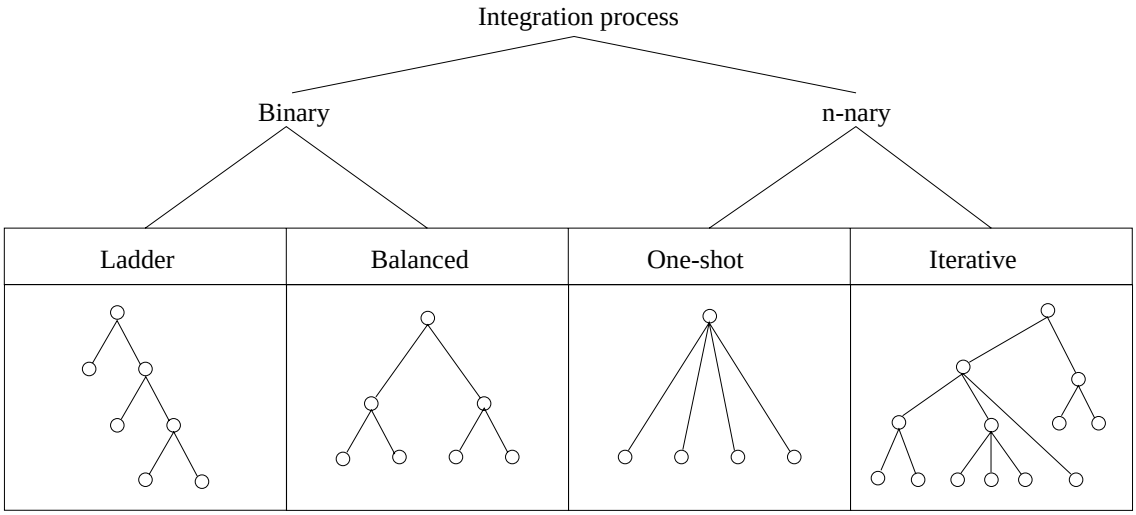

Figure 3.3: Types of integration-processing strategies

The root node is the final result. The general integration process strategies are classified into binary versus n-ary.

Binary strategies allow the integration of two schemas at a time. They are called *ladder strategies* when a new component schema is integrated with an existing intermediate result at each step. A binary strategy is *balanced* when the schemas are divided into pairs at the start and are integrated in a symmetric fashion.

N-ary strategies allow integration of n schemas at a time ($n > 2$). An n-ary strategy is *one-shot* when the n schemas are integrated in a single step, it is *iterative* if the schemas are integrated stepwise through intermediate schemas. This is the most general case.

The advantage of binary strategies is the simplification of the activities of comparison and conforming at each integration step. The disadvantages are an increased number of integration operations and the need for a final analysis to add missing global properties.

The advantages of n-ary strategies are: A considerable amount of semantic analysis can be performed before merging, avoiding the necessity of a further analysis and transformation of the integrated schema. Also the number of steps for integration is minimized. The disadvantages are that the analysis will be more complex.

3.4.2 Comparison

This step is also called the schema analysis step. It involves comparing the concepts of schemas to be integrated to determine conflicts in the representation of the corresponding objects in different schemas. Two main conflict types are *naming* conflicts and *constraint* differences.

The naming conflicts arise due to that people from different application areas of the same organization refer to the same data using different terminology or different data using the same terminology. This gives rise to two conflict types:

1. **Synonyms:** When the same name is used for two different concepts, causing inconsistency unless detected.

2. **Homonyms:** When the same concept is described by two or more different names, preventing a complete merging unless detected.

A special type of homonyms occurs when for the same concept there is a match on names but no match on the corresponding sets of instances. They can occur at various levels of abstraction. An example could be the class `Student` in one database referring to all students registered whereas in the married-student-housing database it refers to married students only.

We use the term *structured conflicts* to include conflicts that arise as a result of a different choice of modeling constructs or integrity constraints. To roughly capture the basic conflict types we present a classification which distinguishes between the following kinds of conflicts:

- **Type Conflicts**. These arise when the same concept is represented by different modeling constructs in different schemas. This is the case when, for example, a class of objects is represented as a defined class in one schema and as an attribute in another schema.

- **Dependency Conflicts**. These arise when a group of concepts are related among themselves with different dependencies in different schemas. For example, the relationship `Marriage` between `Man` and `Woman` is $1:1$ in one schema, but $m:n$ in another accounting for a marriage history.

- **Key Conflicts**. Different keys are assigned to the same concept in different schemas. For example, `SocSec#` and `Emp_id` may be the keys of Employee in two component schemas.

- **Behavioral Conflicts**. These arise when different insertion/deletion policies are associated with the same class of objects in distinct schemas. For example, in one schema a department may be allowed to exist without employees, whereas in another, deleting the last employee associated with a department leads to the deletion of the department itself. Note that these conflicts may arise only when the data model allows for the representation of behavioral properties of objects.

The comparison step also includes the activity of discovering interrelationships among the schema objects. It may be a by-product of conflict detection. However found, any inter-schema properties discovered during this step are saved and processed during schema merging. Although it would be of great advantage to automate the comparison step of integration, its complexity and often semantically influence leads to that it in general is aided by a strong interaction between the designers and users .

We will take a closer look at the conflicts that can arise during this step in chapter 4.

3.4.3 Conforming

After the conflicts have been determined, an effort is made to resolve them so that the merging of the schemas is possible. The goal of this activity is to conform or align schemas to make them compatible for integration. Achieving this goal amounts to resolving the conflicts, which in turn requires that schema transformations be performed. One might have to accept compromises on certain aspects to achieve workable results. This is because some types of conflicts are of such a nature that they can not be resolved completely as desired as we will see later in the thesis. As an example of resolving naming conflicts, we can use simple renaming operations for homonyms, such as prefixing the names with the schema name of which they belong to.

3.4.4 Merging and Restructuring

Now all the analyzing and preparations have been done and the actual merging of the schemas takes place. This step can take place at each temporary integrated schema or just at the final schema. Analyzing the final merged schema can give some last information on how to do a final restructuring in order to achieve more desirable properties.

3.4.5 Summary - Integration Process

We have given an overview of the steps of the process of schema integration according to Batini et. al [BLN86]. These steps are general, basic steps that each include their own subproblems to be solved. It is important to note that this list not necessarily is followed sequentially, but can also be followed stepwise with drop-backs to previous steps and iterations over two or more steps to achieve the desired final result. To clarify the intermediate steps' results we have constructed a graphical representation of the process is given in figure 3.4 that spans the process from the local schemas to the integrated federated schema.

In our proposal later in this thesis (chapter 7), we will come back to a suggestion to how we will conform and merge the conflicts in our case.

3.5 Requirements for Schema Integration

In this section we discuss what might be considered as successful schema integration and give some requirements to how obtain this.

3.5.1 What is Good Schema Integration?

To answer this question, it is important to first look at the goal of schema integration. In our context we are trying to integrate two or more schemas from heterogeneous databases into a global uniform schema to access the information in a homogeneous manner. The goal in this must be to be able to access the underlying data in a transparent way through the integrated schema and that the integration process is done in a way that is understandable

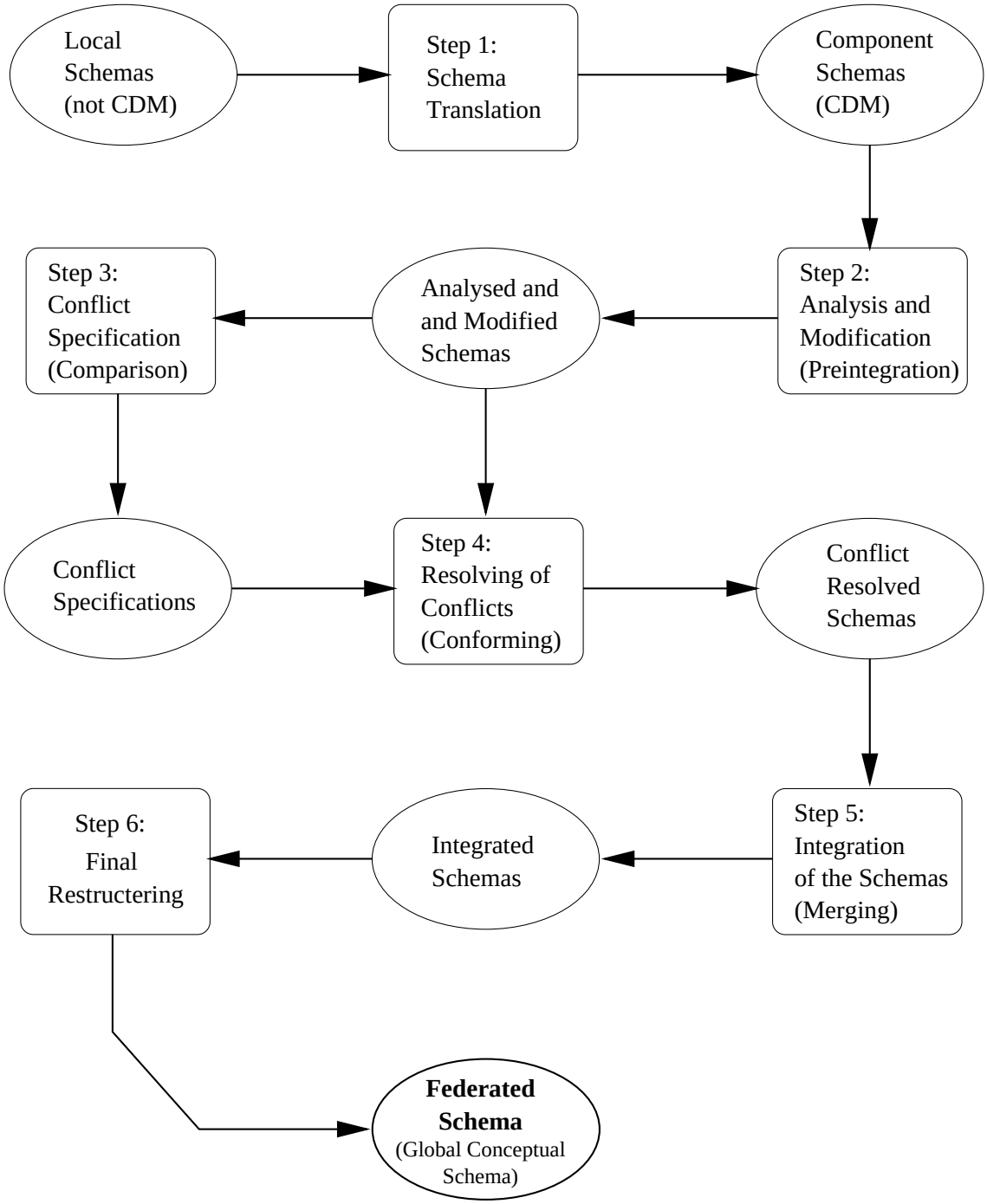

Figure 3.4: Steps of the schema integration process.

and not too complex. The final integrated schema should also meet normal expectations of that of native database schemas.

3.5.2 Requirements

In the following sections we will list a number of requirements for future reference. The requirements will as far as possible be a guide to evaluate solutions as to how well they support schema integration of schemas.

In their comparison survey, Batini et. al [BLN86] give a list of requirements for what qualitative criteria the global schema integration should have: *completeness*, *correctness*, *minimality* and *understandability*. Here we would like to add a new requirement to this list, a schema integration process oriented criteria, namely *schema integration support*. Batini et. al [BLN86] do not discuss this requirement so we introduce it in this thesis as an additional requirement. The requirements are described closer in the following sections.

3.5.3 Completeness

The integrated schema should represent the union of all the concepts in the component schemas. To achieve completeness, the designer has to conclude the analysis and addition of inter-schema properties that is initiated in previous design steps. Note that the variety of inter-schema properties is strongly related to the repertory of schema constructs at the disposal of the data model.

3.5.4 Correctness

Having covered all the concepts of the underlying schemas it is also desired that the information represented in the integrated schema is correct. One might occasionally have a situation where one can slightly compromise this to achieve other goals, but this is generally not advisable as the consequences can be undetermined.

3.5.5 Minimality

Integrating schemas with different concepts will sometimes integrate the same concept from different schemas. The minimality requirement says that multiple equal concepts integrated must only be represented once in the integrated schema. This is of course to avoid redundancy as a basic requirement in all database systems. Also concepts that can be derived by other concepts should be considered when seeking equivalent concepts.

3.5.6 Understandability

The process of integrating schemas and the final global schema should have a reasonable understandability. Solving conflicts by a highly complex method that produces a schema which is hard to understand will cause other problems. This requirement implies that among the several possible representations of results of integration allowed by a data model, the one that is qualitatively the most understandable should be chosen. However

a quantitative and objective measure of conceptual understandability is difficult to define. Some guidelines to parameters in a graphical representation of the conceptual model could be shape of the diagram, the total length of connections, the number of crossings and bends, and so forth.

3.5.7 Schema Integration Support

As far as possible the process of integration should be automated. It is desirable that the system supports the integration process and that the interaction with the developer is minimized to the degree where the developer still can follow and understand the process. Due to the highly semantic degree of understanding schemas, it is obvious that this process cannot be entirely automated, but integration guide tools and mapping methodologies are examples of helpful aids.

3.6 Summary

In this chapter we discussed in further detail our key issue in multidatabase systems; schema integration. We argued why schema diversity arose and identified the need for schema integration. The process was broken down into subprocesses that all should be part of an effective and understandable means of achieving the integrated schemas, from to merging and restructuring of the final schemas. The schema integration process and its resulting schemas should meet some qualitative criteria, so we summarize the criteria as table 3.1 as the requirements they should meet. We will revisit the requirements in

Requirements for schema integration	
RSI-1	Completeness
RSI-2	Correctness
RSI-3	Minimality
RSI-4	Understandability
RSI-5	Schema Integration Support

Table 3.1: Requirements for schema integration

chapter 8 where we will argue how our proposal has met them.

In the 'comparison' step of the schema integration process it is clear that there are several types of conflicts to identify and resolve between schemas. In the next chapter the inter-schema conflicts will be in focus as we will classify the requirement conflict groups from table 1.1.

Chapter 4

Schema Heterogeneities

In this chapter we will outline various schema heterogeneities that have been encountered. It can be considered a study of the *comparison* step in the schema integration process of chapter 3. It is not a canonical overview, but discusses the most important variations that have been sought solutions for. First we will define a general classification that is constructed to give a starting point of discussion. Later we will go into further detail on this classification by adopting classifications from the literature. Two views on the schema heterogeneities are given; a structural classification that conforms to our requirements in table 1.1 in the introduction chapter, and a semantic view expressed by a semantic proximity function. The structural classification breaks up our mentioned requirement conflict groups in table 1.1 into further detail describing the characteristics of each conflict and also giving examples. The semantic approach defines a semantic measure of semantic similarity introduced by Sheth and Kashyap [SK92]

4.1 Introduction

Schemas in independent databases are often designed by different schema designers. This difference in design introduces several problems as to interpreting the semantics of the schemas. One of the key problems that will arise in this context is to be able to verify equivalences and relations between database schemas or parts of them.

Examples of situations where this problem turns up could be [Øre92]:

- Modification of a database schema, in order to obtain a better one.
- Design of the various schemas for a database, using a DBMS with a multi-schema architecture.
- Design of the various schemas for a distributed database
- Splitting of one schema into two or more schemas
- Integration of two or more schemas into one schema

In this thesis the focus will be put on the latter of these examples.

4.2 A general Classification of Schema Comparisons

Here we will cover shortly some of the relations schemas can have between each other. This classification is a general comparison we have defined as a starting point constructed to start the discussion at a higher level of abstraction. Later in this chapter we will go into further detail.

Clearly a minimum of randomly chosen schemas compared with each other will actually be equivalent. They will rather be either disjoint from each other or something in between the two extremes(see fig.4.1).

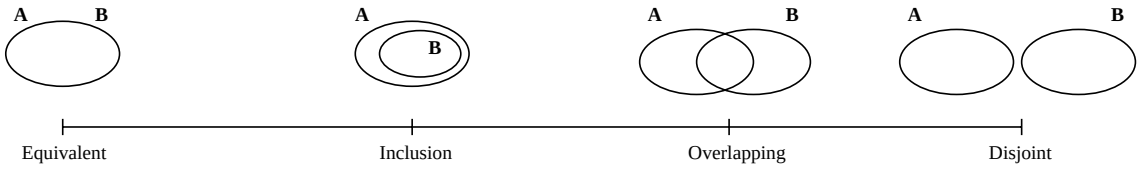

Figure 4.1: Scale of schema relationships

As fig. 4.1 shows, we can have four situations of schema comparison:

1. Equivalent: schema A and schema B are equivalent in some way we have agreed on.
2. Inclusion: All attributes in schema B are also attributes of schema A, however schema A has additional attributes[1].
3. Overlap: Schema A and schema B have a set of equivalent attributes, but each schema also has attributes of their own.
4. Disjoint: Schema A and B do not share any attributes.

Savasere et. al [SSG+91] present these situations in a classification(see fig. 4.2).

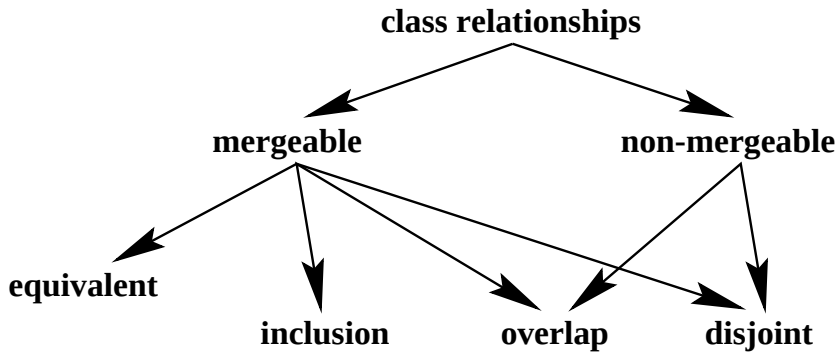

Figure 4.2: Classification

[1] Otherwise we would have situation 1.

They argue that this classification holds for a straightforward type of merging were one uses abstraction or subclassing in the case of inclusion and a one-to-one mapping in the case of equivalence. In the case of overlap or disjoint schemas we need to use more ad hoc approaches.

Note: The non-mergeable overlap class might seem strange since one could argue that all overlapping schemas have some possibility of merging. However, Savasere et. al include this class since they interpret it as the schema intergrator's choice whether it is *necessary* or not. Example: Regardless of whether `Course` and `Instructor` overlap or are disjoint, they need not necessarily be merged into a common class, even though they are related in the sense that instructors teach courses.

In their approach they exploit the notions of subsumption and classification for schema integration to automatically determine relationships among classes. They define subsumption and classification as follows:

Subsumption: A class f **subsumes** a class g if and only if every instance of g also is an instance of f, i.e. f is a superclass of g. The subsumption relationship is computed on the basis of whether the attribute constraints for class g logically imply the attribute constraints for class f.

Classification can be viewed as the process of correctly locating a given class in an existing taxonomy[2].

Let $E[f]$ and $E[g]$ represent the extensions of classes f and g, respectively.
We define the *subsume* function as:

$$subsume(f, g) = \text{ true iff } E[f] \supseteq E[g] \tag{4.1}$$

However, as explained above, subsumption is computed on the basis of class definitions and not the actual extensions. This means that the subsume function returns true if and only if the constraints on each attribute of g logically imply the constraints on the corresponding attribute of f. Therefore, $subsume(f, g) \equiv f$ includes $g \equiv g$ is-included-in f.

Here we present the definitions on each classification from figure 4.2 [SSG+91]:

Equivalence:

$$equivalent(f, g) = \text{ true iff } E[f] \equiv E[g] \tag{4.2}$$

Using subsumption it can be computed as

$$equivalent(f, g) = subsume(f, g) \wedge subsume(g, f) \tag{4.3}$$

Inclusion: Following from the above discussion we have:

$$subsume(f, g) \equiv f \text{ includes } g \equiv g \text{ is-included-in } f \tag{4.4}$$

[2]One way of computing classification is to take the transitive reduction over a boolean matrix generated by computing the subsumption relationship between all possible pairs of classes in the database schema, i.e. class taxonomy. Since there are n^2 such pairs, classification is an $O(n^2)$ algorithm where the fundamental unit of computation is the subsumption operation. However, computing subsumption is at least co-NP-hard[Neb88]

Overlap:

$$overlap(f,g) = \text{ true iff } \neg subsume(f,g) \wedge \neg subsume(g,f) \wedge \neg disjoint(f,g) \quad (4.5)$$

Disjoint:

$$disjoint(f,g) = \text{ true iff } E[f] \cap E[g] = \emptyset \quad (4.6)$$

Disjoint can be computed as:

$$disjoint(f,g) = \text{ true iff } incoherent(conjunction(f,g)) \quad (4.7)$$

where *conjunction* is a function which returns a new class from two given classes such that the extension of the new class is the intersection of the extensions of the given classes:

$$E[conjunction(f,g)] = E[f] \cap E[g] \quad (4.8)$$

incoherent is a boolean function which tests for logical inconsistency in constraints on the attributes of a given class.

We compute incoherence of a class by checking if the class is subsumed by a known incoherent class i:

$$incoherent(f) = subsume(i,f) \quad (4.9)$$

The functions defined above are based on the ability to compute subsumption, i.e. they are based on the semantics of class definitions. However, it is possible to define boolean functions which return true or false based on more syntactic criteria such as attribute relationships. Therefore, we define two additional operators:

1. $attr\_overlap(f,g)$
   This function returns `TRUE` if and only if there exists at least one pair of attributes a_1 and a_2 such that $a_1 \equiv a_2$ or $a_1 \subset a_2$ or $a_2 \subset a_1$ where a_1 is any attribute of f and a_2 is any attribute of g.

2. $attr\_disjoint(f,g)$
   This function returns `TRUE` if and only if there does not exist any pair of attributes a_1 and a_2 such that $a_1 \equiv a_2$ or $a_1 \subset a_2$ or $a_2 \subset a_1$ where a_1 is any attribute of f and a_2 is any attribute of g.

All these functions are argued to be computable automatically. The user can now restructure the global schema based on his perspective of the domain of discourse with the help of these functions.

4.2.1 More on Equivalence

In the database context the term *equivalence* has come up e.g. around talks on equivalence of databases. One has discussed what makes two databases equivalent in different contexts. Work done by [Øre92] enlightens this topic. In projects concerning database construction, it is often needed to verify equivalence of database schemas.

The term *equivalence* has different interpretations in different contexts. The term can mainly be divided in two main categories:

1. Equivalence of content
   (a) Syntactically
   (b) Semantically
2. Equivalence of behavior

In the former definition, the contents of the objects to be compared is taken into consideration. One way to view the contents is by syntactical comparison. In a database schema context this would be comparing attributes with one another, either in the same data model or in different models. Using different models would need a mapping tool of some kind to be able to compare directly. Another way to compare contents is by viewing its semantical meaning, but deciding that two schemas are equivalent based on a semantic consideration may prove not to be easy. Indeed, the semantical aspects of schemas are not fully understood as we shall discuss more in this thesis.

The latter category views our world as objects that interact with one another. These objects have methods or functions that the outside world has access to. The contents of the objects may or may not be visible. Two objects are considered equivalent if the behavior of the same actions on them are the same, i.e. the objects alter their state in an equivalent way when manipulated with equivalent methods.

4.2.1.1 Definitions of 'Equivalence'

A general, overall definition of the term *equivalence* could be the following [Øre92]:

Equivalence Two objects are equivalent for some purpose if they are interchangeable for that purpose.

This general definition is not very helpful to us, but it gives us a starting point of finding a suitable definition in this thesis. We have to be more specific about what we mean by *objects* and *purpose*.

4.3 A Schematic Classification of Heterogeneity

Kim and Seo [KS91] have developed a framework for enumerating and classifying the types of MDBMS structural and representational discrepancies. They assume in their classification that the canonical model in the MDBMS is the relational model, i.e. all local database schemas have been mapped to the relational data model. Later this classification has been expanded to also include aspects of the object-oriented data model in Kim et. al [KCGS95]. We take their classification a step further and adopt it to yield a classification of structural conflicts using a object-oriented model only as a canonical model.

The classification can roughly be be distinguished into three schema conflict groups and one data conflict group. The three schema conflict groups are: "Class-vs-Class", "Attributes-vs-Attributes", and "Class-vs-Attributes". The data conflict group is: "Different Representation for Equivalent Data". These four groups of conflicts correspond to our requirement list in table 1.1 which we presented in the introduction. In the following the conflict groups are presented in sections 4.3.1, 4.3.2, 4.3.3, and 4.3.4 respectively.

4.3.1 Class-vs-Class(RCG-1)

These conflicts occur when different component databases(CDBs) use different definitions to represent information in classes. Class-vs-class conflicts can be further decomposed into one-to-one class conflicts and many-to-many classes conflicts(one-to-one conflicts is a special case of many-to-many). Table 4.1 gives an overview of the conflicts in conflict group RCG-1.

Conflict Group RCG-1: Class-vs-Class
(a) One-to-One Class
i. Class Name
-different names for equivalent classes
-same name for different classes
ii. Class Structure
-missing attributes
-missing but implicit attributes
iii. Class Constraints
iv. Class Inclusion
(b) Many-to-Many Classes

Table 4.1: Conflict Group RCG-1

4.3.1.1 One-to-One Class Conflicts

These conflicts can occur when CDBs represent the same information in single classes using different names, structures, and constraints. We decompose these conflicts further

into class name conflicts, class structure conflicts, class constraints conflicts, and class inclusion conflicts.

4.3.1.1.1 Class Name These conflicts arise due to different names assigned to classes in different CDBs. There are two types:

- conflicts due to the use of different names for semantically equivalent classes (synonyms)
- conflicts due to the use of the same name for semantically different classes (homonyms)

Example: Two equivalent classes across the schemas in the case are `Under_Grad` in `Schma 1` and `Student` in `Schema 3`. Even though their attributes are not identical they represent the same real world class.

4.3.1.1.2 Class Structure These conflicts arise from differences in the number of attributes in CDB classes, i.e. when a class in one CBS is missing some attributes in a corresponding table in another CDB. A class is, among other criteria, not union-compatible with corresponding classes in other CDBs if it is missing some attributes. Missing attributes can be interpreted in two ways:

- The attributes are indeed missing and we have no information about them.
- The missing attributes are implicit of the other attributes and can thus be deduced from them.

Example: The attribute `address` of the `Student` class in `Schema 3` is apparently missing compared to the `Under_Grad` class of `Schema 1`. However the information is available in the separate class `Address` and thus may be derived from it by some appropriate resolution technique.

4.3.1.1.3 Class Constraints These conflicts arise from differences in the specifications of class constraints (such as key constraints). Unlike other constraints, which cause difficulties in the formulation of queries or in the definition of views involving multiple CDBs, constraint conflicts, including attribute constraint conflicts (discussed later), can cause difficulties with simultaneous updates to multiple CDBs. For example, if an attribute is a key attribute in one CDB, but the corresponding attribute in another CDB is not a key attribute, it is difficult to impose the key constraint on the attribute at the MDBS level.

4.3.1.1.4 Class Inclusion A class inclusion conflict arises from the generalization modeling abstraction in OODBSs. This type of conflict occurs when a class in one CDB is logically included in another class in another CDB. A simple example is the classes `Student` and `Grad_student`, taken from the case, which can induce a natural inclusion relationship in an MDB schema. A more complex situation occurs when an inheritance hierarchy from one OODB is to be integrated with a related inheritance hierarchy from another OODB that has a different structure.

4.3.1.2 Many-to-Many Classes Conflicts

These conflicts occur when CDBs use different number of classes to represent the same information. CDB designers typically define their classes in different ways for a variety of reasons. Therefore this type of conflict can occur frequently in an MDBS.

Example: We recognize this conflict type in the case where the concept of *employee* is modeled as the two classes `Employee` and `Emp_other` in `Schema 1`, but in `Schema 3` the concept is split up into three classes, namely `Employee`, `Emp_personal`, and `Emp_tax`.

4.3.2 Attribute-vs-Attribute(RCG-2)

These conflicts are caused by different definitions for semantically equivalent attributes in different component databases, including different names, attribute data types, and integrity constraints. Like the class-vs-class conflicts, these conflicts can be further decomposed into one-to-one and many-to-many conflicts. Table 4.2 gives an overview of the conflicts in conflict group RCG-2.

Conflict Group RCG-2: Attribute-vs-Attribute
(a) One-to-One Attribute
i. Attribute Name
-different names for equivalent attributes
-same name for different attributes
ii. Attribute Constraints
-integrity constraints
-data values
-composition
iii. Default Values
iv. Attribute Inclusion
v. Methods
(b) Many-to-Many Attributes

Table 4.2: Conflict Group RCG-2

4.3.2.1 One-to-One Attribute Conflicts

These are due to different definitions for semantically equivalent attributes in different classes. We decompose one-to-one attribute conflicts into attribute name conflicts, attribute constraint conflicts, default value conflicts, attribute inclusion conflicts, and methods conflicts.

4.3.2.1.1 Attribute Name Attribute name conflicts are similar to the class name conflicts discussed earlier. There are two types in this category:

- one arising from the use of different names for semantically equivalent attributes in different CDBs (synonyms)
- one arising from the use of the same name for semantically different attributes (homonyms)

The latter type is often caused by the use of incompletely specified names. For example, one CDB uses an attribute name *salary*, meant semantically as gross salary, and another CDB uses the same name for net salary. Similarly, attribute name *price* may represent the price including VAT in one CDB and the price before adding VAT in other CDBs.

Example: An example of the first from the case is the attribute `major` from the `Grad-student` class and the `dept` attribute from the `Faculty` class of `Schema 4` which have the same meaning but different names.

4.3.2.1.2 Attribute Constraints These conflicts are further decomposed into integrity constraints conflicts, data type conflicts, and composition conflicts.

Integrity Constraints This type of conflict, will often show to be a problem during an update to the MDBS. It arises from the differences in defined constraints in different CDBs. Depending on what constraint has been adopted by the MDBS, the update may not succeed in one of the CDBs.

Data Type These conflicts occur when semantically equivalent attributes in different CDBs have different domain or type. For example, taken from the case, the attribute `ssn` has the type `string` in `Schema 5` but `integer` the the others. A more general conflict arises when integrating OODBs, because an attribute can have a user-defined type based on some class. This latter example can be regarded as part an aggregation conflict.

Example: The `dept` attribute of `Faculty` in `Schema 4` is of type `string` while the corresponding attribute in `Schema 5` is of the user-defined type `Department`

Composition Attribute composition conflicts arise when similar concepts are represented in one CDB data model as an *aggregation* or *composition* abstraction, while not in the other CDB.

Example: In `Schema 4`, consider the attribute `course` of the class `Enroll` with the domain of the user-defined type `Course` which in turn has an attribute `prereq` whose domain is `SET_OF(Course)`. Compare this with the corresponding classes in `Schema 3`.

Note that this type of conflict is different from the many-to-many class conflicts because it is not the difference in the number of classes involved, rather it arises because non-object-oriented models often do not support the aggregation abstraction. However, conflicts between related aggregation hierarchies in more than one OODB can be thought of as a special case of the data type conflict mentioned above.

4.3.2.1.3 Default Values This conflict type, like constraint conflicts, can manifest itself during update. It occurs when there are different choices of default values for attributes in different CDBs. Updates to the MDBS view may result in a conflict of what default values to insert in the CDBs if the value is not explicitly specified in the MDBS update.

Example: The `bonus` attribute in `Schema 1` might have a default value of 10% while in `Schema 3` the same attribute might have no default value, but it is rather expected that some value be provided at every update.

4.3.2.1.4 Attribute Inclusion This conflict arises when an inclusion relationship exists between two or more attributes. For example an attribute `son_name` can be regarded as being included in `child_name`. Clearly, this conflict falls into a category different from different names or data types. An inclusion relationship between two attributes may be used to induce a natural inheritance hierarchy among the corresponding classes in the MDB schema.

4.3.2.1.5 Methods Since a method declaration is part of the definition of an OODB class, methods can be treated just like an attribute. For example, when two classes E_1 and E_2 are identical except for a missing method, for our purposes, we may regard one class as missing an attribute. Likewise, if two classes have methods with different names but equivalent semantics, the situation can be considered as identical to the attribute name conflict. When methods have arguments with different types, the two methods may be integrated by considering the data type conflicts between the corresponding arguments. In some sense, this situation may be seen as similar to an attribute composition conflict. When integrating an inheritance hierarchy, if a specializing method is defined in a subclass, the situation is analogous to an attribute inclusion conflict.

Example: In the case only `Schema 4` and `Schema 5` have the method `gpa()`. The others are missing it and they can in this case be regarded as missing an attribute. However the corresponding methods `gpa()` in `Schema 4` and `Schema 5` can differ in number of arguments and the type of the arguments such that some matching method must be provided to resolve the conflict.

4.3.2.2 Many-to-Many Attributes Conflicts

This category of conflicts arises when the CDBs use a different number of attributes to represent the same information. As remarked for the many-to-many class conflicts, these conflicts may combine many-to-many attribute conflicts with sub-categories of one-to-one attribute conflicts. These occurrences can be interpreted as the compound conflicts mentioned earlier and further decomposed into several types of basic conflicts.

Example: A simple case of this conflict is represented in the case by the name information for students being broken into a `lastname` and `firstname` in `Schema 3` and `Schema 5` while it is simply `name` in the other CDBs.

4.3.3 Class-vs-Attribute(RCG-3)

These conflicts occur if some CDBs use classes and others use attributes to represent the same information. This conflict type can be regarded as a combination of many-to-many class conflicts and many-to-many attribute conflicts. We present conflict group RCG-3 in table 4.3.

Conflict Group RCG-3
Class-vs-Attribute

Table 4.3: Conflict Group RCG-3

Example: This conflict type is typically represented in the case by the address information. The attribute `address` represents this information in `Schema 1` while the same information is represented as a class in `Schema 3`.

4.3.4 Different Representation for Equivalent Data(RCG-4)

There are three different aspects to the representation of data across CDBs: expressions, units, and precision. Table 4.4 gives an overview of the conflicts in conflict group RCG-4.

Conflict Group RCG-4: Different Representation for Equivalent Data
(a) Different Expression denoting same Information
(b) Different Units
(c) Different Levels of Precision

Table 4.4: Conflict Group RCG-4

4.3.4.1 Different Expression denoting same Information

Conflicts in expressions arise between CDBs when they use the same type of data or different types of data for the same information. The following examples show various expressions for the same data:

- Different words for the same data: `Oslo`, `OSL`, `Osl`
- Different strings for the same data:
  `Forskningsveien 1, Room 335A(Third floor), 0314 Oslo`
  or
  `Forsknvn.1, 3-335A, 0314-O`

- Different codes for the same data:
  `*****,A,Excellent,1,5,S`
  `****,B,Good,2,4,Mg`
  `***,C,Fair,3,3,G`
  `**,D,Poor,4,2,Ng`
  `*,E,Bad,5,1,Lg`

The latter shows an example of different data types used for the different representations of the same data.

4.3.4.2 Different Units

These conflicts arise when CDBs use different units for numeric data. Different units give different meanings to numeric values, as in the attribute `weight` with value 3 meaning three pounds in one CDB and meaning three kilograms in another CDB.

This conflict type can, in a sense, be regarded as an attribute name conflict. Thus, if a fully qualified name is used for each attribute(e.g. `weight_in_lb` and `weight_in_kg` respectively), the attributes in different units can be regarded as distinct attributes. However we regard attributes in different units as carrying semantically equivalent information.

4.3.4.3 Different Level of Precision

Conflicts in precision occur when CDBs use values from the domains of different cardinalities for the same data.

Example: Suppose the `grade` attribute in `Schema 1` had its value given along a scale from 1 to 100. If the corresponding `grade` attribute in `Schema 3` was given along an alphabetic scale from A to E the precision conflict is obvious and some sort of range from the more precise scale would have to correspond to each value of the less precise. Note that we will lose some information in doing this.

We note that when different CDBs use different values from domains with same cardinalities, they are in expressions conflict rather than in precisions conflict, as in the third example of "Different Expression denoting same Information".

4.3.5 Compound Conflicts

We regard compound conflicts as combinations of different conflict types in this classification. This approach makes it possible to classify arbitrarily complex conflicts as we can decompose them into the basic conflict types of this classification.

4.4 A Semantic Proximity Approach

Apart from trying to map equivalent objects on a one-to-one syntactical basis, one can define a semantic measure for equivalence of objects. Here we present a semantic proximity function, first introduced by Sheth and Kashyap [SK92], to define a measure of comparison of schemas.

4.4.1 The `semPro` Function

To provide a classification of semantic similarities, we here present the concept of *semantic proximity* to characterize semantic similarities between objects. The above classifications are mainly focused on structural aspects in schemas. The following work is a semantic approach to schema comparison.

Webster's 7th edition dictionary defines semantics to be "the meaning or relationship of meanings of a sign or set of signs." The *real world* and the *model world* differ in that the *model world* is a representation of *real world*. It seems clear it would be an impossible task to completely define what an object of interest denotes or means in the *model world* [SG89]. However, it is still possible to introduce a certain level of formal reasoning as to considering an object's semantics. The following is an approach toward this.

The semPro function was developed by Sheth and Kashyap [SK92]. As the name suggests it is a function that qualitatively measures semantic proximity. It is a function between two objects based on four concepts: *context*, *abstraction*, *domain*, and *state*. A context is where the objects are compared, abstraction is a mapping between the objects' domains and the state is the current value of the objects.

4.4.2 Definition

Given two objects O_1 and O_2, the *semantic proximity* between them is defined by the 4-tuple given by

$$semPro(O_1, O_2) =< Context, Abstraction, (D_1, D_2), (S_1, S_2) > \tag{4.10}$$

where D_i is domain of O_i and S_i is state of O_i.

Context Every object is interpreted according to its given context. The context of semantic proximity is where semantic proximity holds. Two objects may be semantically closer in some contexts than in others. A context can have many representations, but in this formal reasoning we are interested in representing and reasoning about context as an explicit concept.

In this classification scheme, we are often interested in the cases where the context of the objects under consideration can be determined to be one of the following:

- **ALL**, i.e. the `semPro` of the objects is being defined with respect to all possible contexts.

- **SAME**, i.e. the `semPro` of the objects is being defined with respect to the same context.
- **SOME**, i.e. the `semPro` of the objects is being defined with respect to more than one context.
- **SUB-CONTEXTS**, when the `semPro` can be defined in a previously defined context that is further constrained.
- **NONE**, i.e. the objects under consideration do not exhibit any useful semantic similarity under any context.

Abstraction We can refer to the mechanism of mapping the domains of objects to each other or to the domain of a common third object as *abstraction*. Useful abstractions can be:

- A **total 1-1 value mapping** between the domains of the objects, i.e. for every value in the domain of one object, there exists a value in the domain of the other object and vice versa. Also there is a one-to-one correspondence between the values of the two domains.
- A **partial many-one mapping** between the domains of the objects. In this case some values in either domain might remain unmapped, or a value in one domain might be associated with many values in another domain.
- The **generalization** abstraction to relate to domains of the concerned objects. One domain can generalize/specialize the other, or domains of both the objects can be generalized/specialized to a third domain
- The **aggregation** abstraction to relate the domains of the objects. This can be expressed as a partial, 1-1 mapping between the cross-product of the domains of the objects being aggregated and the domain of the aggregated object.
- **ANY**, is a special term used to denote that any abstraction such as the ones defined above may be used to define a mapping between two objects.
- **NONE**, is a special term used to denote that there is no mapping defined between two semantically related objects.
- **NEG**, is a special term used to denote that there is no mapping possible between two semantically unrelated objects.

Domain A *domain* is referred to as the set of values an object can take its values from. Domains can be further decomposed, called a composite domain or it can be atomic, i.e. can not be decomposed any further. The "leaf nodes" of such a structure are atomic ones.

State An object is said to be in a particular state according to its stored values. Objects typically change state whenever they are updated or otherwise manipulated. It is important to note that two objects with different model values can be equivalent in a real world context.

As the structural definitions of schematic similarities were given earlier, we will here give a semantic comparison overview based on the semPro function.

4.4.3 Semantic Equivalence

In the semantic proximity measurement *semantic equivalence* is the strongest meaning of how close two objects are. We say that two objects are said to be semantically equivalent if they represent the same real world concept or class. This means that there should be a one-to-one mapping between the domains of the two objects in any and all contexts. In semPro we can express this by :

$$semPro(O_1, O_2) =< ALL, \text{ total 1-1 value mapping}, (D_1, D_2), - > \tag{4.11}$$

This variant of semPro could also be called *domain semantic equivalence* because it depends on the definitions of the domains of the objects. [3]

Example: Synonyms – The attributes objects are semantically alike but with different names. Mapping can be established between them with respect to all contexts. Therefore, the two objects can be considered semantically equivalent. This can be found between the `Student` classes of `Schema 4` and `Schema 5`.

4.4.4 Semantic Relationship

We say that two objects are *semantically related* to one another when given O_1, we can identify O_2 but not vice versa. We find this situation when there exists a partial many-one mapping between the domain of the objects or a generalization or aggregation abstraction(which could be thought of as a many-one relation). This is a relaxation of the equivalence requirements, but the context requirements remain the same and thus we can define *semantically relationship* in semPro as:

$$semPro(O_1, O_2) =< ALL, M, (D_1, D_2), - > \tag{4.12}$$

where M = partial many-one mapping, generalization, or aggregation

Example: The attributes of two objects might have a precision conflict as described earlier. There may be a one-to-one or many-to-one mapping from the domain of the precise attribute to the one of the coarse attribute with respect to all contexts. The objects can in this case be considered to have a semantic relationship.

4.4.5 Semantic Relevance

If two objects can be related using some abstraction in the *same* context we say that the two objects are *semantically relevant*. The context dependency means that two objects may be semantically relevant in one context, but not in another. However, any abstraction will hold for this proximity measure and we define it as

$$semPro(O_1, O_2) =< SAME, ANY, (D_1, D_2), - > \tag{4.13}$$

[3] Sheth and Kashyap[SK92] also mention a stronger notion of equivalence calling it *state semantic equivalence* which takes under consideration the states of the databases the objects belong to. This is why the state parameters are included in the definition. We include the state parameters for the completeness of Sheth and Kashyap's definition, but do not use them any further.

Example: Suppose we in the case introduced a `Person` class, and the `Student` and `Employee` classes were subtyped from the `Person` class. For the simplicity of this example assume further that `Student` and `Employee` are semantically incompatible. The semantic relationship and possible mappings between a `Person` class and a `Student` class would vary depending on what context the `Person` had. If the `Person` class actually was considered as a student in some context then the `Person` class and the `Student` class are equivalent, but if the `Person` class is considered an `Employee` then it will be semantically incompatible with the `Student` class. This example shows that the `Person` class and the `Student` class only can have a defined mapping if they are considered to be in the same context, thus they are semantically relevant to each other.

4.4.6 Semantic Resemblance

The weakest measure of semantic proximity we call *semantic resemblance*. It considers a somewhat difficult form of semantic proximity, i.e. difficult to specify. Semantic resemblance considers the case where the domains of two objects cannot be related to each other by any abstraction in any context. To be able to specify this type we need an aspect of context, which we will call *role*. *role* is a binary function mapping an objects participation of an object in a context to a role name.

$$role - of : object \times context \rightarrow rolename$$

With this function we define *semantic resemblance* as

$$semPro(O_1, O_2) =< Context, NONE, (D_1, D_2), - > \tag{4.14}$$

where $Context = context(O_1) \cup context(O_2)$
and $D_1 \neq D_2$
and $role - of(O_1, Context) = role - of(O_2, Context)$

Example: Suppose the `Employee` class has an attribute `hPrice1` which represents the hourly fee a customer would have to pay for the associated employees work. Suppose a different schema has the same attribute, `hPrice2`, for an `Employee` class. The former schema may have a constraint attached to the `hPrice1` attribute stating that no employee may charge more than NOK 650 hourly, while the latter schema may have a constraint on the corresponding attribute stating that no employee may charge less than NOK 700 hourly. The two constraints are incompatible with each other. But nevertheless these two attributes play the same role in the two schemas which make the attributes semantically resemble each other where

$Context = context(hPrice1) \cup context(hPrice2)$
and $Domain(hPrice1) \neq Domain(hPrice2)$
and $role - of(hPrice1, Context) = role - of(hPrice2, Context) = StandardPrice$

4.4.7 Semantic Incompatibility

As we have defined measurements for some degree of semantic similarity we also include a variant of *semPro* that describes *semantic incompatibility*. It describes the lack of any

semantic similarity. This doesn't automatically imply that the considered objects are semantically incompatible. The requirements to establish semantic incompatibility are that there is no context nor abstraction in which the domains of the objects can be related and that the two objects cannot have similar roles in the context(s) in which they exist. This can be expressed by:

$$semPro(O_1, O_2) =< NONE, NEG, (D_1, D_2), - > \tag{4.15}$$

where $context_1 = context(O_1)$ and $context_2 = context(O_2)$
and $Abstraction = NEG$, signifying dissimilarity
and D_1 may or may not be equal to D_2
and $role - of(O_1, context_1) \neq role - of(O_2, context_2)$

Example: Homonyms – The attributes are semantically unrelated but with the same name. Thus, there cannot be any context in which an abstraction maps one to the other, and they are considered semantically incompatible.

4.4.8 A Semantic Proximity Taxonomy

The classifications we have discussed form a taxonomy [SK92], from weakest to strongest semantic similarity. Figure 4.3 shows the taxonomy as a directed graph with assignments attached to the edges.

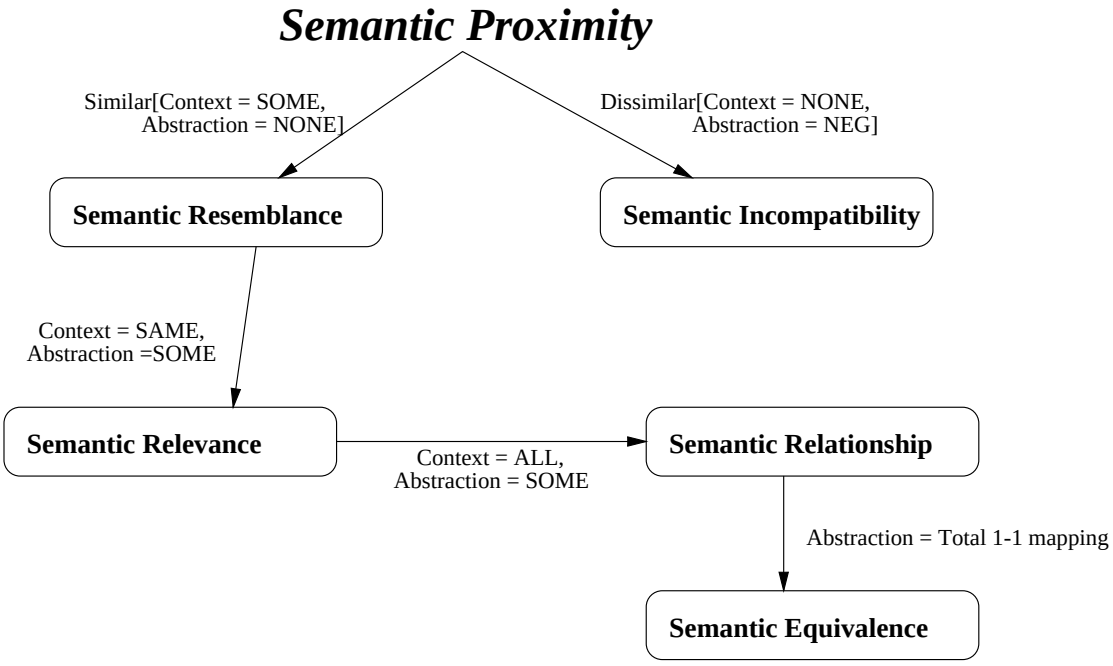

Figure 4.3: Semantic proximity: a taxonomy

Structural conflicts don't always have a general semantic category they correspond to. However table 4.5 sums up the most likely semantic categories of some of the structural conflict groups [Kor94].

Comparison Structural/Semantical	
Synonyms	Semantic equivalence
Homonyms	Semantic incompability
Data representation conflicts	Semantic equivalence
Data unit conflicts	Semantic equivalence
Data precision conflicts	Semantic relationship
Default value conflicts	Semantic relevance
Integrity constraint conflicts	Semantic resemblance

Table 4.5: Structural conflicts and their semantic proximity

4.5 Summary

This chapter presented two views of schema heterogeneity; schematic and semantic. The emphasis was put on the structural conflicts, but in interpreting these we had to use semantics alongside them. The schematic conflicts were presented as an adopted classification from Kim et. al [KCGS95] and its basic structure conformed to the requirement conflict groups we defined in table 1.1. The semantic viewpoint was presented by a semantic measure called `semPro` which categorized objects' similarities according to different parameters than our structural classification did. We will come back to both the structural classification of conflicts and the `semPro` function in chapter 7 where we will suggest how to resolve the structural conflicts found and how we might be able to use `semPro` to aid this process.

In the initial chapters of this thesis we have discussed the background theory for our problem area. In the next chapter we will step into the real world and see some approaches to multidatabase systems have been implemented and how they handle integration.

Chapter 5

Object-Oriented Multidatabase Systems

The first chapters in this thesis have given a thorough background for our problem area. It is important not to forget the real world and its limitations and therefore we here present some existing prototypes and project systems that address the multidatabase system effort. This is to get an idea of different approaches towards our problem area and to realize how theory is one thing while real life is another. We hope to get some ideas on how our own proposal could be built, but realize that the level of detail we have investigated our problem area is too deep to really be able to investigate these systems at the same level of detail. Nevertheless, the systems that are given an overview show how the first attempts of how the multidatabase systems are being approached and serve as possible guidelines to how future commercial systems might operate.

Several multidatabase systems have been developed or are under development. Here we present various systems where we focus on these main dimensions:

- System architecture
- Common data model
- Translation model and integration aspects.

The systems are limited to where the common data model(CDM) is object-oriented or the system uses object-orientation in its management of the global and underlying systems.

5.1 Pegasus

Pegasus [AAD+93, ADD+91, ADK+91] is an object-oriented multidatabase system being developed at Hewlett-Packard Laboratories.

The goal of the system is to provide facilities for multidatabase applications for accessing and manipulating multiple autonomous heterogeneous object-oriented, relational and other databases. A *native* database is created in Pegasus, and both its schema and

data are managed by Pegasus. *External* databases are accessible through Pegasus, but not directly controlled by it.

The focus of Pegasus is thus in the area of multi-model data integration.

5.1.1 System Architecture

Pegasus provides three functional layers:

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

The architecture is outlined in more detail in figure 5.1.

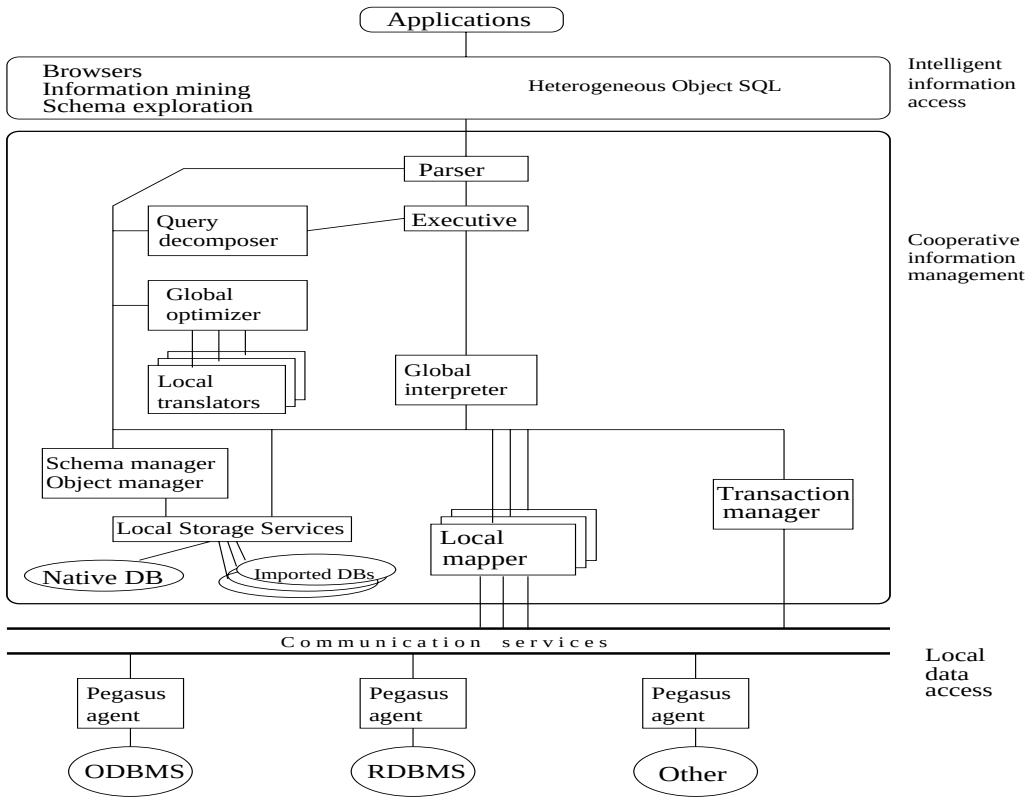

Figure 5.1: Pegasus architecture

Pegasus takes advantage of the encapsulation mechanisms of object-oriented programming by hiding the heterogeneous aspects of various systems in the implementation part of

types. Different external databases might be attached to the local native database which contains the schema of the local database represented in the canonical object-oriented data model of Pegasus.

5.1.2 Common Data Model

The common data model is based on the IRIS object-oriented model. It consists of three basic constructs: objects, types and functions. The types represent what we also know as classes. The types are organized in a hierarchy supporting inheritance(also multiple), generalization and specialization. Object properties, relationships between objects and computation on objects are expressed in terms of functions.

Pegasus introduces a language called HOSQL(Heterogeneous Object SQL), and it serves both as a data definition and data manipulation language. HOSQL provides non-procedural statements to manipulate multiple databases. It also provides for attachment and mapping of schema of local databases. Support is provided for specification of types and functions in this language and these specifications can be imported from underlying databases.

5.1.3 Translation and Integration

Schema integration is done on the canonical data-model layer. HOSQL provides mechanisms for importing a schema from a participating database, as mentioned. It is possible to define a type as a generalization of two underlying types. It is then possible to give criterias for equivalence between two objects from different sub-types. Techniques for dealing with domain mismatches have been investigated.

Pegasus represents external databases by *imported schemas*. The translation from the native schema to the imported schema and the importation of the external schemas are performed in a single step, using the view mechanism of the HOSQL language. Importation of an external data model can be done by developing a separate module and installing it independently of other external models.

Pegasus provides integration with a basis of distinguishing the views of the data administrator and the end user. Two kinds of types are defined, *unifying* types and *underlying* types. Each underlying type has a unifying type. The initial assumption is that every type is its own unifier. Pegasus supports *unifying inheritance*, i.e. every function defined for a type is also defined for its unifying type. Resolution problems are resolved explicitly by the administrator who defines a *reconciler* algorithm for each overloaded function. This algorithm specifies which function that will be used.

Pegasus handles the following types of conflicts:

- **Semantic conflicts** are handled by defining appropriate functions at the unifying type.
- **Naming conflicts** referring to function synonyms are solved by defining aliases. Additionally, names of functions and types can be prefixed by their database names to prevent ambiguities.

- **Structural conflicts** can be handled by defining adequate imported functions.
- **Identity conflicts** are resolved by allowing the user to specify equivalences among objects.

5.2 VODAK

ViewSystem [KDN91] is an object-oriented environment that has been developed as a first prototype of a project at GMD-IPSI. The project is called KODIM[1] [KFM+96, KDN91] and is mainly concerned with the dynamic integration of heterogeneous and autonomously administrated information bases. ViewSystem provides an object-oriented query language with extensive view facilities for defining virtual classes. The following describes a later development of this project.

5.2.1 System Architecture

The basis modules of the system architecture are:

- A **transaction manager**, which provides services for processing transactions.
- A **message handler**, which is responsible for exchanging messages between objects.
- A **communication manager**, which is used by the message handler to send the message to a component system.
- An **object manager**, which creates more complex objects by combining objects of the underlying storage system, handles persistent and non-persistent global objects and dynamically loads and stores objects from and to the underlying systems.
- **Query processor and compiler components**, which compile global schema definitions onto internal representations.
- A **schema integrators workbench**, which lays on top of the above modules and provides for the integration of export schemas and for the construction of integrated views.

5.2.2 Common Data Model

The common data model, called VML (VODAK Model Language[DKT88], consists of classes, objects and object types, structural properties and methods. A class describes the structure and behavior of a collection of similar objects, called the extension of the class. The extension of a class is defined by the commonly known instantiation of the class. Each class has an associated object type that defines the structure and behavior of the instances of the class. Different classes can share the same object type.

[1] Knowledge-Oriented Distributed Information Management

VML supports *application classes* and *metaclasses*. The application developer defines the application classes to organize and classify the objects dealt with by the application. The system administrators and application developers define the metaclasses to organize the application classes and to make sure the model can meet the requirements of a task. Metaclasses are used to described the common structure and behavior of the application classes and their instances.

5.2.3 Translation and Integration

Local schemas are connected to the system and translated to the VODAK model. The export schemas of the local systems are defined by metaclasses that define interfaces to their modeling constructs. *Augmenting* transformations are performed on subschemas of the local databases to overcome structural heterogeneities. The system distinguishes between four types of augmenting transformations:

- Augmentations that use independent properties of a class to generate *roles* of the class.
- Augmentations that introduce *additional abstractions*.
- Augmentations that use a categorizing property to generate *categories* of the class.
- Augmentations that introduce a category *generalization* for properties.

A semi-automatic method is supported by the system where the user defines the correspondences between schemas and then the augmenting transformations are automatically generated by the system. After this homogenization, corresponding classes are combined by generalization, and corresponding properties are combined by user-defined methods to form the actual integrated schema. During this last phase, possible data conflicts are also resolved by user system defined methods.

5.3 SISIP

SISIP – A Systems Integration Platform based on Distributed Persistent Objects [BHR+95] is a framework under development at the Department for Informatics at SINTEF. It is a distributed heterogeneous object management system with support for heterogeneous implementations for objects, and an object model which unifies concepts from distributed systems, database systems and object-oriented systems.

5.3.1 SISIP architecture

SISIP is an integration architecture that takes an object-oriented approach to the following integration areas:

Control Integration: The degree to which tools are able to interact with each other. It can be further refined into two subareas:

- *Request-oriented* is the extent to which tools are able to interact directly with each other, by requesting and providing functional services.
- *Notification-oriented* is the extent to which tools are able to interact by sending out notification about certain events. These notifications may be picked up by other tools that have registered interest in them.

Data Integration: The degree to which tools are able to share common data. It can be further refined into two subareas:

- *Single-model* is the extent to which tools are able to share common data and information that are stored and manipulated through one single data model and a corresponding storage service.
- *Multi-model* is the extent to which tools are able to share common data and information that are stored and manipulated through multiple data models and corresponding storage services.

Presentation Integration: The degree to which a user-interface program might provide the access to the functionality needed by the user through a uniform look and feel. It can be further refined into two subareas:

- *Display-oriented* is the degree to which a common look-and-feel is provided by the tools which are used.
- *Model-oriented* is the degree to which the functionality presented through the display is accessed and combined from one or more underlying functional models.

Process Integration: The degree to which a user's working process and use of tools can be guided by a model of the work process and the methodology to be followed, possibly in cooperation with other users. It can be further refined into two subareas:

- *Interaction-oriented* is the extent to which the user's working-process and use of tools can be guided by a model if the work process and the methodology to be followed.
- *Inter-working-oriented* as the degree to which group-work and inter-working between people is supported.

5.3.2 Common Data Model

SIOM is the SISIP Object Model. It is a fully object-oriented model, as defined in [Dit86], as a merge of structurally and behaviorally object-oriented models (EXPRESS, ODMG object model, OMG IDL). Initially SIOM has language bindings to C++ and Smalltalk, both being object-oriented.

Here are some of the principles for SIOM:

- The representation of functionality and data of heterogeneous systems and databases as encapsulated objects.
- All interaction happens through messages sent to encapsulated objects.
- A set of objects "belonging" together can express certain semantics, i.e. attributes, relationships.
- The use of three languages: SIODL, SIOML and SIOQL.
- The separation between interface, implementation and extent.
- The representation of run-time information about interfaces, implementations and extents.

5.3.3 Translation and Integration

The SISIP framework supports a pool of distributed objects that serves as the conceptual model the application programmer relates to (see fig. 5.2). It should be transparent which

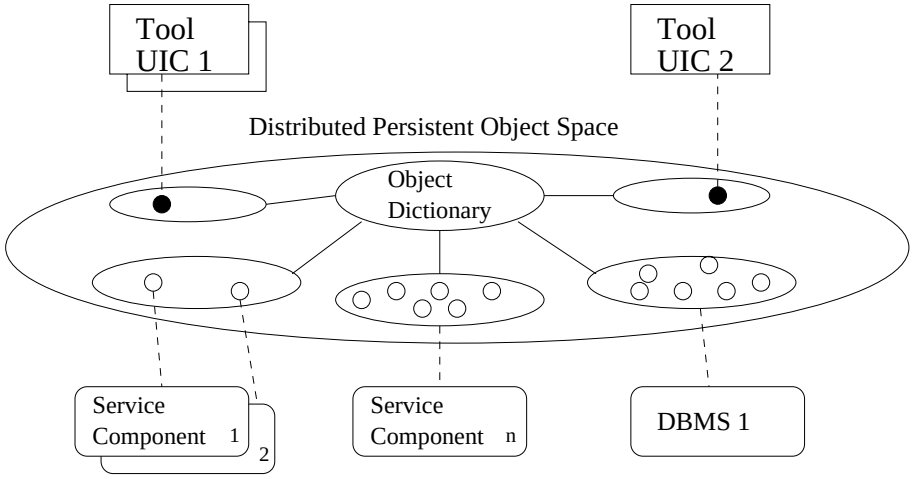

Figure 5.2: Integration through a distributed persistent object space in SISIP

systems in the underlying connected systems, the objects might represent information and functionality from. The distributed objects represent the totality of functionality and data available as objects.

The SIOM model introduces abstract attributes and relations. Together with the OQL they serve as a foundation for the support of integration. The separation between interface and implementation allows several implementations for an interface which in turn supports mapping to different types of constructs in the underlying systems and also supports behavioral integration.

Implementation managers manage the implementation of objects from the underlying systems and the *interface managers* manage implementation managers that might represent classes with the same concepts, e.g. a METAEmployee class could manage an EmployeeRDB and an EmployeeOODB implementation manager thereby supporting the integration of employees from two underlying systems.

5.4 The EIS/XAIT OMS Project

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

5.4.1 Common Data Model

The common data model is called FUGUE. It is an object/function model that consists of three basic constructs:

- Objects
- Functions
- Classes – called types.

The model does not support class hierarchies.

5.4.2 Integration

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

5.5 DOMS

DOMS, Distributed Object Management System [MHG+92, BOH+92], is being developed at the GTE Laboratories. It is an object-oriented environment in which heterogeneous and autonomous local systems can be integrated and native objects can be implemented. The local systems can be different systems like e.g. conventional systems, hyper-media systems and application programs.

5.5.1 System Architecture

The DOMS architecture is based on the general principles of the distributed object-based architectures. Object managers are implemented as *DOMs*. A local application interface (LAI) provides an interface between a DOM and a local system that allows the DOM to access local data and the local system to make requests to access objects from other local systems or to use DOM services.

5.5.2 Common Data Model

The common data model is called Functional/Relational Object-Oriented Model (FROOM). The CDM consists of three basic constructs:

- Objects
- Functions – which model both state and behavior
- Types – The subtype relation is determined implicitly; any type that has the interface required by a type T is implicitly a subtype of T.

Objects of the same type may support different implementations of the same function due to the supported distinguishing between implementation and interface.

The definition of FROOM includes an object algebra that resembles an extended relational algebra. The object algebra includes a set of high-level functions, which are defined for collections of objects and create new collections as results.

5.5.3 Integration

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

5.6 Carnot

Carnot is a project at Microelectronics and Computer Technology Corporation (MCC) [HJK+92, WCH+93, TLM+92, WSHC92]. It addresses the problem of logically unifying physically distributed, enterprise-wide heterogeneous information, coming from different systems, such as database systems, database applications, expert systems/knowledge bases, business workflows and the business organization itself.

5.6.1 System Architecture

Carnot has developed and assembled a large number of generic facilities. These facilities are organized into five sets of services:

- *Communication services* provide the user with a uniform method for connecting heterogeneous equipment and resources.
- *Support services* implement basic network utilities. A distributed shell environment is a central component. This shell environment is called extensible service switch (ESS) [TLM+92] and it provides interpretive access to communication resources, local information sources and applications at a local site.
- *Distribution services* support relaxed transaction processing and a distributed agent facility.
- *Semantic services* provide a global view of all the resources integrated within a Carnot-supported system.
- *Access services* provide mechanisms for manipulating the other four Carnot services.

5.6.2 Common Data Model — Translation and Integration

Carnot considers the integration of knowledge based systems and process models in addition to database schemas. Instead of translating the local systems schemas into a common data model, Carnot compares and merges them with Cyc [CHS91], a common sense knowledge base. Cyc has knowledge about most data models and about the relationships among them in addition to its common sense knowledge of the world. The common language is called global context language (GCL).

Integrating a resource is done by specifying a syntax and a semantics translation between the resource and the global context. The syntax translation produces a bidirectional translation between the local resource management language and GCL. The semantics translation is a mapping between two expressions in GCL that have equivalent meaning. The model integration software tool (MIST) is a graphical tool that automates some of the routine aspects of model integration.

5.6.3 Object-Orientation in Carnot

Carnot does not follow any of the three dimensions of object-orientation listed in the introduction to this chapter, it rather uses object-orientation in the implementation of its various tools (e.g the ESS is an actor object).

5.7 Other Systems

We have also investigated a few other systems, but they did not show to be schema-integration-relevant enough to be included in the same manner as the systems above.

They do, however, have some features that in general are interesting to multidatabase systems, so we include them in short here and also mention them in the summary of this chapter.

CIS [BGN+88, BGN+89] (Comandos Integration System) is part of the ESPRIT project COMANDOS. Several different application environments(e.g. RDBMSs, graphical databases, public databanks) have been integrated using this system.

FBASE [Mul92] is a decentralized heterogeneous object-oriented database system. A prototype has been implemented at the InterBase Lab at Purdue University.

InterBase* [ME93] is being implemented as part of the InterBase project at Purdue University[BCD+93]. It supports global applications accessing many local systems, such as SAS, Sybase, Ingres, D2 and Unix utilities.

The *A la carte* framework [DKH92] is part of the University of Colorado's L'Heureux toolkit, a set of tools addressing interoperability at different system levels. The framework provides a reusable and extensible architecture in which a set of heterogeneous database management systems can be integrated.

5.8 HKBMS

Heterogeneous Knowledge Base Management System (HKBMS) [SDS96] is a system being implemented at the Database Research and Development Center of the University of Florida. This work investigates the problem of the integration of multiple heterogeneous rule-based systems and a database management system. The HKBMS system currently consists of three expert systems and an INGRES relational DBMS.

The HKBMS System doesn't really fit in with the framework of our thesis, but it is an interesting system that touches upon some of the same problems we are dealing with, but in a related research field, so we will include a description of it.

5.8.1 System Architecture

The architecture of HKBMS is generally composed of two layers of managers:

- The global knowledge administrator module (GKAM) at the global level.
- The Multiple local knowledge administrator modules (LKAMs) at the local level.

Global information resources are defined by a global knowledge schema (GKS) and a function graph (FG).The GKAM manages this information. The GKS is the user's view of the integrated knowledge bases and defines all data items and their relationships as seen by the user, and the FG defines the relationships among variables referenced in the rules of the component systems.

5.8.2 Common Data Model

The global view is defined by the Object-Oriented Semantic Association Model (OSAM*). This model integrates the concepts of semantics modeling and the object-oriented paradigm. In contrast to the normal object-oriented inheritance concept, classes in the OSAM* are related by five predefined *associations*; Aggregation, Generalization, Interaction, Composition and Close-Product. Further classes consist of a specification and an implementation part. The specification part is an addition to the methods' definitions and includes the association of the class with other classes. The implementation part is the procedures that implement the methods. Two types of classes are distinguished:

- Entity classes – contain a set of instances that are explicitly created and modified by the users of the database, and have OIDs associated with them.
- Domain classes – which are virtual classes which are type declarations where the values that satisfy the declaration are self-naming and have no OID assigned.

5.8.3 Integration

All the classes defined in the local systems and the associations between them are included in a global schema (GKS). *Knowledge derivation paths* and *triggering conditions* keep track of the relationships between the classes. The knowledge paths specify what data item can be derived from what data types and the triggering conditions indicate under what conditions to activate the associate knowledge paths.

The function graph (FG) is a merge of all the rules in the component expert schemas. The nodes of FG represent an attribute used in a rule and the edges represent the rules in the integrated rule base. Common rules or cooperation among the component expert systems will be reflected in shared nodes in the graph which accordingly are associated with several expert systems. Path optimization is also sought for values that can be obtained through more than one path.

5.9 Comparison of the Systems

In the following we present three tables that summarize the overview of multidatabases chapter.

Table 5.1 characterizes the types of systems and their support for integrated systems. In table 5.1 we characterize as complete systems, systems that, in addition to providing an integration framework, support network communication and various operating system facilities. HKBMS differs in that it does not consider the integration of heterogeneous database systems, but the integration of various heterogeneous expert systems with a database system.

Table 5.2 summarizes the common data model sections from the systems we gave an overview to. We notice that each system defines its own data model, however there are similarities between them since they follow the basic object-oriented concepts.

System	Type	Integrated Systems
Pegasus	Complete data management system	Information systems of various data models
VODAK	Complete multidatabase system	Heterogeneous database systems
SISIP	Integration framework	Various heterogeneous systems
OMS	Framework	Engineering information systems
DOMS	Complete system	In addition to database systems, hyper-media, application programs, etc.
Carnot	Complete system	Knowledge-based systems, and process models
CIS/OIS	Integration tool	File systems, databanks, information retrieval systems, etc.
FBASE	Integration framework	Database systems
Interbase*	Complete system	Database systems and UNIX utilities
A la carte	Framework for the integration of DBMS	Heterogeneous database management systems
HKDBMS	System that integrates expert systems with a database system	Many heterogeneous expert systems with one database management system

Table 5.1: Heterogeneous systems

Finally, table 5.3 compares the various integration techniques used. The *importation* entry refers to ways of defining a virtual global class that corresponds directly to a class in a component database, whereas the *derivation* entry refers to ways of defining a virtual global class that combines information stored in more than one class in the component databases. We discuss this table in section 5.9.2.

5.9.1 System Architecture

The criteria for categorizing an architecture as object-oriented is that the resources are modeled as objects, and all provided services are modeled as object methods. Object managers handle objects and the communication between them. OIS(CIS), DOMS, VODAK and A la carte support such an object-based architecture. A la carte offers object managers only for transaction management related services (see table 5.1).

5.9.2 Integration and Translation

The overview shows that there are numerous ways to approach the problem of integration (see table 5.3).

Our "streamline" description of a multidatabase system in chapter 2 is not necessary the skeleton followed. However, from the systems described, OIS, CIS, FBASE and Inter-

Base* are *tightly coupled* based on that they support the creation of a global schema. All the other systems described are *loosely coupled* because they do not support the integration of the schemas of their component databases. A la carte differs from the others in that ii focuses on integrating transaction management services, and therefore does not discuss schema integration. All the tightly coupled systems use the view definition of their query languages. The approach of VODAK is to resolve structural conflicts by transposing the conflicts to corresponding graph operations for different categories.

The approach to how virtual classes share the functionality of their base classes is often using the object-oriented inheritance concept. However OMS and DOMS introduce the means of delegation for information sharing. It is difficult to analyze which approach that would be most useful, but we choose to not investigate this any further – rather be aware of the diversity of approaches.

5.10 Summary

We have given an overview of some existing multidatabase systems and related systems. As we have experienced there are several different approaches to implement multidatabase systems and closely related systems.

These systems all have a basis in object-orientation, but as we have seen, they all define their own frameworks with specially developed object-oriented data models for their purpose. In this thesis we would like to avoid developing a system from scratch, but rather use some existing system and expand it if necessary to meet our requirements. A natural direction to go would therefore be to choose an existing standard as our data model. The ODMG-93[2] database standard [Cat94] is such a standard. The ODMG group [Cat94] is a group of vendors who have been working on a standard which they commit themselves to follow in the development of database products. It is considered as state-of-the-art in database research and therefore seems a natural choice as a starting point for our effort towards schema integration.

In the next part of the thesis we will see how we can use the ODMG-93 database standard [Cat94] and its ODL object model to support schema integration.

[2]ODMG = Object Database Management Group

System	Data model	DD/DM Language	Translation
Pegasus	Iris data model	HOSQL Extension of SQL	During importation Supports automatic translation of relational models
VODAK	VODAK data model	VML	During importation Uses metaclasses which implement interfaces to the modeling of the local systems
SISIP	SIOM	SIODL, SIOML, SIOQL	During importation Uses metaclasses as implementation managers which implement interfaces to the local systems
OMS	FUGUE model	Extension of a functional-based query language	Not discussed
DOMS	FROOM	Extension of a functional-based query language	Not discussed
Carnot	Instead of a CDM it uses a common-sense knowledge base called Cyc	GCL - Global Context Language Based on extended first-order logic	Special frames are defined for common information sources
CIS/OIS	Abstract data model(CIS) Integration data model(OIS)	QL Extension of a logic-based query language	Operational mapping
FBASE	Object-Oriented Defines a class hierarchy to model the integrated system	FSQL Extension of SQL	Performed by special FBASE servers
InterBase*	Object-Oriented	InterSQL Based on FSQL It also provides transaction specification facilities	Performed by special servers called CSIs
A la carte	Not applicable		
HKBMS	OSAM	Natural language-based	Not discussed

Table 5.2: Data model and translation

System	Integration		
	Importation	Derived Classes	Conflicts
Pegasus	By queries Virtual classes are called producer types and the query that defines them producer expression	By queries Virtual classes are called unifying types and functions are inherited from the base classes by unifying inheritance	Domain mismatch Naming & Schema Mismatch Object Identification
VODAK	Uses metaclasses to map the modeling constructs of local systems to the CDM	Uses the graphical representation of the local schemas to identify structural correspondences among them and then applies augmentation constructors. Then combines classes using the generalization constructor	Resolves structural conflicts by applying augmentation constructors Resolves data conflicts by defining appropriate methods
SISIP	The SIOM model introduces abstract attributes and relations. Together with the OQL they serve as a foundation for the support of integration. The separation between interface and implementation. Interface managers and implementation managers.		Not discussed
OMS	By queries and functions(constructors) Virtual classes are called derived classes An object algebra is defined with a set of functions that produce new sets of objects from existing ones.		Not discussed
DOMS	By queries and functions(constructors) An object algebra is defined with a set of functions that produce new sets of objects from existing ones.		Not discussed
Carnot	Uses articulation axioms to express mappings between two expressions that have equivalent meaning		Not discussed
CIS/OIS	Not supported		
FBASE	Not supported		
InterBase*	Not supported		
A la carte	Not applicable		
HKDBMS	Deals with the integration of rule-based (expert) systems In addition to global knowledge schema, a function graph is defined to desribe relationships between variables referenced in the rules of the component systems		Additional information is stored to determinate the best way to get a value Conflicts in values are resolved by the administrator

Table 5.3: Integration

Part II

Schema Integration in ODL-M

Chapter 6

ODL-M – A Mapping Language Extension to ODL

In the previous chapter we argued that none of the systems we investigated used a standard as part of their framework. We want to avoid developing a new model and therefore choose to extend an existing one. The ODMG-93[1] database standard [Cat94] is a state-of-the-art standard we will consider. As a starting point we look into the ODMG ODL [Cat94], object definition language, and will use it as our canonical data model. None of the existing systems we have investigated have used the ODMG ODL [Cat94] directly as their canonical model, so in this part of the thesis we will give a proposal to how we can use ODL as a canonical model in multidatabase systems and how we will support schema integration in this framework.

Using ODMG ODL as a canonical model means that we will perform our schema integration process on ODL schemas. Our idea is to extend ODL with some construct that allows us to define object types in the federated schema that act like "virtual" classes whose instances are mapped from the underlying systems (see figure 6.1). The figure shows the mapping from the source schemas to the target schema and describes which underlying classes serve as source data for the target schema object type instances. So the data flow can be thought to propagate along the defined mappings to populate the target schema object types. From this intension we need to define a mapping extension to ODL, since ODL doesn't support this inherently.

6.1 Introduction

In this chapter we will develop and define an extension to the ODMGs ODL [Cat94], namely ODL-M (Object Definition Language - Mapping). As the name suggests, ODL-M is an extension supporting various mapping constructs, being supportive to e.g. schema integration work. The idea is taken from the EXPRESS-M mapping language [Bai95], an extension to EXPRESS. The intended use of EXPRESS-M is to map entities from

[1] Object Database Management Group

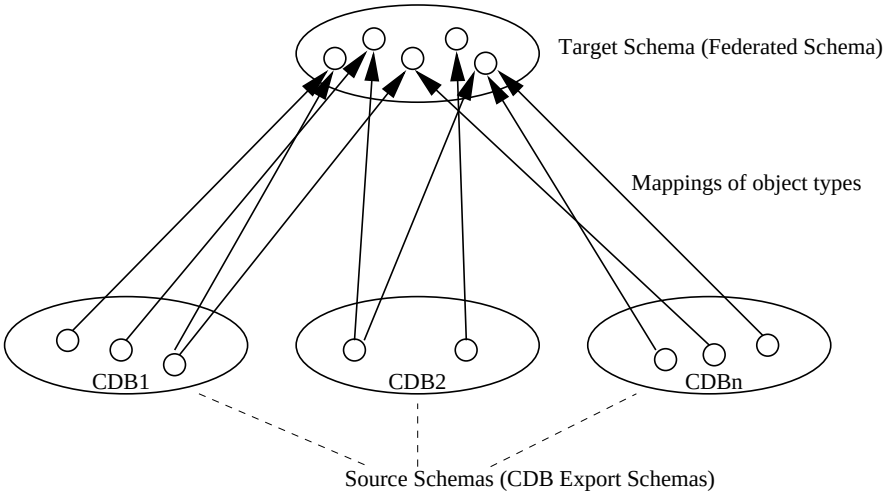

Figure 6.1: The idea of mapping ODL schemas.

one EXPRESS schema to any number of other schemas (i.e. source and target schemas). ODL-M is not as extensive as EXPRESS-M, it will only borrow the constructs that are needed for our purpose and add the constructs which are needed in addition to what EXPRESS-M offers. The above mentioned purpose of ODL-M is to extend the object-oriented data model ODL with the notion of *views*. We will use ODL/ODL-M as a support for schema integration within the ODMG-93 database standard [Cat94].

ODL-M will be used in chapter 7 as a mapping tool to resolve our problem classification from chapter 4.

Since the ODMG-93 database standard is a basis for our contribution it is in place to give a brief overview of this standard in the following section.

6.2 The Object Database Standard – ODMG-93

This section briefly describes the ODMG-93 standard [Cat94]. For a more in-depth overview, we refer to appendix B.

6.2.1 Introduction

The ODMG is a group of vendors who got together and decided to standardize their efforts of developing an object database instead of going in their own directions, developing non-interoperable products. The result of their work is the ODMG Object Database Standard which is an ongoing process. The participating vendors have committed themselves to follow this standard in their products. The goal of the project has been to allow an ODBMS user to write portable applications, i.e. applications that could run on more than one standard compliant ODBMS product. The hope for the member companies is that this proposal will become a de facto standard for the industry.

6.2.2 Object Model

The common data model has used the OMG Object Model [Sol90] as a basis. Components have been added to support the intended needs of the ODMG group.

The Object Model is simply summarized as:

- The basic modeling primitive is the *object*.
- Objects that exhibit common behavior and have a common range of states are categorized into *types*.
- The behavior of objects is defined by a set of *operations* or *messages* that can be executed on an object of the type[2].
- An object has a set of properties that can be either *attributes* of the object itself or *relationships* between the object and one or more objects. The state of an object is defined by the value it has for its properties.

6.2.3 Object Definition Language

The Object Definition Language (ODL) is a specification language to define the interfaces to object types that conform to the ODMG Object Model. The ODMG group has had a primary objective with the ODL to facilitate portability of database schemas across conforming ODBMSs. ODL is not intended to be a full programming language nor is it meant to be programming-language dependent. It is a specification language for interface signatures. It defines the characteristics for types, including their properties and operations, but is does not address the definition of the methods that implement those operations. Further, ODL provides a context for integrating schemas from multiple sources and applications. These source schemas may have been defined with any number of object models and data definition languages, and they may all be translated to ODL as a common basis (see fig.6.2). This common model allows the various models to be integrated with common semantics. An ODL specification can be realized concretely in an object programming language like C++ or Smalltalk (see section 6.2.5 and fig.6.2).

6.2.4 Object Query Language

The object query language (OQL) for the ODMG data model will be described briefly in the following. The ODMG group designed the OQL with the following principles and assumptions:

- OQL is not computationally complete. It is a query language which provides easy access to an object database.
- OQL provides declarative access to objects.
- OQL relies on the ODMG object model.

[2]E.g. you can "draw" an object of type Circle

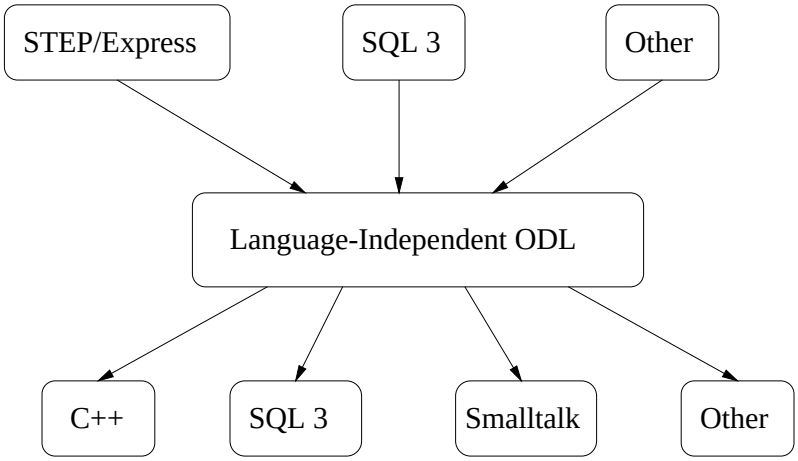

Figure 6.2: Mapping from other models to ODL, and from ODL to other languages

- OQL has an abstract syntax.
- The formal semantics of OQL can easily be defined.
- OQL has one concrete syntax which is SQL-like, but it is easy to change the concrete syntax. Other concrete syntaxes are defined for merging the query language into programming languages (e.g. a syntax for preprocessed C++ and a syntax for Smalltalk)
- OQL provides high-level primitives to deal with sets of objects but does not restrict its attention to this collection construct. Thus, it also provides primitives to deal with structures and lists, and treats all such constructs with the same efficiency.
- OQL does not provide explicit update operators but relies on operations defined on objects for that purpose.
- OQL can be easily optimized by virtue of its declarative nature.

OQL can be a stand alone language or it can be embedded into a programming language. The query language supports both types of objects, mutable and literals, depending on the way these objects are constructed or selected.

6.2.5 Language Bindings

The standard describes language bindings for both C++ and Smalltalk. The programming language-specific bindings for ODL/OML for C++ and Smalltalk are based on one basic principle: The programmer should feel that there is one language, not two separate languages with arbitrary boundaries between them.

6.2.6 ODL/ODL-M as a Canonical Model

As we recall the canonical data model (CDM) is the model that the local schemas are translated into. The advantages of using an *object-oriented* CDM has been discussed, among others, by Pitoura et. al [PBE95]. They summarize their discussion of advantages with an object-oriented CDM as the following enumeration:

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

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

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

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

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

As we discussed in section 2.5 the canonical model should have the properties of *expressiveness*, *semantic relativism*, and *support for views*. Saltor et. al [SCG91] argue that object-oriented models are among the best suited models to serve as a canonical model in that they meet the two first required properties to a better degree than other models. The only lack of essential properties, according to Saltor et. al [SCG91], is that they don't support views as a rule. This is a point Pitoura et. al [PBE95] also discuss. They define an *object oriented view* as a way of defining a *virtual database* on top of one or more existing databases by in general defining a set of *virtual classes* that are populated by existing objects or by *imaginary objects* constructed from existing objects.

ODL is a specification language for the ODMG Object Model. The ODMG Object Model's definition includes the basic concepts of an object-oriented model and thereby has the advantages discussed by Pitoura et. al [PBE95] and it satisfies the desired properties according to Saltor et. al [SCG91] with the exception of the support of views. The ODL-M extension covers this last requirement by supporting mapping of attributes and methods of the interface in ODL, creating a "virtual" database, thereby also supporting views. We therefore argue that ODL/ODL-M is a suitable specification language for an object model that meets the requirements specified by Saltor et. al [SCG91] and our additional requirement of supporting views.

Another important reason for choosing ODL as a basis for the canonical model is that it is part of the ODMG-93 Object Database Standard. The vendors participating in the ODMG-group are committed to support this standard and therefore ODL will hopefully be a standard we can rely on to be with us for some time.

6.3 Motivation – ODL-M

The current approaches to integrating schemas involve schema examination and ad hoc methods for translation and mapping of data between schemas. ODL-M provides for a more generic method of mapping schemas and facilitates automatic production of mapping from simple mapping instructions. This way the multidatabase system designer has a more unified method of resolving the mapping and translation tasks. Other systems we have investigated, e.g. Pegasus and VODAK discussed in chapter 5, defined views with the query language they supported. However, we have not seen any system that describes an approach to views in ODMG/ODL. This proposal will therefore fill the gap and define an extension to ODL that will support this.

6.4 What is ODL-M?

ODL-M is a schema mapping language for ODL schemas. It describes how object type instances in ODL are to be mapped between schemas in order to facilitate data transfer between the models described by the mapping schemas and the mapped schemas. An ODL-M mapping description uses mapping commands to specify which characteristics are to be mapped from which schemas (source), which schema to map to (target) and how the mapping should be done. Thus it maps each construct of the ODL interface characteristics.

Fundamental principles:

- The ODL-M language is a means by which ODL object types can be mapped from one schema to another.
- ODL-M does *not* describe *how* to read/write individual types to/from applications, it references ODL schemas to resolve definitions of object types.

6.5 ODL-M Compiler

Here we give a short description of how an ODL-M compiler can be used.

The first stage is the generation of C++ [Str91] code from the mapping file. The mapping file and the source and target files are run through their respective compilers as figure 6.3 shows. The result mapping code is in C++[3]. Work has been done on a ODL-to-C++ compiler [LS92]. The EXPRESS-M language has a compiler developed and

[3]The output language may be other languages, e.g. Smalltalk. C++ and Smalltalk are the two languages which have a described mapping from ODL in the ODMG-93 standard

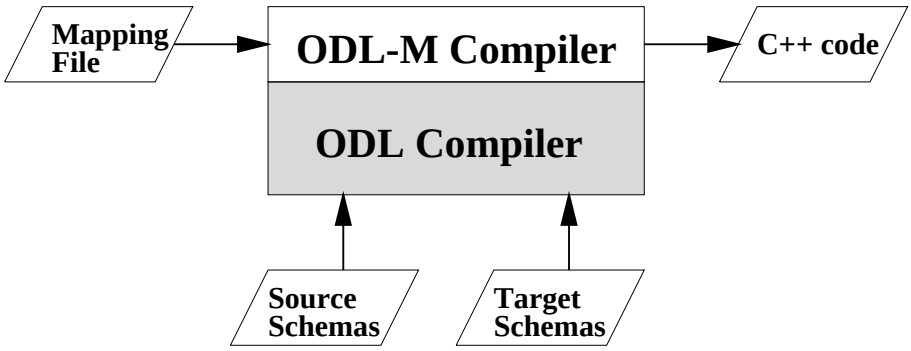

Figure 6.3: ODL-M Compiler

it should be a feasible task to develop a compiler for the ODL-M language since it is close to EXPRESS-M.

Once the mapping file is translated to C++ it may be integrated with the rest of the application to implement the mappings between schemas.

6.6 Declarations

An overview of the mapping constructs in ODL-M is given in the following. Since ODL-M is intended to map object type interfaces, we must define constructs that allow mapping of each characteristic in the ODL interface definition (see figure B.5 in appendix B). The syntax of the constructs defined here is described in appendix C.

6.6.1 Type declarations

Here we define some limitations on type declarations:

- ODL-M includes all the data types that are available in ODL, both simple (integer, boolean) and aggregate (array, bag).
- Named data types may *not* be declared in ODL-M. Types which have been declared in the source and target schemas may be mapped.
- Constructed types may not be declared within the scope of an ODL-M map, but may be referenced in a map.

6.6.2 Schema Map

ODL-M mappings require that the names of the source and target schemas be declared. The `SCHEMA_MAP` declaration is used to specify which source and target schemas may be mapped. The syntax of the declaration is:

```
SCHEMA_MAP target_schema <- source_schema1, source_schema2;
<body>
END_SCHEMA_MAP;
```

There is no limit on the number of source schemas, but there may be only one target schema. The body of the map declaration is contained between the schema map header, `SCHEMA_MAP schema <- schema`, and the `END_SCHEMA_MAP` statement. All the object type map and instantiation commands that make out the mapping will be contained within the schema map body. In the following sections of this chapter we will omit the `SCHEMA_MAP` statement because it is not important for the understanding of the examples and would take up unnecessary space. We will, however, use it in chapter 7 in our resolving techniques.

6.6.3 Object Type Map

The `MAP` declaration in ODL-M defines which object types are to be mapped from the source schemas and which are to be mapped from the target schema and how their attributes and operations are to be mapped. It is within the object type maps the specifics of the mappings are described. We will describe the attribute maps and operation maps in following sections, but first the general object type map will be described.

A `MAP` declaration consists of a series of attribute maps and statements. Local variables may be declared in a `MAP` declaration. Assignment statements may also be declared in a map declaration, but only if they are to be used to assign values to local variables. The syntax of the `MAP` statement is as follows:

```
MAP target_object_type <- source_object_type1, source_object_type2;
<body>
END_MAP
```

The body of the map is contained between the map header,
`MAP target_object_type <- source_object_types`, and the `END_MAP` command. The body is made up of a series of attribute and operation mapping commands, which specify how equivalent attributes and corresponding operations are to be converted and mapped from source to target.

Example: This is a simple example of object type mapping. The source schema describes an inheritance hierarchy of a `man` and a `woman` being subtypes of `human`. The target schema defines the generic type `human_beeing`.

Source schema:

```
typedef float inches;

interface Human{
   Integer age;
   Inches height;
   };

interface Man:Human{};

interface Woman:Human{};
```

Target schema:

```
typedef float centimeters;
typedef enum{male, female} male_or_female;

interface human_beeing{
   male_or_female sex;
   Integer how_old;
   centimeters how_tall;
   }
```

The mapping between these schemas would be:

```
SCHEMA_MAP to_schema <- from_schema

MAP human_beeing <- man;
   sex :- 'male';
   how_old  :- age;
   how_tall :- height*2.54;
END_MAP;

MAP human_beeing <- female;
   sex :- 'female';
   how_old  :- age;
   how_tall :- height*2.54;
END_MAP

END_SCHEMA_MAP;
```

The example above shows a simple map which assigns attribute values from the object types of the source schema to those in the target schema. The attributes are mapped as described in the following section. The example shows direct assigning of values to the attribute `sex`, which is forced to take the value of '`male`' or '`female`'. Simple transfer of attribute values is shown with the `age` attribute being mapped, and the use of mathematical operators is shown in the `height` attribute map. We will come back to these mapping types in the following sections.

6.6.3.1 Attribute Maps

In this section we describe how the attributes of the ODL interface may be mapped.

An attribute map assigns values to attributes defined in one of the target properties of the map. These values may be derived from the values of attributes which are defined in the source properties.

There is no requirement to make a *total* mapping, i.e. to map all the properties of the source properties.

An attribute map will be a statement consisting of a left hand operand and a right hand operand separated by an assignment operator ':-'. The left hand operator must be a qualified attribute defined in the target object of the `MAP`. The right hand operand must be either a simple expression, or a qualified attribute defined in one of the source object types of the map.

Attributes inherited from supertypes may be mapped the same way. That is, attributes inherently belonging to a subtype (inherited from its supertype) may be mapped the same way as its other attributes.

Relationships will be interpreted and mapped in the same manner as attributes. The `inverse` specification is up to the designer to maintain by taking care of the inverse relationship in the corresponding object type. This can however become automated in ODL-M by defining a special syntax for it in a later definition of ODL-M.

Attribute map example:

```
1.)  attribute1 :- attribute_a;

2.)  attribute1.attribute2 :- attribute_a;

3.)  objtyp1.attribute1 :- attribute_a;

4.)  attribute1 :- attribute_a * 4;
```

1. maps `attribute_a` to `attribute_1`
2. maps `attribute_a` to `attribute_2` of `attribute_1`
3. shows `attribute_1` being qualified by an object type name
4. shows a simple mathematical operation being carried out on the mapping.

6.6.3.2 Operation Mapping

Here we describe how operations in ODL interfaces may be mapped.

An operation map will map an operation defined in the source object type to an operation defined in the target object type. The parameters of the operation will be mapped specifically to the corresponding target parameters. Source and target operations can only be mapped if their parameters can be matched, i.e. each attribute in the source operation can be mapped to a corresponding parameter in the target operation, possibly with a cast (see section 6.6.8). However, mapping can also be allowed to an expression that expresses the nature of the underlying operation. This way, missing operations in CDBs can be encountered for by including a map to an expression on some of the CDBs other data.

The operation map is built up by an operation name map header followed by the list of parameter mappings. If the parameter lists has an ordered one-to-one correspondence

matching, the parameter list map may be omitted as ODL-M automatizes the mapping process in that case. The map header consists of a left and right hand operation name, from the target and source object types respectively, separated by the '= symbol. A `WHERE` clause follows that qualifies each parameter mapping. If an expression is provided instead of the operation-to-operation map, it will simply have the syntax: `op_name = expression`, where the expression usually is based on some other attributes in the source schema.

Example:

Source object type:

```
interface Cube{
   Integer x_axis;
   Integer y_axis;
   Integer z_axis;
   Integer size(in x_length:Integer, in y_length:Integer, in z_length:Integer);
   };
```

Target object type:

```
interface Square_Block{
   Integer xcoord;
   Integer ycoord;
   Integer zcoord;
   Integer volume(in z:Integer, in y:Integer, in x:Integer);
   }
```

The mapping could be:

```
MAP Square_Block <- Cube;
   xcoord :- x_axis;
   ycoord :- y_axis;
   zcoord :- z_axis;
   volume() = size()
   WHERE (xcoord :- z,
          ycoord :- y,
          zcoord :- x);
END_MAP;
```

In the above example the parameters of `size` and `volume` were in different order and therefore had to be mapped specifically to rearrange them so they matched up with the target parameters. If the operation definitions instead had read:

```
Integer size(in x_length:Integer, in y_length:Integer, in z_length:Integer);
```

for the source, and

```
Float volume(in x:Integer, in y:Integer, in z:Integer);
```

then the mapping declaration could simply have been:

```
volume() = (Float) size();
```

indicating that the parameter lists had a direct corresponding structure.

6.6.3.3 Instantiating Multiple Properties

In some mapping cases the object type being mapped from expresses several instantiations of the target object type. This section describes how we handle these cases.

The object types which are to be instantiated by the map are specified in the map header. More than one object of the same type may be instantiated by using an index qualifier `[]` as a suffix to the object type name. An index qualifier may specify an indeterminate number of objects `[?]`. Several objects of different types may be instantiated by listing the object type names, separated by commas.

Example: Mapping to several objects from one source object.

```
MAP line[4] <- polyline;

MAP line[?] <- polyline;

MAP wheels, shoe <- rollerskate;
```

6.6.4 Creating Target Objects from Multiple Source Objects – Build

Sometimes we do not wish to map all instances from a merge of source schemas, but rather restrict the mapping to certain conditions. This section describes how we can manage this.

Mapping from multiple object types is possible in ODL-M. However, when mapping from more than one object type, the instances of those source object types have to be accounted for. Some value criteria has to be given for the map, otherwise every possible combination of instances of the source object types will be mapped.

A `BUILD` statement is used to construct object type instances in the target model from unrelated instances of different types in the source model. The conditions under which the target types will be created are given in the `WHERE` rule in the body of the `BUILD` command, and all combinations of source instances which satisfy the rule will create a target instance.

The `BUILD` declaration states the source object types which are to be mapped, and the target object type to be created from them. The body of the declaration may contain attribute maps, and flow control statements in the same way as a map declaration body.

Example: Create a target type `Couple` from the source types `man` and `woman` using the build command.

Source ODL:

```
Interface Man{
    string   name;
    Integer  masculinity;
};
```

```
Interface Woman{
    string   name;
    Integer  femininity;
};
```

Target ODL:

```
Interface Couple{
    string   husband_name;
    string   wife_name;
};
```

The ODL-M would be:

```
BUILD Couple <- Man, Woman;
   WHERE
      ABS(man.masculinity - woman.femininity) <= 2;

   husband_name :- man.name;
   wife_name    :- woman.name;
END_BUILD
```

The condition to build a couple from a `Man` and a `Woman` is the (absolute) difference in masculinity and femininity[4]. Thus for every `Man` and `Woman` instance that satisfies the condition, a `Couple` instance will be built in the target application. This can be seen as a way of creating *possible* couples, as a man or woman may be in more than one couple.

The `WHERE` rule used in the build command must be specified very precisely to avoid unwanted instances being created in the target model.

6.6.5 Copy

To simplify straightforward mappings we define the `COPY` command described in the following.

The `COPY` command may be used to map classes without mapping the attributes. This may only be used when the source and target object type have the same attribute names and types, i.e. the object types are identical with respect to their attributes.

[4]The femininity/masculinity condition is just an example of possible conditions. An age condition could just as well be used.

Example:

Source ODL:

```
Interface Worker{
   Integer age;
   string name;
};
```

Target ODL:

```
Interface Employee{
   Integer age;
   string name;
};
```

The ODL-M would be:

```
COPY Employee <- Worker;
```

6.6.6 Discarded Data

If a class cannot be mapped to any structure listed in the target application then that class may be discarded. We can use a `NO_MAP <object_type>` to achieve this.

6.6.7 Type Mapping

In order to provide a high level method of data exchange, one may use type mapping. Named types other than object types may be mapped using a `MAP_TYPE` declaration. Type mapping takes two forms; defined type mapping and enumeration type mapping.

6.6.7.1 Mapping of Defined Data Types

Defined type mapping is used to declare which elements of a defined type map to their equivalent elements in a target defined type. It may also be used in a trivial case mapping where defined types are renamed simple types.

Here is an example of trivial type mapping[5] where the map header shows the simple relationship between the types:

Source type:

```
typedef float inches;
```

Target type:

```
typedef float centimeters;
```

The type mapping would be:

```
MAP_TYPE centimeters = inches * 2.54;
END_MAP_TYPE;
```

The above example is very simple, the type mapping will be called when casting (see section 6.6.8) an attribute of type inches to an attribute of type centimeters in an attribute map.

[5] A similar mapping was made in section 6.6.3, but there the attributes were mapped explicitly whereas here the *type* is being mapped.

The more complex case of type mapping is when the types involved are not simple types, but compound types, such as arrays or bags. The array is an ordered collection and its members can be mapped according to their index position. Bags however are unordered so the the entire contents is mapped provided the sizes of the types involved in the map are the same.

6.6.7.2 Mapping of Enumeration Types

Enumeration type mapping is used for declaring equivalences between elements of source and target enumeration types. The map header will be the same as for defined type mapping, but the individual corresponding components of the enumeration will also be mapped. The ':-' operator is used to indicate which elements correspond to each other in the map. The left hand side of the operator will be the target enumeration identifier and the right hand side will be a list of one or more source enumeration identifiers, separated by commas.

Example: Enumeration of colors

Source enumeration type:

```
typedef enum{red, green, blue, burgundy, transparent, aquamarine} hues;
```

Target enumeration type:

```
typedef enum{red, green, blue} colors;
```

A possible mapping could be:

```
MAP_TYPE colors = hues;
   red :- red, burgundy;
   blue :- blue;
   green :- green, aquamarine;
END_MAP_TYPE
```

In the above example the `transparent` enumeration element has no equivalent to map to, and so is not mapped at all. Also, the example illustrates how more than one source element may be mapped to a single target element.

6.6.7.3 Using Type Mapping

Type mapping should generally be used in cases where more than one object type map uses a specific type so that there is no repetition of verbose attribute mapping. It also has another function; that of semantic enrichment to the model, improving the understandability. This is because adding named types clearly adds information to the schemas much like qualified attributes do (e.g. `weight_in_kilograms`). In some cases, however, it is simpler to carry out the task with just an attribute map.

6.6.8 Casting

Casting is used to convert data types between source and target attributes. It can be used to convert between simple types and in addition it can be used to call object type maps and type maps to carry out conversion of non-simple attribute types.

Example: Simple type mapping

Source object type:

```
interface product{
   Integer serial_number;
   };
```

Target object type:

```
interface device{
   String identification_code;
   };
```

The mapping could be:

```
MAP device<- product;
   identification_code :- (String) serial_number;
END_MAP
```

Another use of casting is when using it with attributes that are object types. In these cases the casting specifies that the object type should be converted to a specific type, and thereby calls the appropriate object type map. The following example shows casting with object types and defined types.

Example:

Source schema:

```
typedef Float Inches;

interface Car{
   String color;
   String make;
   String model;
   Integer age;
   };

interface Person{
   Integer age;
   Inches height;
   Car vehicle;
   };
```

Target schema:

```
typedef Float Centimeters;

interface Automobile{
   String manufacturer;
   String model;
   String paint;
   };

interface Human{
   Float age;
   Centimeters height;
   Automobile transport;
   };
```

The type mapping in this case could be:

```
MAP_TYPE Centimeters = Inches*2.54;
END_MAP_TYPE
```

And the object type mapping can be:

```
MAP Automobile <- Car;
   manufacturer :- make;
   model :- model;
   paint :- color;
END_MAP;

MAP Human <- Person;
   height :- (Centimeters) height;
   age :- age;
   transport :- (Automobile) vehicle;
END_MAP;
```

In the above example the cast `(Centimeters)` calls the appropriate type map to carry out the conversion from `Inches` to `Centimeters`. The `(Automobile)` cast calls the object type map from `Car` to `Automobile`).

6.7 Instance Control

We need to control how the target object types are instantiated. Sometimes our defined mapping constructs have undesired effects or may not have the functionality to instantiate the way we want it to. This section describes how we can fine-tune the instantiation.

Instantiation of target classes can be controlled in four ways:

1. *Mapping precedence* by order.
2. *Default instantiation* of objects when they are mapped.
3. *Manual creation* of specific instances which are to be referenced.
4. *Pruning of classes* to prevent a source instance being mapped more than once.

We describe them in the following.

6.7.1 Mapping Precedence

Mapping precedence is defined by the order of the mapping statements. When a source instance is mapped by two separate map statements, the target instance will be created by the statement which is first in the `SCHEMA_MAP`. An important feature is that the key attributes of the target objects act like pruning flags automatically. If an instance of a target object type is attempted mapped more than once with the same key attributes, the latter attempts will not create duplicate instances, but may add to missing attribute values in the target if the alternative source objects can provide it. Mapping precedence will in these cases ignore following clashes in attribute values or possibly notify the designer.

6.7.2 Default instantiation

ODL-M distinguishes between target type instances that are instantiated as a result of type maps and those that are the result of attribute maps.

Example: Mapping `Point`s.

Source ODL:

```
interface Line{
   Point start;
   Point end;
   }

interface Point{
   Float x, y, z;
   }
```

Target ODL:

```
interface Line_Vector{
   Point begin;
   Point terminate;
   }

interface Cartesian_Point{
   Array<Float> vector;
   }
```

The mapping can be done in two different ways. The first is to cast to map the points.:

```
MAP Line_Vector <- line;
   begin     :- {Cartesian_Point} start;
   terminate :- {Cartesian_Point} end;
END_MAP

MAP Cartesian_Point <- Point;
   vector[0] :- x;
   vector[1] :- y;
   vector[2] :- z;
END_MAP;
```

In the above case, the points will only be mapped once – in the point map (Note: the `[n]` after vector is not a multiple instance declaration as we described earlier, but simply the elements of a vector).

It is possible to map points without a cast. Here is an example of this:

```
MAP Line_Vector <- Line;
   begin.vector[0] :- start.x;
   begin.vector[1] :- start.y;
   begin.vector[2] :- start.z;
   terminate.vector[0] :- end.x;
   terminate.vector[1] :- end.y;
   terminate.vector[2] :- end.z;

MAP Cartesian_Point <- Point;
   vector[0] :- x;
   vector[1] :- y;
   vector[2] :- z;
END_MAP;
```

In this second case, the points will be instantiated twice – once by the `Line` map and once by the `Point` map. This may be undesirable and we will see in the section "Type Instance Pruning" below how we can avoid this.

6.7.3 Manual Creation

One may create specific instances of target object types by using *instantiation clauses*. The instantiation clause is used to explicitly create object types. The form of the clause is as follows:

```
#instance_id = objtype_id (parameter_1, parameter_2 ... parameter_n);
```

The parameters are of the following types:

object type instance_id	– preceded by the `#` symbol
numerical value	– corresponding to `integer`, `float` types
binary value	– hexadecimal e.g. `F45ED20`
boolean value	– `.TRUE.` `.FALSE.`
string	– contained in quotes e.g. `'hello'`
aggregates	– contained in parentheses e.g. `(1,2,3,4,5)`
enumeration element	– e.g. `.enum_id`
null element	– `$` symbol used where optional attributes are not assigned.

Example:

```
interface Thing{
   Float x;
   }
```

```
interface Widget{
   Float a;
   logical b;
   binary c;
   String d;
   Array<Integer> e;
   Optional Integer f;
   Thing g;
   }
```

Two instance clauses could be:

```
#objtype_1 = Thing(6);
#objtype_2 = Widget(4.56, .UNKNOWN., F64E, 'Hello', (1,2,3,4,5), $, objtype_1);
```

6.7.4 Type Instance Pruning

As mentioned above our mappings may have undesired effects. In some cases we might map more than one object at the result of a map from the same source instance. This section describes a pruning definition we can use to avoid this.

Object type instances may be subject to a pruning algorithm. Pruning clauses are used to prevent more than one object being instantiated at the result of a map from the same source instance.

A prune statement may be declared within the scope of a map statement. A prune clause has a prune identifier or list of identifiers. The clause contains a list of target attributes and object types which the pruning process acts on. The attribute identifiers contained in a prune clause reference attributes of the target object types described in the map header. The object type identifiers is a subset of the target object types listed in the header.

A prune clause may contain more than one pruning identifier. The identifier may be combined using the logical operators `AND`, `OR` and `XOR` to enable control of object type instancing under different circumstances.

Example:

```
PRUNE prune_id1, prune_id2;
   <target attributes and/or object types>
END_PRUNE;
```

`PRUNE prune_id1, prune_id2;` means that the following object types will be pruned if they have been mapped from the same source object type by another map that is subject to `prune_id1` *or* `prune_id2`.

```
PRUNE prune_id1 AND prune_id2;
   <target attributes and/or object types>
END_PRUNE;
```

`PRUNE prune_id1 AND prune_id2;` means that the following object types will be pruned if they have been mapped from the same source object type by two maps which are subject to `prune_id1` *and* `prune_id2`

Example: When object types are mapped using attribute maps they may be instantiated in the target more than once. To avoid this pruning may be used as follows:

Source ODL:

```
interface Line{
   Float start_x, start_y;
   Float end_x, end_y;
   }

interface Point{
   Float x,y;
   }
```

Target ODL:

```
interface Line{
   Point start;
   Point end;
   }

interface Point{
   Float x_coord, y_coord;
   }
```

One approach to mapping is:

```
MAP Line <- Line;
   start.x_coord :- start_x;
   start.y_coord :- start_y;
   end.x_coord :- end_x;
   end.y_coord :- end_y;
END_MAP
```

```
MAP Point <- Point;
   x_coord :- x;
   y_coord :- y;
END_MAP
```

In this case, there would be duplicate instantiations of the point object types in the target model. Pruning may be used to prevent point instances of the same value being created:

```
MAP Line <- Line
PRUNE Line_Points_pruning;
   start, end;
END_PRUNE;

   start.x_coord :- start_x;
   start.y_coord :- start_y;
   end.x_coord :- end_x;
   end.y_coord :- end_y;
END_MAP
```

```
MAP Point <- Point
PRUNE Line_Points_pruning;
   Point;
END_PRUNE;

   x_coord :- x;
   y_coord :- y;
END_MAP
```

In the above case, both maps are subject to the same pruning identifier, namely `Line_Points_pruning` so any points created by either map will be value compared, and only one instance of that value will be allowed to exist in the target model.

6.8 Summary

We have defined a mapping language, ODL-M, that can be used to map interfaces from one target object type to multiple source object types. Not only is it able to map the properties of the ODL interface, but also can perform type mapping, a strong feature that introduces better semantics to the model and eases the understanding of mapping constructs. The instance control that ODL-M offers, secures that we don't achieve any unwanted instantiations and that we can have full control of our mapping intensions.

In the next chapter we will use ODL-M to realize suggested resolution techniques.

Chapter 7

Schema Integration with ODL-M

7.1 Introduction

In this chapter we will revisit the conflicts detected in chapter 4. A classification of resolution techniques will be presented that will show that they cover the conflicts we listed in the four tables 4.1, 4.2, 4.3, and 4.4, from section 4.3. With reference to the schema integration process steps of chapter 3, this chapter deals with the *conforming* and *merging* steps, but also the *preintegration* step to a degree. To perform the resolution techniques we will use the ODL-M mapping language that we developed in chapter 6. Finally we will discuss a proposal for how to use of the `semPro` function from chapter 4. But first we will give a brief suggestion to an architecture this system could run under. The architecture borrows some concepts from the systems mentioned in chapter 5.

7.2 An Architecture Basis

Here we will design a rough framework of how a full working system could be implemented.

According to the goals of the ODMG group, we would like to comply with how they intend for the database standard to participate in a distribution of heterogeneous databases. Our proposal is to use the OMG Object Request Broker [OMG92] as the network service to support the object passing between the local and the global systems. We borrow the idea of a distributed persistent object space from SISIP [BHR+95] as the uniform integration space. The component ODL schemas will be integrated in this integration space to the object types of the federated schema which in turn offers its interface to the external global users. The basic nature of the proposed framework is outlined in figure 7.1. We will not discuss the details in this architecture any further since our focus is on resolution techniques in this chapter. Besides, all the constructs in figure 7.1 have been discussed earlier in the thesis.

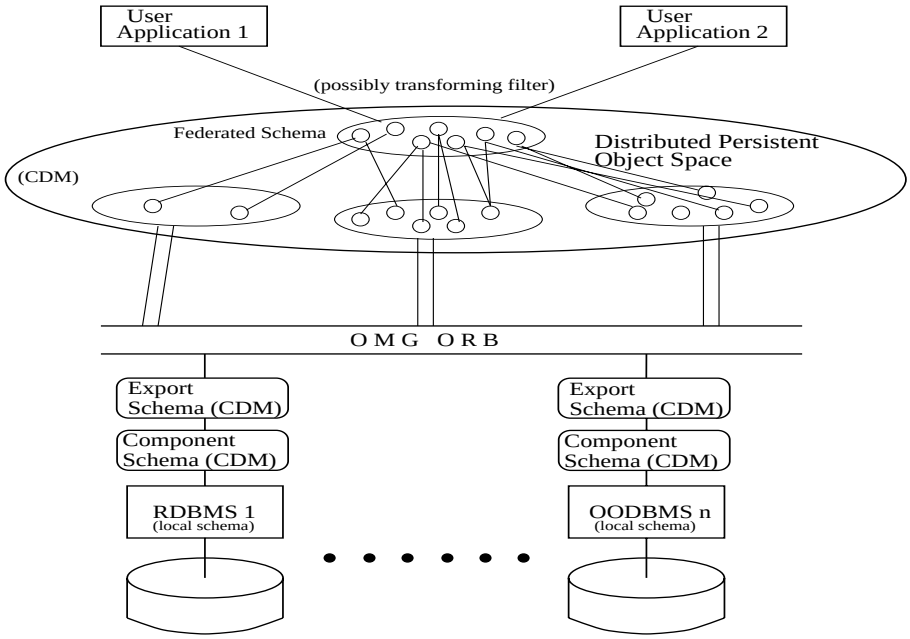

Figure 7.1: Proposed framework/architecture

7.3 Conflict Resolution in ODMG-93 using ODL-M

A classification of resolution techniques will be introduced in the next sections. The technique classification will cover the conflicts discussed in the classification tables of section 4.3. The techniques are partly inspired by Kim et. al [KCGS95] and partly independently designed as part of this thesis. Th main structure of the resolution techniques is taken from Kim et. al [KCGS95], but we have adopted the techniques to an object-oriented context and introduce the use of ODL-M as a means of performing our techniques. This approach has not been investigated before, as far as we know. In resolving the conflicts we will use ODL/ODL-M as defined in chapter 6, either directly through the properties of the data model of ODL or by means of the mapping rules available in the ODL-M language.

The mappings will typically be of the form:

```
MAP MDB_object_type <- SCHEMA1_object_type, SCHEMA2_object_type
<body>
END_MAP
```

where the CDB object types are the sources of the mapping and the MDB object type is the source.

7.3.1 Introduction

The case mentioned in the introduction chapter is revisited here (see fig. 7.2), this time

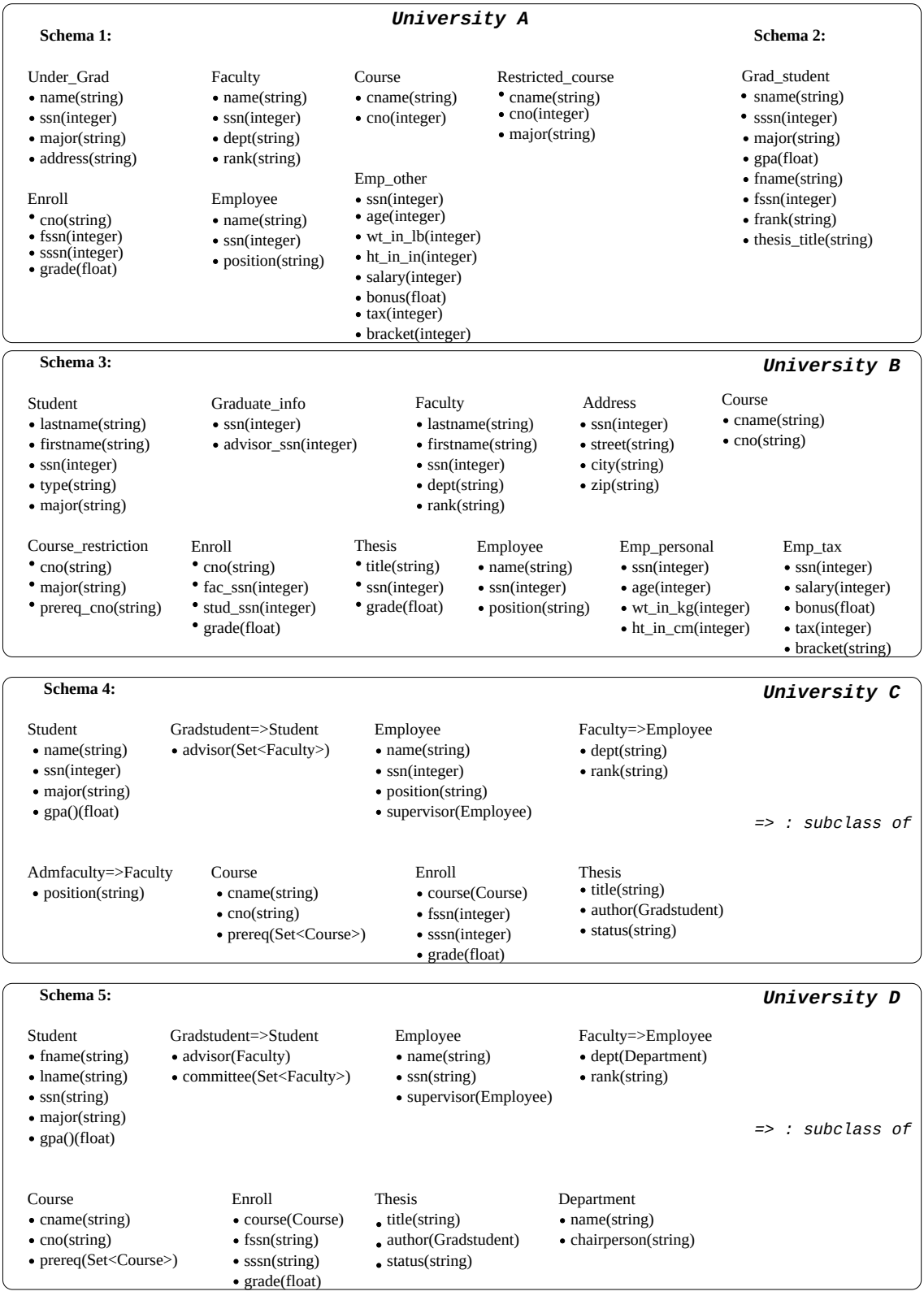

Figure 7.2: Revisited Case

with types according to the ODL data model definition. In this chapter we will go into more detail in the case and resolve conflicts between the classes being integrated.

In order to be able to use the ODL-M language, the schemas must be expressed in the ODL language. The translation of the case will be used in the examples of resolution techniques in the following sections. The ODL-representation of all the object types in the case is presented in appendix D.

7.4 Resolution Techniques

The classification used in this section is shown in fig. 7.3. The conflicts these resolution techniques are meant to address are listed in the four requirement tables in section 4.3. For each type of conflict, in each of the four requirement tables, the resolution technique can be modeled as a transformation from one or more classes/object types defined in the CDB schemas to a single class/object type defined in the MDB schema. Whenever this transformation is isomorphic, the global class is updatable [KCGS95]. We will only briefly mention when a resolution meets this criteria and not focus on it here.

We present a classification of resolution techniques and for each group in the classification we enter a section containing three parts:

Conflict: In this part we refer to which conflict in which table from section 4.3 we will address in order to refresh our memory and to keep track of which conflicts we have covered so far.

Resolution: In this part we suggest one or more resolution techniques to resolve this particular conflict type. Resolutions of different types of conflicts might resemble each other but we choose to separate them in order to address the conflict types apart from one another.

Example: In this part we generally extract an example from the case and use the resolution technique recently stated to demonstrate its use.

1. Renaming Classes and Attributes
2. Homogenizing Representations
   (a) Expressions
   (b) Units
   (c) Precision
3. Homogenizing Attributes
   (a) Type Coercion
   (b) Extraction of a Composition Hierarchy
   (c) Default Values
   (d) Attribute Concatenation
4. Horizontal Merges
   (a) Union Compatible
      i. Simple Union Compatible Merge
      ii. When Attribute is Missing
      iii. When Attribute is Missing but Value is Implicit
   (b) Extended Union Compatible
      i. For Class Inclusion
      ii. For Attribute Inclusion
5. Vertical Merges
   (a) For Many-to-Many Classes
   (b) For Class-vs-Attributes
   (c) For Aggregation Hierarchies
6. Mixed Merges
7. Homogenizing Methods

Figure 7.3: A Classification of ODL-M Resolution Techniques

7.5 Renaming Classes and Attributes

Conflict Conflicts of type "Class Name" and "Attribute Name" in table 4.1 and table 4.2 arise when concepts(classes or attributes) with similar meaning have different names (synonyms) or when different concepts bear the same name (homonyms) in the CDB schemas.

Resolution A catalog is maintained in the MDB that captures the correspondence between MDB names and CDB names. The entrances in the catalog can be maintained by either the designer or it could be semi-automatic maintained by a semantic measure that assigns similarity values (such as the `semPro` function from chapter 4) to encounter synonyms. The problem of equal names for different concepts can be avoided by prefixing the class names by their schema names.

Example A resolution from Pegasus (see section 5.1) on handling ambiguity is this: Objects with equal names can be prefixed with their respective schema names to define unambiguous names of the form: `schemaname.objectname`. Likewise the attributes can be prefixed with their respective schema names and object names of the form: `schemaname.objectname.attributename`.

Case example:
The object type `Under_Grad` in Schema 1 and `Student` in the remaining CDBs are similar concepts bearing different names. Similarly, the attributes `major` in `Gradstudent` and `dept` in `Faculty` have the same meaning but different names.

7.6 Homogenizing Representations

Here we discuss homogenization of different expressions denoting the same information, different units, and different levels of precision. They correspond to the class of conflicts identified as "Different Representation for Equivalent Data" in table 4.4.

7.6.1 Different Expressions Denoting the Same Information

Conflict Conflicts of type "Different Expression denoting same Information" from table 4.4 arise when different scalar values are used to represent the same data. Of particular interest are cases when different CDBs use separate codes to denote the same data.

Resolution Since this type of conflict arises when different scalar values denote the same data, it is resolved by defining an isomorphism between different representations. This can be achieved either by defining type mappings denoting the isomorphism or by direct mappings object type to object type.

Example Considering different representations of grades from the case as `enum` types we could have the following mapping between types:

Source enumeration type:

```
typedef enum{A, B, C, D, E, F} grade_alpha;
```

Target enumeration type:

```
typedef enum{1, 2, 3, 4, 5, 6} grade_digit;
```

A mapping between the two would be:

```
MAP_TYPE grade_alpha=grade_digit;
   A :- 1;
   B :- 2;
   C :- 3;
   D :- 4;
   E :- 5;
   F :- 6;
END_MAP_TYPE
```

We use the type mapping defined here in the following example:

Assume we change the `Enroll` object types slightly by using the different types for the `grade` attribute that we just defined. We use the `grade_digit` type for the `grade` attribute of `Schema 1` and the `grade_alpha` type for the `grade` of `Schema 4` and `Schema 5`. We could map the source `Enroll` object types integrating them into a target `All_Enroll` object type using the above defined types and type mapping as the following:

Source object types:

Schema 1:

```
interface Enroll{
   extent enrolls;
   keys cno,fssn,sssn;

   string cno;
   integer fssn;
   integer sssn;
   grade_digit grade;
   }
```

Schema 3:

```
interface Enroll{
   extent enrolls;
   keys cno,fac_ssn,stud_ssn;

   string cno;
   integer fac_ssn;
   integer stud_ssn;
   float grade;
   }
```

Schema 4 and 5:

```
interface Enroll{
   extent enrolls;
   keys Course,fssn,sssn;

   Course course
   integer fssn;
   integer sssn;
   grade_alpha grade;
   }
```

Target object type:

```
interface All_Enroll{
   extent all_enrolls;
   keys cno,fssn,sssn;

   string cno;
   integer fssn;
   integer sssn;
   grade_alpha grade;
   }
```

The mapping could be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3, SCHEMA4, SCHEMA5;

MAP All_Enroll <- SCHEMA1.Enroll;
   cno :- cno;
   fssn :- fssn;
   sssn :- sssn;
   grade :- (grade_alpha) grade;
END_MAP;

MAP All_Enroll <- SCHEMA3.Enroll;
   cno :- cno;
   fssn :- fac_ssn;
   sssn :- stud_ssn;
   IF 1 <= Enroll.grade <= 1.9 THEN
      grade :- 'A';
   ELSE_IF 2 <= Enroll.grade <= 2.9 THEN
      grade :- 'B';
   ELSE_IF 3 <= Enroll.grade <= 3.9 THEN
      grade :- 'C';
   ELSE_IF 4 <= Enroll.grade <= 4.9 THEN
      grade :- 'D';
   ELSE_IF 5 <= Enroll.grade <= 5.9 THEN
      grade :- 'E';
   ELSE
      grade :- 'F';
END_MAP;
```

```
MAP All_Enroll <- SCHEMA4.Enroll;
   cno :- course.cno
   fssn :- fssn;
   sssn :- sssn;
   grade :- grade;
END_MAP;

MAP All_Enroll <- SCHEMA5.Enroll;
   cno :- course.cno
   fssn :- fssn;
   sssn :- sssn;
   grade :- grade;
END_MAP;

END_SCHEMA_MAP;
```

In the mapping from `SCHEMA1` we cast the grade attribute thereby triggering the type mapping defined. In the mapping from `SCHEMA3` we had to divide the grade scale into ranges that fit the target grade attribute. This latter method is actually an example of different precision conflict and we will see how to resolve this conflict more efficiently in the "Different Levels of Precision" section.

The two last mappings from `SCHEMA4` and `SCHEMA5` were trivial.

7.6.2 Different Units

Conflict Conflicts of type "Different Units" from table 4.4 arise when numerical data denoting the same physical quantity are represented in different units across CDBs. Different units give different meanings to numerical data.

Resolution Since this is a conflict among numerical data, it is resolved by defining arithmetic expressions to convert numeric value in one unit to another. There are limitations to the accuracy of such conversions due to at least two reasons:

1. Not all arithmetic operators are closed on numeric values, e.g. division is not closed for integers.
2. There are limitations of machine representations for real values.

Example `Schema 1` and `Schema 2` of the case consequently use different units for height and weight. If the source MDB schema had decided to use `kg` and `cm` as units we could map the units either by type mapping or by direct mapping in each case.

The type mapping case would look like this:

Source type:

```
typedef float inches;
typedef float pounds;
```

Target type:

```
typedef float centimeters;
typedef float kilograms;
```

The type mapping would be:

```
MAP_TYPE centimeters = inches * 2.54;
END_MAP_TYPE;

MAP_TYPE kilograms = pounds * 2.24;
END_MAP_TYPE;
```

The example in the next section shows how we can use the type mappings we defined here.

7.6.3 Different Levels of Precision

Conflict Conflicts of type "Different Levels of Precision" from table 4.4 arise when semantically equivalent attributes draw values from domains with different cardinalities. This difference in cardinality results in different scales of precision for similar data.

Resolution This type of conflict is resolved by defining a mapping between the domains of semantically equivalent attributes. The mapping can be done either by creating a special (static) object_type, as a lookup-table, with information about the bounds needed for the mapping or defining a type mapping on range. Since the cardinality of these domains are different, we define a many-to-one mapping for converting a value from a more precise domain to a value from a less precise domain.

Example The problem arises with the attribute `bracket` of the case which has a numeral representation in `Schema 1` and a string representation in `Schema 3`. With a few `typedef` definitions the mapping would be range-wize from numerical values to strings as follows:

Source type:

```
typedef integer bracket_num;
```

Target type:

```
typedef string bracket_char;
```

A possible mapping could be:

```
MAP_TYPE bracket_char = bracket_num;
   ''upper''  :- 500000..10000000;
   ''middle'' :- 250000..499999;
   ''lower''  :- 0..249999;
END_MAP_TYPE
```

Here follows an example approach using the type mapping defined here (and in the previous section). We have used the types defined in this and the previous section where appropriate. The example integrates the employee information from `Schema 1` and `Schema 3` into a target object type, `All_Employee`:

Source object types:

Schema 1:

```
interface Employee{
   extent employees;
   key ssn;

   string name;
   integer ssn;
   string position;
   }
```

```
interface Emp_Other{
   extent emp_others;
   key ssn;

   integer ssn;
   integer age;
   pounds wt_in_lb;
   inches ht_in_in;
   integer salary;
   float   bonus;
   integer tax;
   bracket_num bracket;
   }
```

Schema 3:

```
interface Employee{
   extent employees;
   key ssn;

   string name;
   integer ssn;
   string position;
   }
```

```
interface Emp_Personal{
   extent emp_personals;

   integer ssn;
   integer age;
   kilograms wt_in_kg;
   centimeters ht_in_cm;
   }
```

```
interface Emp_Tax{
   extent emp_taxes;

   integer ssn;
   integer salary;
   float bonus;
   integer tax;
   bracket_char bracket;
   }
```

Target object type:

```
interface All_Employee{
   extent all_employees;
   key ssn;

   string name;
   integer ssn;
   string position;
   integer age;
   kilograms wt_in_kg;
   centimeters ht_in_cm;
   integer salary;
   float bonus;
   integer tax;
   bracket_char bracket;
   }
```

The mapping would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3;

MAP  All_Employee <- SCHEMA1.Employee;
   name :- name;
   ssn  :- ssn;
   position :- position;
END_MAP;

MAP  All_Employee <- SCHEMA1.Emp_other;
   ssn :- ssn;
   age :- age;
   wt_in_kg :- (kilograms) wt_in_lb;
   ht_in_cm :- (centimeters) ht_in_in;
   salary :- salary;
   bonus :- bonus;
   tax :- tax;
   bracket :- (bracket_char) bracket;
END_MAP;

MAP  All_Employee <- SCHEMA3.Employee;
   name :- name;
   ssn  :- ssn;
   position :- position;
END_MAP;

MAP  All_Employee <- SCHEMA3.Emp_personal;
   ssn :- ssn;
   age :- age;
   wt_in_kg :- wt_in_kg;
   ht_in_cm :- ht_in_cm;
END_MAP;

MAP  All_Employee <- SCHEMA3.Emp_tax;
   ssn :- ssn;
   salary :- salary;
   bonus :- bonus;
   tax :- tax;
   bracket :- bracket;
END_MAP;

END_SCHEMA_MAP;
```

7.7 Homogenizing Attributes

An object type is a sequence of attributes; an attribute and its domain qualify an object type by defining membership criteria for instances to belong to that object type. Similarly, the "signature" of a target object type qualifies it, and the mapping statements determine how this target object type is to be materialized. However, each `MAP` statement must retrieve objects from the CDBs such that their attribute values conform to the interface of the target object type. Thus, each corresponding attribute of the CDB object types being integrated must be redefined and appropriately transformed such that each attribute is compatible with the interface of the target object type. We describe such transformations below.

7.7.1 Type Coercion

Conflict Conflicts of type "attribute data type" in table 4.2 arise when the domains (types) are different for semantically equivalent attributes.

Resolution In many cases it is possible to resolve this conflict by coercing the type of one attribute to another type, thus homogenizing the attributes in consideration. Such a coercion is made possible in ODL-M by either casting directly where it is meaningful, or defining an explicit type mapping. We may or may not loose information in such a coercion. For example it is always possible to convert an integer value from a CDB to a real in the MDB and back. However it is likely that a real value from a CDB will be truncated when converted to an integer in the MDB and thereby losing information.

Table 7.1 shows various meaningful type coercions.

Coercion	BOOLEAN	CHAR(n_1)	INTEGER	FLOAT
BOOLEAN	BOOLEAN	(ad hoc)	INTEGER	FLOAT
CHAR(n_2)	(ad hoc)	CHAR($max(n_1, n_2)$)	(ad hoc)	(ad hoc)
INTEGER	INTEGER	(ad hoc)	INTEGER	FLOAT
FLOAT	FLOAT	(ad hoc)	FLOAT	FLOAT

Table 7.1: Type Coercion Rules

Example An example of coercion from the case schemas could occur if we were to integrate the `Employee` object type of `Schema 5` with any of the other `Employee` object types from the other schemas. The attribute `ssn` of `Employee` in `Schema 5` has the type CHAR(or `string` in ODL) while the other equivalent `ssn` attributes in the other schemas have the type INTEGER. In this case, we could do a simple `atoi(ssn)`[1] to resolve the type mismatch providing the `CHAR ssn` is a digit string representing the

[1] ASCII-to-Integer conversion

ssn. Otherwise one could define a specific type mapping from `CHAR` to `INTEGER` with the type mapping construct of ODL-M.

7.7.2 Extraction of a Composition Hierarchy

Conflict Composition hierarchies occur naturally in OODBs. Conflicts of type "attribute omposition" in table 4.2 arise when there are structurally differences in related classes such that the domain of a semantically equivalent attribute in one is a user-defined class whereas that in another class is an atomic type. This situation occurs when integrating OODBs with translated RDBs (to OOCDM) or other OODBs.

Resolution In general it is possible to combine the different conflict resolution types to achieve a resolution. However a frequent method to resolve the conflict could be to "extract" the attributes needed for an MDB object type by use of mapping constructs.

Example The `Grad_student` object type of `Schema 2` inherently has information of faculties by the `fname`, `fssn`, `major` (same as dept. name) and `frank` attributes. We use this information in the following example where we integrate the faculty concepts of the underlying systems into an `All_Faculty` object type in the global schema.

Source object types:

Schema 1:

```
interface Faculty{
   extent faculties;
   key ssn;

   string name;
   integer ssn;
   string dept;
   string rank;
   }
```

Schema 2:

```
interface Grad_Student{
   extent grad_students;
   key fssn;

   string sname;
   integer sssn;
   string major;
   float gpa;
   string fname;
   integer fssn;
   string frank;
   string thesis_title;
   }
```

Schema 3:

```
interface Faculty{
   extent faculties;
   key ssn;

   string lastname;
   string firstname;
   integer ssn;
   string dept;
   string rank;
   }
```

Schema 4:

```
interface Faculty:Employee{
   extent faculties;

   string dept;
   string rank;
   }
```

Schema 5:

```
interface Faculty:Employee{
   extent faculties;

   Department dept;
   string rank;
   }
```

Target object type:

```
interface All_Faculty{
   extent all_faculties;
   key ssn;

   string name;
   integer ssn;
   string dept;
   string rank;
   }
```

The mapping of these object types would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA2, SCHEMA3, SCHEMA4, SCHEMA5;

COPY All_Faculty <- SCHEMA1.Faculty;

MAP All_Faculty <- SCHEMA2.Faculty;
   name :- fname;
   ssn :- fssn;
   dept :- major;
   rank :- frank;
END_MAP;

MAP All_Faculty <- SCHEMA3.Faculty;
   name :- concatstring(lastname,firstname);
   ssn :- ssn;
   dept :- dept;
   rank :- rank;
END_MAP;

COPY All_Faculty <- SCHEMA4.Faculty;

MAP All_Faculty <- SCHEMA5.Faculty;
   name :- name;
   ssn :- atoi(ssn);
   dept :- dept;
   rank :- rank;
END_MAP;

END_SCHEMA_MAP;
```

7.7.3 Default Values

Conflict This conflict type is related to the conflict with the same name in table 4.2. This type arises when the default values of semantically equivalent attributes in different CDBs are different. The problem may show when updating against the MDB schema.

Resolution The conflict can be resolved in a manner similar to the case of missing but implicit attributes (see description in the "Horizontal Merges" section). The example below also gives a possible solution.

Example The `bonus` attribute in `Schema 1` may have a default value of `10%`, whereas in `Schema 3` the actual bonus value is expected to be provided when the object is instantiated. Thus, if the `Employee` objects were to be integrated, choosing a default value would cause problems at update time. However, as long as the MDB schema has an update constraint of always providing a value, the problem is avoided.

7.7.4 Attribute Concatenation

Conflict Information can be represented at different levels of detail, especially when represented as character strings. Thus, conflicts of type one-to-many attributes (which is a special case of the "many-to-many attributes" in table 4.2) arise if information captured by a single attribute in one CDB class is equivalent to that in more than one attribute belonging to another CDB class.

Resolution This type of conflict is resolved by defining an operator for concatenating attributes with the same domains (possibly coerced to the same domains). In general we have some operator `concatdomain()` which takes as its argument a list of attributes and returns the logical concatenation of these attributes.

Example In `Schema 3` the name of a person is broken into `firstname` and `lastname`, while it is simply `name` in the other CDBs. In the example above, integrating the `ALL_Faculty` object type, we included an example of string concatenation, using a special `concatstring` function for the purpose, in the `All_Faculty <- SCHEMA3` map.

7.8 Horizontal Merges

A *horizontal merge* is a means to homogenize CDB classes by taking the union of all instances materialized from each CDB class. There are two kinds of horizontal merges: *union compatible* and *extended union compatible*. The union compatible merge allows the user to integrate classes across CDBs such that the resulting target class has a signature that is very similar to that of the CDB classes. The extended union compatible merge extends this notion to provide a means to deal with inheritance hierarchies.

7.8.1 Union Compatible

Union Compatibility: Two classes are *union compatible* if and only if they have equivalent signatures.

NoteSCHEMA1 not have to be identical since we may use simple transformations such as renaming or coercion. Thus, two signatures are equivalent if and only if for each attribute in one signature there exists a corresponding attribute in the other signature such that the attributes can be obtained from the other after due transformation. There are three kinds of union compatible merges: no structural conflicts, when attribute is missing, and when attribute is missing but value is implicit.

7.8.1.1 No structural Conflicts

Conflict Conflicts of type "one-to-one class" in table 4.1 arise when various CDB classes have similar or even identical definitions. In the simple case, there is no conflict.

Resolution The simple case is resolved by simple object type mapping from the underlying CDB object types which match the MDB target object type to. The integrated object type will then become the union of the underlying instances, automatically pruned by the key attributes. This type of simple map is usually employed in conjunction with other conflict resolution operations described in this chapter. A merge is simple union compatible if the CDB attributes are transformed to be compatible with the interface of the integrated object type such that the integrated object type is updatable.

Example As an example let us define an MDB integrated class, `All_Course`, representing the union of the courses in the underlying schemas:

Source object types:

Schemas 1 and 3:

```
interface Course{
   extent courses;
   key cno;

   string cname;
   integer cno;
   }
```

Schemas 4 and 5:

```
interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   Set<Course> prereq;
   }
```

Target MDB object type:

```
interface All_Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   }
```

The mapping would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3, SCHEMA4, SCHEMA5;

COPY All_Course <- SCHEMA1.Course;
COPY All_Course <- SCHEMA3.Course;

MAP All_Course <- SCHEMA4.Course;
   cname :- cname;
   cno   :- cno;
END_MAP;

MAP All_Course <- SCHEMA4.Course;
   cname :- cname;
   cno   :- cno;
END_MAP;

END_SCHEMA_MAP;
```

7.8.1.2 Missing Attributes

Conflict Conflicts of type "missing attribute" in the "one-to-one class" case in table 4.1 arise when the numbers of attributes in similar classes across CDBs are different.

Resolution One way of resolving this conflict could be to coerce nonexistent attributes in the CDBs to `NA`[2]. Alternatively we could map the object types that resemble each other closely by integrating them separately in such a way that the object type with fewer attributes will be a superclass of the other, provided that the object types in question induce a natural inclusion relationship.

Example As an example of the latter mentioned resolve method, we define an inheritance hierarchy for students and graduate students in `schema 4` and `schema 5` as follows:

Source object types:
Schema 4:

```
interface Student{
   extent students;

   string name;
   integer ssn;
   string major;
   float gpa();
   }
```

```
interface Gradstudent:Student{
   extent gradstudents;

   Set<Faculty> advisor;
   }
```

[2]Not Applicable

Schema 5:

```
interface Student{
   extent students;

   string fname;                        interface Gradstudent:Student{
   string lname;                           extent gradstudents;
   string ssn;
   string major;                           Faculty advisor;
   float gpa();                            Set<Faculty> committee;
   }                                       }
```

Target object types:

```
interface All_Student{
   extent all_students;

   string name;
   integer ssn;
   string major;
   float gpa();
   }


interface All_Gradstudent:All_Student{
   Set<All_Faculty> advisor;
   }

interface All_Gradstudent_C:All_Gradstudent{
   Set<All_Faculty> committee;
   }
```

The mapping for this inheritance hierarchy would be:

```
SCHEMA_MAP MDB <- SCHEMA4, DDB5;

COPY All_Student <- SCHEMA4.Student;

MAP All_Student <- SCHEMA5.Student;
   name :- concatstring(lname, fname);
   ssn :- ssn;
   major :- major;
   gpa :- gpa();
END_MAP:

COPY All_Gradstudent <- SCHEMA4.Gradstudent;
```

```
BUILD All_Gradstudent <- SCHEMA5.Gradstudent;
WHERE NOT EXISTS(SCHEMA5.Gradstudent.committee);
   name :- concatstring(lname, fname);
   ssn :- atoi(ssn);
   major :- major;
   gpa() :- gpa();
   advisor :- advisor;
END_BUILD;

BUILD All_Gradstudent_C <- SCHEMA5.Gradstudent;
WHERE EXISTS(SCHEMA5.Gradstudent.committee)
   name :- concatstring(lname, fname);
   ssn :- atoi(ssn);
   major :- major;
   gpa() :- gpa();
   advisor :- #Set<Faculty>.insert(advisor);
   committee :- committee;
END_BUILD;
END_SCHEMA_MAP;
```

In this mapping we first mapped the "ordinary" students from `Schema 4` and `Schema 5` into the `All_Student` class, then we mapped the graduate students of `Schema 4` to a subclass of `All_Student`: `All_Gradstudent`. We also mapped those graduate students from `Schema 5` that conformed to `All_Gradstudent`. Finally we defined a subclass of `All_Gradstudent`, namely `All_Gradstudent_C` that could handle the objects mapped from the graduate students of `Schema 5`. The mapping is graphically described in figure 7.4.

7.8.1.3 Missing Attributes with Implicit Value

Conflict Conflicts of type "missing but implicit attribute" in the "one-to-one class" case in table 4.1 arise when an attribute is missing but can be implicitly given a default value derived from the information available.

Resolution In general, one can resolve this conflict by an expression for the missing attribute in the form *cdb_attr_name* == *value_expression* where *cdb_attr_name* is the name of the attribute in the CDB object type that has a default value denoted by *value_expression*. Further the above mentioned expression will appear on the right side of the mapping operator ':-'.

Example Suppose we use the `Student` object type in `Schema 3` and its attribute called `type` to denote whether a student is a graduate or undergraduate student. But we know that the `Student` object types in the other CDBs denote only undergraduate students. Thus, one way to integrate these object types would be to think

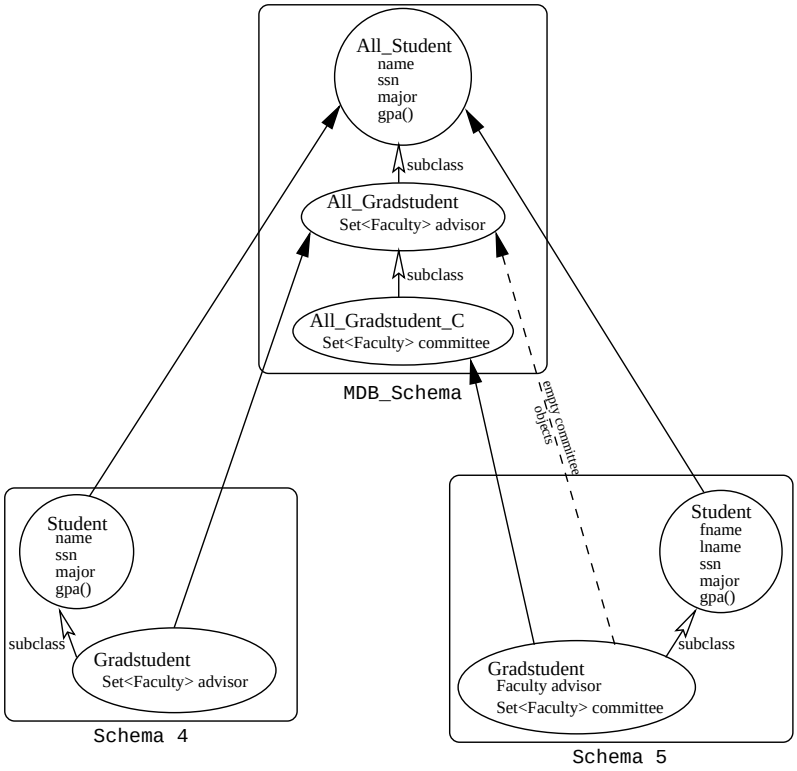

Figure 7.4: Mapping of students and graduate students.

of the `Student` object types in all CDBs except `Schema 3` as having an attribute `student_type` with a default value denoting undergraduate students.

7.8.2 Extended Union Compatible

The notion of union compatibility needs to be extended to deal with inheritance hierarchies. As one goes lower in such a hierarchy, classes tend to have more attributes defined in their respective signatures or to have attributes with more specialized domains. A class C_1 can be a subclass of C_2 if and only if the signature of C_2 subsumes that of C_1 and there exists an inclusion relationship between C_1 and C_2 (C_2 `subsumes` C_1, see section 4.2). This means that for each attribute of C_2, there is a corresponding attribute in C_1 such that its domain is union compatible (in the sense defined in the previous section) with that in C_2. An inheritance hierarchy also implies a set inclusion relationship between the instances of a class and its subclasses.

Extended union compatible: Given two classes, C_1 and C_2, when the signature of C_2 subsumes that of C_1, and the extent of C_1 is a subset of the extent of C_2, then C_1 and C_2 are said to be *extended union compatible*.

7.8.2.1 For Class Inclusion

Conflict Conflicts of type "class inclusion" in table 4.1 arise when similarly related classes are distributed across more than one CDB. A more complex situation is when an inheritance hierarchy from one OODB is to be integrated with a related inheritance hierarchy from another OODB that has a different structure. This may be found to be a compound conflict which can be resolved by decomposing it into the more primitive conflicts. Thus, when integrating two inheritance hierarchies, we must first integrate two CDB classes using other resolution techniques described in this section, such that the resulting MDB inheritance hierarchy reflects the inclusion relationships in the CDB hierarchies.

Resolution We use the notion of an extended union compatible merge to resolve class inclusion conflicts. We do this by organizing a set of related CDB object types into a generalization hierarchy.

Example As an example, we define a hierarchy of courses that existed in neither of the CDB schemas, it only existed between them. We use the `All_Courses` object type we defined before and define a subclass of it. `All_Course_R`, which represents restricted courses. Neither of the CDB schemas had both restricted and non-restricted courses defined in a hierarchy. This mapping example defines this hierarchy:

Source object types:

Schema 1:

```
interface Restricted_Course{
   extent restricted_courses;

   string cname;
   integer cno;
   string major;
   }
```

Schema 3:

```
interface Course_Restriction{
   extent course_restrictions;

   string cno;
   string major;
   string prereq_cno;
   }
```

Schema 3:

```
interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   }
```

Schema 4 and 5:

```
interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   Set<Course> prereq;
   }
```

Target object types:

```
interface All_Course_R:All_Course{
   extent all_course_rs;

   string major;
   Set<All_Course> prereq;
   }
```

The mapping for this subclass would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3, SCHEMA4, SCHEMA5;
MAP All_Course_R <- SCHEMA1.Restricted_Course;
   cname :- cname;
   cno   :- cno;
   major :- major;
   prereq :- NA;
END_MAP;

BUILD All_Course_R <- SCHEMA3.Course, SCHEMA3.Course_Restriction;
WHERE Course_Restriction.prereq = Course.cno;
   cname :- Course.cname;
   cno   :- Course.cn
   major :- Course_Restriction.major;
   prereq :- #Set<All_Course>.insert(#All_Course(Course.cname,Course.cno));
END_BUILD;

BUILD All_Course_R <- SCHEMA4.Course;
WHERE EXISTS SCHEMA4.prereq;
   cname :- cname;
   cno   :- cno;
   major :- NA;
   prereq :- prereq;
END_BUILD;

BUILD All_Course_R <- SCHEMA5.Course;
WHERE EXISTS SCHEMA5.prereq;
   cname :- cname;
   cno   :- cno;
   major :- NA;
   prereq :- prereq;
END_BUILD;
END_SCHEMA_MAP;
```

The intension with this mapping was to classify the courses into those that have restrictions, and those that do not.

7.8.2.2 Attribute Inclusion

Conflict Conflicts of type "attribute inclusion" in table 4.2 arise when there is an inclusion relationship between two or more attributes. This conflict falls into a category distinct from that in which attributes have different names or data types, as discussed before. An inclusion relationship between two attributes can be used to induce a natural inheritance hierarchy among the corresponding classes in the MDB schema.

Resolution Attribute inclusion and class inclusion are different kinds of conflicts. However, since an attribute inclusion relationship induces an class inclusion relationship, both conflicts can be resolved using the extended union compatible merge operation.

Example We construct two small new schemas here for the purpose of demonstrating this conflict resolution. In the following two schemas the attribute `son_name` can be regarded as being included in the attribute `child_name`.

```
Schema 6:
interface People{
   string name;
   integer age;
   string son_name;
   }
```

```
Schema 7:
interface Person{
   string name;
   integer age;
   string child_name;
   }
```

The inclusion relationship between `son_name` and `child_name` can induce a natural inclusion relationship between `People` and `Person` such that the former includes the latter. Thus we can integrate the two as follows:

Target object types:

```
interface Parents{
   string name;
   integer age;
   string child_name;
   }
```

```
interface Parents_of_Men:Parents{
   string name;
   integer age;
   string son_name;
   }
```

And the mapping a simple `COPY` for both:

```
SCHEMA_MAP MDB <- SCHEMA6, SCHEMAA7;

COPY Parents <- People;

COPY Parents_of_Men <- Person;

END_SCHEMA_MAP;
```

7.9 Vertical Merges

A vertical merge is used to integrate a number of classes or attributes across one or more CDBs into a single class at the MDB level representing the construct that spanned the CDB schemas.

7.9.1 Many-to-Many Classes

Conflict Classes in CDB schemas may be defined in different ways for various reasons, such as to remove redundant data or to reduce possibilities of inconsistency during updates or improve the efficiency of evaluating queries. This causes a given concept to be decomposed into a number of classes. The normalization of a RDB schema is an example of such a decomposition. Thus, conflicts of type "many-to-many classes" in table 4.1 arise when, for example, integrating relating concepts that are normalized in different degrees in CDB schemas translated from RDB schemas or when integrating a concept represented by many object types with a single MDB object type.

Resolution We use the vertical merge to integrate many CDB classes into one class representing the class that spans across the CDB schemas. In order to integrate many CDB classes into many MDB classes, we need to perform a sequence of vertical merges; we consider the many-to-many classes conflict as a composite case of the many-to-one class conflict.

Example Our example of constructing the `All_Employee` object type in section 7.6.3was an example which homogenized the object types `Employee` and `Emp_other` from `Schema 1` and `Employee`, `Emp_personal` and `Emp_tax` from `Schema 3`. Since personal or tax information about employees is not available in `Schema 4` or `Schema 5`, the extent of `All_Emp_Info` only contains instances as result of a vertical merge between `Schema 1` and `Schema 3`. The result is graphically represented in table 7.5.

All_Employee

ssn	name	position	age	wt_in_kg	ht_in_cm	salary	bonus	tax
Employee			Emp_Other					
ssn	name	position	age	wt_in_lb	ht_in_in	salary	bonus	tax
⋮ extent from schema 1								
Employee			Emp_personal			Emp_tax		
ssn	name	position	age	wt_in_kg	ht_in_cm	salary	bonus	tax
⋮ extent from schema 3								

Figure 7.5: Graphical representation of All_Employee extent

7.9.2 Class-versus-Attributes

Conflict Conflicts of type "class-versus-attributes" in table 4.3 arise when a concept or part of a concept is represented as an class in one CDB but as a set of attributes, possibly belonging to a related class, in another CDB.

Resolution This type of conflict can be resolved in two ways; either by splitting an object type into two or more parts or by integrating two object types (or parts of them) into one by performing a vertical merge. Note that this is distinct from an attribute concatenation because in that case the domain of each attribute must be the same; there are no such restrictions here.

Example Look at the `address` field of the `Under_Grad` object type of `Schema 1`. It is used to represent the address of each undergraduate student. The same information can be found as an object type of itself, namely `Address`, in `Schema 3`. We can resolve this conflict in two ways. One way is to split the `Under_Grad` object type in `Schema 1`;

Source object types:

Schema 1:

```
interface Under_Grad{
   extent under_grads;
   key ssn;

   string name;
   integer ssn;
   string major;
   string address
   }
```

Schema 3:

```
interface Student{
   extent students;
   key ssn;

   string lastname;
   string firstname;
   integer ssn;
   string type;
   string major;
   }
```

```
interface Address{
   extent addresses;
   keys ssn,street,city,zip;

   integer ssn;
   string street;
   string city;
   string zip;
   }
```

Target object types:

```
interface All_Under_Grad{
   extent all_under_grads;
   key ssn;

   string name;
   integer ssn;
   string major;
   }
```

```
interface All_Address{
   extent all_addresses;

   integer ssn;
   string address;
   }
```

The mapping to these targets would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3;

MAP All_Under_Grad <- SCHEMA1.Under_Grad;
   name :- name;
   ssn :- ssn;
   major :- major;
END_MAP;

MAP All_Under_Grad <- SCHEMA3.Student;
   name :- concatstring(lastname,firstname);
   ssn :- ssn;
   major :- major;
END_MAP;

MAP All_Address <- SCHEMA1.Under_Grad;
   ssn :- ssn;
   address :- address;
END_MAP;

BUILD All_Address <- SCHEMA3.Student,SCHEMA3.Address;
WHERE SCHEMA3.Student.ssn = SCHEMA3.Address.ssn;
   ssn :- SCHEMA3.Student.ssn;
   address :- concatstring(street,city,zip);
END_MAP;

END_SCHEMA_MAP;
```

The other way to resolve this conflict type is to integrate the `Address` and `Student` object types in `Schema 3` into a single target object type by performing a simple vertical merge.

Using the same source object types as in the previous example the target object type now looks like this:

```
interface All_Under_Grad_X{
   extent all_under_grad_xs;
   key ssn;

   string name;
   integer ssn;
   string major;
   string address;
   }
```

And the mapping this time will be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3;

COPY All_Under_Grad_X <- SCHEMA1.Under_Grad;

BUILD All_Under_Grad_X <- SCHEMA3.Student, SCHEMA3.Address;
WHERE SCHEMA3.Student.ssn = SCHEMA3.Address.ssn;
   name :- concatstring(lastname,firstname);
   ssn :- SCHEMA3.Student.ssn;
   major :- major;
   address :- concatstring(street,city,zip);
END_BUILD;

END_SCHEMA_MAP;
```

7.9.3 Aggregation Hierarchies

Conflict Conflicts of type "class structure" in table 4.1 and "attribute composition" in table 4.2 in combination denote aggregation hierarchy conflicts. Thus, the conflicts arising when integrating aggregation hierarchies in different OODBs that are similar but have different structures are said to be due of the aggregation hierarchy.

Resolution To resolve this conflict we can perform a vertical merge of the CDB object types that compose the target object type being defined.

Example To exemplify this conflict we define a target object type called `Advisement`. The `Advisement` class integrates concepts at class and attribute level into a global object type. Further, `Advisement` gives info on who (advisor) gives advisement to whom (advisee) at what department (dept) for which thesis (thesis).

Source object types:
Schema 2:

```
interface Grad_Student{
   extent grad_students;

   string sname;
   integer sssn;
   string major;
   float gpa;
   string fname;
   integer fssn;
   string frank;
   string thesis_title;
   }
```

Schema 3:

```
interface Student{
   extent students;

   string lastname;
   string firstname;
   integer ssn;
   string type;
   string major;
   }
```

```
interface Faculty{
   extent faculties;

   string lastname;
   string firstname;
   integer ssn;
   string dept;
   string rank;
   }
```

```
interface Graduate_Info{
   extent graduate_infos;

   integer ssn;
   integer advisor_ssn;
   }
```

```
interface Thesis{
   extent thesises;

   string title;
   integer ssn;
   float grade
   }
```

Schema 4:

```
interface Student{
   extent students;

   string name;
   integer ssn;
   string major;
   float gpa();
   }
```

```
interface Thesis{
   extent thesises;

   string title;
   Gradstudent author;
   string status;
   }
```

```
interface Gradstudent:Student{
   extent gradstudents;

   Set<Faculty> advisor;
   }
```

```
interface Student{
   extent students;

   string fname;
   string lname;
   string ssn;
   string major;
   float gpa();
   }
```

```
interface Faculty:Employee{
   extent faculties;

   string dept;
   string rank;
   }
```

```
interface Gradstudent:Student{
   extent gradstudents;

   Faculty advisor;
   Set<Faculty> committee;
   }
```

```
interface Faculty:Employee{          interface Thesis{
   extent faculties;                    extent thesises;

   Department dept;                     string title;
   string rank;                         Gradstudent author;
   }                                    string status;
                                        }
```

And the target object type to be integrated into is:

```
interface Advisement{
   string advisor;
   Set<string> advisee;
   string dept;
   string thesis;
   }
```

The mapping would be:

```
SCHEMA_MAP MDB <- SCHEMA2, SCHEMA3, SCHEMA4, SCHEMA5;

MAP Advisement <- SCHEMA2.Grad_student;
   advisor :- fname;
   advisee :- #Set<string>.insert(string(sname));
   dept :- major;
   thesis :- thesis_title;
END_MAP;

BUILD Advisement<- SCHEMA3.Student,SCHEMA3.Graduate_Info,SCHEMA3.Faculty,SCHEMA3.Thesis;
WHERE Student.ssn = Graduate_Info.ssn AND
      Graduate_Info.advisor_ssn = Faculty.ssn AND
      Student.ssn = Thesis.ssn;
   advisor :- concatstring(Faculty.lastname,Faculty.firstname);
   advisee :- concatstring(Student.lastname, Student.firstname);
   dept :- Student.major;
   thesis :- Thesis.title;
END_BUILD;

BUILD Advisement <- SCHEMA4.Gradstudent, SCHEMA4.Faculty, SCHEMA4.Thesis;
WHERE Faculty IN Gradstudent.advisor AND Thesis.author == Gradstudent;
   advisor :- Faculty.name;
   advisee :- Gradstudent.name;
   dept :- Gradstudent.major;
   thesis :- Thesis.title;
END_BUILD;
```

```
BUILD Advisement <- SCHEMA5.Gradstudent, SCHEMA5.Faculty, SCHEMA5.Thesis;
WHERE Faculty == Gradstudent.advisor AND Thesis.author == Gradstudent;
   advisor :- Faculty.name;
   advisee :- Gradstudent.name;
   dept :- Gradstudent.major;
   thesis :- Thesis.title;
END_BUILD;

END_SCHEMA_MAP;
```

Again we show the mapping as a pictorial representation in figure 7.6.

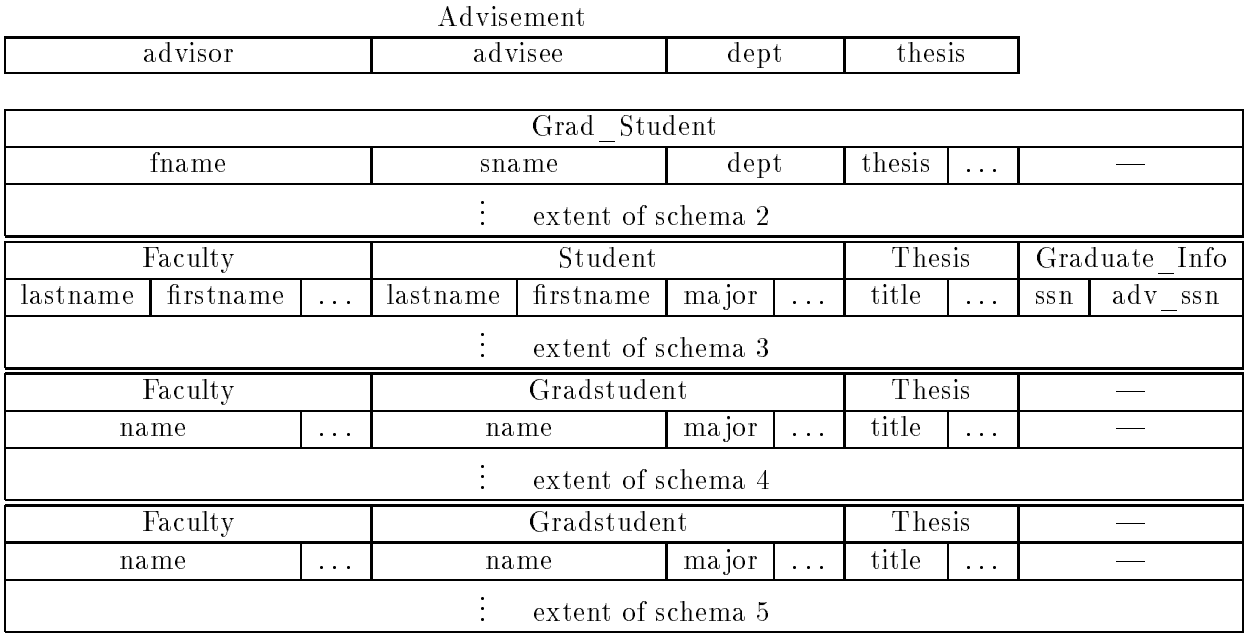

Figure 7.6: Pictorial representation of Advisement extent

7.10 Mixed Merges

A *mixed merge* is a combination of vertical and horizontal merges. It is used to integrate arbitrarily fragments of classes from one or more CDBs to define a target class. In general, compound conflicts[3], as we defined them in section 4.3.5, are resolved by a combination of the two merge types and other simpler resolution techniques. In this manner we can handle arbitrarily complex conflicts by breaking them down into elements that fit into our resolution technique classification.

[3]Conflicts that appear in combination

7.11 Homogenizing Methods

Conflict Method conflict can be interpreted as attribute conflicts. We have discussed the conflict and touched upon how to resolve it in section 4.3.2.

Resolution As mentioned earlier, methods can be treated as attributes(see section 7.7). In spite of this it is not possible in general to define methods in the interface of a target object type. However, in some cases we can define methods when integrating. In particular we can redefine a method in a target object type if it is possible to define that method as a derived attribute using some expression.

Example As an example we define a target object type called `All_Student_Method`. It will represent the global student class that incorporates students with a grade-point-average calculated. In `Schema 1` and `Schema 3` we have to calculate the `gpa` in the map, while the `Student` class of `Schema 4` and `Schema 5` already has the `gpa()` operation, so the mapping in these cases is trivial.

Source object types:

Schema 1:

```
interface Under_Grad{
   extent under_grads;

   string name;
   integer ssn;
   string major;
   string address
   }
```

```
interface Enroll{
   extent enrolls;

   string Cno;
   integer fssn;
   integer sssn;
   float grade;
   }
```

```
interface Student{
   extent students;

   string lastname;
   string firstname;
   integer ssn;
   string type;
   string major;
   }
```

Schema 3:

```
interface Enroll{
   extent enrolls;

   string cno;
   integer fac_ssn;
   integer stud_ssn;
   float grade
   }
```

```
Schema 4:
interface Student{
   extent students;

   string name;
   integer ssn;
   string major;
   float gpa();
   }
```

```
Schema 5:
interface Student{
   extent students;

   string fname;
   string lname;
   string ssn;
   string major;
   float gpa();
   }
```

Target object type:

```
interface All_Student_Method{
   extent all_students;

   string name;
   integer ssn;
   string major;
   float gpa();
   }
```

The mapping would be:

```
SCHEMA_MAP MDB <- SCHEMA1, SCHEMA3, SCHEMA4, SCHEMA5;
BUILD All_Student_Method <- SCHEMA1.Under_Grad, SCHEMA1.Enroll;
WHERE Under_Grad.ssn = Enroll.sssn;
   name :- Under_Grad.name;
   ssn :- Under_Grad.ssn;
   major :- Under_Grad.major;
   gpa() :- AVERAGE(Enroll.grade) BY Enroll.ssn;
END_BUILD;

BUILD All_Student_Method <- SCHEMA3.Student, SCHEMA3.Enroll;
WHERE Student.ssn = Enroll.sssn;
   name :- concatstring(Student.lastname,Student.firstname);
   ssn :- Student.ssn;
   major :- Student.major;
   gpa() :- AVERAGE(Enroll.grade) BY Enroll.ssn;
END_BUILD;

COPY All_Student <- SCHEMA4.Student;
COPY All_Student <- SCHEMA5.Student;

END_SCHEMA_MAP;
```

7.12 Semantic Proximity in ODL-M

In the previous sections we have discussed resolving structural conflicts using the ODL-M mapping language. Although we did not mention it, we constantly used semantics to some degree in our mapping. We will always have to look at our model from two viewpoints; one structural and one semantical. The structure has no meaning without interpreting the semantics of it and the semantics have no usefulness without a structure to realize them in. Thus these two concepts are tightly bound and our analysis should consider them both.

In chapter 4 we introduced a function that measured semantic proximity called `semPro()` and we would like to utilize the `semPro()` operation as an aid in our schema integration process. Koren [Kor94] has done work toward this goal. He suggests an expansion of the ODMG type hierarchy to enhance the ODMG object model with support for semantic proximity. The main contribution of the expansion is the defined type `Context` which is an abstract type which provides semantic proximity to its subclasses. The subclasses will be types which are used when describing new types of databases: `Type`, `Schema`, and `Subschema`. Figure 7.7 shows the connection to the original full type hierarchy(see fig.B.1 in appendix B).

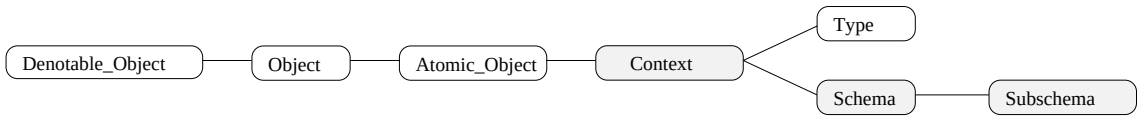

Figure 7.7: The new types Context, Schema, and Subschema

The interface description of the `Context` type includes the `semPro` function as:

$$semPro(oid_1, oid_2) \rightarrow t : semProvalues$$

where `semProvalues` is an enumeration of the return values of `semPro`, namely "*Semantic Equivalence*", "*Semantic Relationship*", "*Semantic Relevance*", "*Semantic Resemblance*", and "*Semantic Incompability*".

Further we define new characteristics to the type `Type`. The new characteristics are four new instance operations:

$context() \rightarrow c : Context$
$role() \rightarrow r : String$
$domain() \rightarrow d : Set < Atomic\_Object >$
$state() \rightarrow s : Set < Atomic\_Object >$

The operation `context()` returns the instance of `Context` in which the type is defined. The operation `role()` returns a user-supported string which is a name on the role of the type in its context. This is fragile because the model does not have a notion of roles. The operation `domain()` returns a set of all real world phenomena that may be represented by instances of the type.

All these operations are abstract, that is they are not implementable. The reason for specifying them as operations is that it later may be possible to make automatic, at least some of the reasoning about semantic similarity between schema objects. If semantic reasoning will be possible, these operations must be implemented by the type programmer.

The last part of `semPro`, the abstraction between the domains of the schema objects, cannot be user specified. It is the responsibility of the `semPro` to find an abstraction between the actual domains, and from the abstraction deduce the actual semantic proximity level.

The `semPro` function can hardly be implemented at current time since capturing the semantics of objects is impossible to do entirely automated by a machine. However, a pseudo-algorithm could look like this:

Is there a function $f : O_1.domain() \rightarrow O_2.domain()$?
If No: Is $O_1.role() = O_2.role()$?
    No: Semantic Incompability
    Yes: Semantic Resemblance
If Yes: Is f 1-1 and total?
    No: Semantic Relationship
    Yes: Semantic Equivalence

If this algorithm returns "Semantic Relationship" or "Semantic Equivalence" only when $O_1.context() = O_2.context()$ we have "Semantic Relevance".

From Savasere et. al [SSG+91] we have discussed a classification (see section 4.2) of schema comparisons into the four classifications: Equivalence, Inclusion, Overlap, and Disjoint. Each of these classifications can be computed automatically according to Savasere et. al [SSG+91] using the *subsume* function 4.1 defined in section 4.2. Therefore the *subsume* function might be a candidate for `semPro` to initiate an implementation. However, we will not pursue this any further, just suggest that seems possible.

With these expansions to the object model we now have ODL-M which supports the schematic mapping of constructs and to aid this process we have a built in function `semPro` to capture the semantic aspects of schemas we wish to integrate, thus we have managed to achieve our goal of analyzing from both a structural and semantic viewpoint.

A suggested use of the `semPro` function in the ODL-M mapping framework would be to first analyze classes from the considered schemas to encounter the semantic similarities between them. The search for identifying relations or possible conflicts may be guided by the class hierarchy; Instead of comparing all classes in a random manner, classes may be compared following the class-hierarchy in a top-down fashion [GCS93].

Semantic equivalence typically points out candidates for synonyms. We might find correspondences that are hard to find by only analyzing the syntax of the schemas. When all semantic similarities are found, we sort them by strength according to the taxonomy presented in figure 4.3 on page 55. The integration process should start by integrating classes and concepts that have the closest resemblance to keep the resulting schemas as simple as possible. The lesser strong similarities should be merged in with the rest until all similarities are encountered for. This way we have the closest semantic similar objects

modeled as the main objects in the target and other objects and concepts express the additions we need to complete the integration.

7.13 Implementing our Proposal

A full-developed compiler for ODL-M has not been developed yet. Because of this, we have not been able to try out the techniques in an implementation. However we have investigated possible constructs one could use to implement the mappings that ODL-M defines. An object-oriented database system called ObjectStore from Object Design is conform[4] to the ODMG-93 standard. ObjectStore features a pointer type denoted `Ref`. This is actually a feature of the ODMG-93 database standard [Cat94]. Pointers of this type are not "hard" links that are computed at compile-time, but they are rather computed at run-time. We argue that this supports our mapping constructs since they are the support for views. This means that if the mappings defined be ODL-M are implemented with the `Ref` feature, the objects defined by these mappings will only be computed at run-time when they are referenced. This is exactly what we want according to how we defined views.

Although we have a good idea of how to implement the main contribution in this thesis, we will not investigate it any further since the amount of work would be too large for this thesis. It will be a project for future work.

7.14 Summary

We have presented a classification of resolution techniques (table 7.3to resolve the conflict groups identified in the four tables, table 4.1, table 4.2, table 4.3, and table 4.4 in section 4.3 of chapter 4. The resolution techniques showed to cover our conflicts list. The only two conflict groups that were not explicitly addressed by our resolution techniques were "Class Constraints" from table 4.1 and "Attribute Integrity Constraints" from table 4.2. We explained in chapter 4 why these two conflicts groups were complex and needed ad hoc solutions that met the specific solution.

We can conclude this chapter by claiming a near complete means of resolving our conflicts specified in chapter 4. All our techniques had understandable mappings using the ODL-M language as their basis and they should not be difficult to follow by the examples. We used extent figures (fig. 7.5 and fig. 7.6) to suggest how we captured all instances of the concept we integrated from the underlying systems. The only time we did not feel we included all information possible from the underlying systems it was because of the different representations made in the underlying systems which unavoidably led to information loss (e.g. data precision conflicts).

Our approach corresponds to the *iterative* schema integration strategy described in chapter 3, some examples were *one-shot*. But in general our approach can integrate several schemas at a time and we can integrate stepwise towards the final target schemas.

[4] So they claim.

We also showed how a possible use of the `semPro` function could be implemented as an extension of the type hierarchy of ODMG. All in all our ODL-M solution seems to have good advantages for a possible means to resolve schema integration conflicts in object-oriented systems using ODL or a conforming model as its canonical model.

Part III

Conclusion and Future Work

Chapter 8

Conclusion & Future Work

We have come to our final remarks in this thesis. We will relate to our stated goal in the introductory chapter and see how we managed to reach our goal. Further we will discuss our main three tables of requirement groups defined, and discuss how our proposal met with these requirements. The three requirement group tables were structural conflicts, the canonical model, and the schema integration process and its results.

Finally we suggest some future guidelines for how this work could be carried on towards a full working system. But first we summarize briefly what our work has been focused on.

8.1 Summary

In the first part of this thesis we investigated the concepts of multidatabases and schema integration. In chapter 2 we defined the basics of multidatabases including the five- and eight-schema architecture. We also identified three key issues for a full fledged multidatabase: constructing a global schema by schema integration, processing of queries, and management of transactions. We chose to focus on schema integration and gave a more in-depth description of the schema integration process and its characteristics in chapter 3. One of the basic problems in schema integration is to identify the conflicts that can arise, and further to resolve these conflicts in an effective way. In chapter 4 we classified the conflict possibilities from two viewpoints, a schematic and a semantic, giving an overview and understanding of the complexity of schema conflicts. The schematic viewpoint was presented as a classification of structural conflicts and the semantic view was presented as a semantic measure for semantic proximity, called `semPro`. In chapter 5 we took a look at some existing approaches to manage multidatabase systems and briefly included their suggestions to schema integration.

The second part of the thesis was the proposal for a method to resolve conflicts in schema integration. In chapter 6 we defined a mapping language, ODL-M, as a schema integration support tool for mapping concepts between the underlying component schemas and the global schema in the multidatabase system. In chapter 7 we revisited the conflict types encountered in chapter 4 and suggested resolving techniques using the ODL-M mapping language we defined. We also suggested a method for using the semantic measure

defined in chapter 4 as an aid in the schema integration process so that the structural and semantic approach were integrated during the process. Our case from the introduction has been used to exemplify the conflicts we encountered in chapter 4 and also to show how these conflicts could be solved using the resolution techniques developed in chapter 7.

8.2 The Goal

The essence of our goal in this thesis was: "To identify requirements for, and propose solutions to schema integration in object-oriented multidatabase systems."

As our focus narrowed down to the schema integration issue of multidatabase systems we outlined the process of schema integration in chapter 3 and presented some alternative strategies from the literature. From this discussion we summarized our requirements to the schema integration process and its result schemas in table 3.1.

In chapter 4 we approached the complexity of conflicts between schemas from two viewpoints; one schematic and one semantic. We made a classification of possible structural conflicts that can arise and described each subgroup of conflicts. The classification was really a detailed version of our initial requirement table 1.1 from the introduction. We also presented a semantical measure called `semPro` [SK92] and gave a taxonomy of semantical similarity based on it.

Our connection to real world solutions was covered in chapter 5 where we gave an overview of some prototypes and projects on multidatabase systems and similar systems. This overview gave us some ideas as to how our problem area has been approached by others and some of these ideas we brought into our own proposal later. The main contribution of chapter 5, however, was that we would rather work in the framework of a standard instead of either creating our own framework or using one of the stand-alone frameworks discussed in this chapter. We argued that the ODMG-93 database standard [Cat94] is a state-of-the-art standard we would like to investigate further in our proposal.

To solve the four requirement conflict groups we identified in chapter 4 we first developed and defined a mapping language in chapter 7 to support our proposal. It was an extension to the ODMG ODL object model [Cat94], a supposed de facto standard that we briefly described first. Our proposed solution to resolve the conflicts was presented as a classification of techniques in chapter 7, each technique addressing a subgroup of conflicts. The resolution techniques showed to mainly cover the list of conflicts so we had a method of resolving nearly all our identified conflicts. To aid this method we suggested to use the `semPro` function as a semantical measure included in the ODL model.

In reference to the essence of our goal at the start of this section, identified requirements for schema integration in three tables, one for the conflicts we wanted to resolve, one for the canonical data model in which the schema integration process is restricted, and one for the schema integration process itself and its resulting schemas. We proposed a solution as a mapping language, called ODL-M, and demonstrated how the mapping language in the context of the ODMG database stsndard (extended), resolved our defined conflicts. Thereby we met our goal in this thesis.

8.3 Evaluation of the Requirements

In our discussion we identified three requirement groups in three tables at different levels. We here discuss how we met our requirements.

8.3.1 General Requirement Conflict Groups

In the introduction we presented a case from which we derived four groups of conflicts (see table 8.1). From these groups we wanted to go into detail of which problems belonged

Initial general requirement conflict groups	
RCG-1	Class Conflicts
RCG-2	Attribute Conflicts
RCG-3	Class vs Attribute Conflicts
RCG-4	Data Representation Conflicts

Table 8.1: General requirement conflict groups

to which groups. We achieved this in the classification of table 4.3 and further achieved resolving techniques for these conflict groups in table 7.3 by the use of the mapping language ODL-M.

We now split table 8.1 into four tables, one for each general conflict group according to our requirement tables in section 4.3, and evaluate how each specific conflict has been resolved by our proposal. In the evaluation we will use the following symbols: + means the requirement was met, +/− means the requirement was partially met, and − means the requirement was not met.

Table 8.2 gives an evaluation of conflict group RCG-1.

Conflict Group RCG-1: Class-vs-Class	
(a) One-to-One Class	
i. Class Name	
-different names for equivalent classes	+
-same name for different classes	+
ii. Class Structure	
-missing attributes	+
-missing but implicit attributes	+
iii. Class Constraints	−
iv. Class Inclusion	+
(b) Many-to-Many Classes	+

Table 8.2: Evaluation of conflict Group RCG-1

"Class Name" conflicts were resolved by our "Renaming" proposal in straightforward-chapter 7. Two union compatible horizontal merges resolved the "missing attribute" and

"missing, but implicit" conflicts that were classified as "Class Structure" conflicts. "Class Constraints" conflicts are difficult to solve in general, they are rather resolved ad hoc depending on the constraint. We developed an extended union compatible horizontal merge technique to resolve the "Class Inclusion" conflicts. Finally "Many-to-Many Classes" conflicts were resolved by a vertical merge technique.

Table 8.3 gives an evaluation of conflict group RCG-2.

Conflict Group RCG-2: Attribute-vs-Attribute	
(a) One-to-One Attribute	
i. Attribute Name	
-different names for equivalent attributes	+
-same name for different attributes	+
ii. Attribute Constraints	
-integrity constraints	−
-data values	+
-composition	+
iii. Default Values	+
iv. Attribute Inclusion	+
v. Methods	+/−
(b) Many-to-Many Attributes	+

Table 8.3: Evaluation of conflict Group RCG-2

Renaming techniques were also used to resolve "Attribute Name" conflicts. Like "Class Constraints" the "Attribute Integrity Constraints" conflicts are difficult to solve in a general way – it depends on the situation. We used "Type Coercion" to resolve "Data Type" conflicts and "Extraction of a Composition Hierarchy" to resolve "Composition" conflicts. We suggested a similarity to "missing, but implicit attribute" to resolve "Default Values" conflicts. Like we did for classes, we developed an extended union compatible horizontal merge to resolve "Attribute Inclusion" conflicts. We feel our proposal for resolving "Methods" conflicts was not complete, but nevertheless resolved specific conflicts. Finally the "Many-to-Many Attributes" conflicts were resolved mainly by homogenizing with attribute concatenation.

The evaluation of the third conflict group, RCG-3, is given in table 8.4.

Conflict Group RCG-3	
Class-vs-Attribute	+

Table 8.4: Evaluation of Conflict Group RCG-3

For the "Class-vs-Attribute" conflicts we developed a resolution technique where we either split a class in two or more parts or integrated two classes into one by performing a vertical merge.

The fourth and last conflict group, RCG-4, is evaluated in table 8.5.

Conflict Group RCG-4: Different Representation for Equivalent Data	
(a) Different Expression denoting same Information	+
(b) Different Units	+
(c) Different Levels of Precision	+

Table 8.5: Evaluation of conflict Group RCG-4

These conflicts deal with different representations for equivalent data and we resolve them with different homogenizing techniques. "Different Expression denoting same Information" was resolved by an expression technique, for "Different Units" we used a units technique and the "Different Levels of Precision" conflicts we resolved by developing a precision technique.

8.3.2 Requirements for a Canonical Data Model

We stated that to overcome the problem of syntactical language differences between heterogeneous schemas we translated each schema in a native model to a canonical model. This supported schema integration by avoiding translation as part of the schema integration process. The choice of a canonical model was required to meet three properties; expressiveness, semantic relativism, and support for views (see table 8.6). We chose ODL,

Requirements for a canonical data model		
RCDM-1	Expressiveness	+
RCDM-2	Semantic Relativism	+
RCDM-3	Support for Views	+

Table 8.6: Evaluation of requirements for a canonical data model

an object-oriented model, as our canonical model for two reasons. First it is a strong model that meets our requirements. The meeting of the two first requirements was discussed according to Saltor et. al [SCG91]. They argue that the object-oriented models in general are well suited as canonical models, but their only drawback is that they do not support views as a rule. This, however, was covered in our proposal by our ODL-M extension. Second we assume that the ODMG-93 database standard [Cat94] will become an important and supported standard so our solutions in ODL/ODL-M can be implemented in conforming products.

8.3.3 Requirements for Schema Integration

The requirements for schema integration were stated in section 3.5 as: Completeness, Correctness, Minimality, Understandability, and Schema Integration Support (see table 8.7).

Requirements for schema integration		
RSI-1	Completeness	+
RSI-2	Correctness	+
RSI-3	Minimality	+
RSI-4	Understandability	+
RSI-5	Schema Integration Support	+

Table 8.7: Evaluation of requirements for schema integration

To start with the last one, we have defined a mapping language that strongly supports the schema integration process and achieves the other requirements at the same time.

The mapping process we discussed covered all our conflicts groups and thereby completely includes the underlying conflict concepts. The correctness was not always perfect, but this was due to non-avoidable information loss inherited from the designers choice of representation. A more formal approach to state completeness and correctness in each resolution technique could have been investigated, but this would be out of the scope and time bounds of this thesis and we therefore leave it as future work work.

Our examples all defined target concepts that represented the union of similar concepts in the underlying systems without having to duplicate the information thereby keeping the minimality requirement.

We argued that a qualitative measure of understandability was difficult to define. However, when analyzing our techniques, we find that we mostly use mappings that are relatively easy to follow. In addition the resulting target object types were unifying concepts very similar to the ones in the source schemas or inheritance hierarchies mapped directly from the underlying schemas without any complicating steps. A strong feature for understanding the schemas and the mapping process is the type mapping (`TYPE_MAP`) construct. It strongly introduces semantics to the schemas as it attaches meaningful names to types rather than non-informative built in types.

8.4 Conclusion

We have defined and developed ODL-M, a mapping language for ODMG-93/ODL. ODL-M is a well suited mapping tool for achieving schema integration within the ODMG standard. It is based on a strong data model that meets our requirements for a canonical model. Further it supports the resolution techniques we identified to cover[1] all our conflicts groups. It does so by meeting our requirements for schema integration to a good degree.

[1] Well, mostly

8.5 Future Work

Having given a conclusion to our contributions, we finally give some pointers to how this work could be carried on. We have divided our suggestions into a general and a specific part in the following.

8.5.1 General Considerations

- As we mentioned the proving of completeness and correctness was argued at a non-formal level. The argument of these properties would be strongly enhanced by formal reasoning to show that our techniques do what they are intended to. But this is left as additional work as this would demand a whole new thesis work.

- In our mapping proposal and support in ODL-M we feel that the mapping of relationships might be too weak, especially because it doesn't specifically cover the `inverse` statement in ODL. This shouldn't be to hard to incorporate into ODL-M but is still left out for later development.

- The two evaluations that we failed to give a general resolve technique to were "Class Constraints" in table 8.2 and "integrity constraints" in table 8.3. These type of conflicts are difficult to resolve in general because they are often constraints specifically specified for some situation. Also their nature may be such that different constraints from different component databases are non-merge-able. The conflicts in these groups also include dependency conflicts, behavior conflicts, and key-attribute conflicts, discussed in section 3.4.2, which we haven't focused on too strong. The conflicts that arise from constraints as mentioned here might be more extensive than we thought, so work should be done to try to resolve these conflicts in general.

- Also method mapping might be lacking completeness (see table 8.3). It covers direct mapping between parameters and expression mapping, but we feel this might not be strong enough. This could however be as far as we can reach with ODL because ODL is an interface specification language and does not specify the implementation of methods. Still we should investigate this further to be sure.

- The type coercion table 7.1 of section 7.7 listed several '`ad hoc`' entries. This isn't really a lack of method, but the rules for these entries have to be resolved separately at each implementation since they depend strongly on the semantics of the corresponding types. We therefore include this as a future consideration.

8.5.2 An Implementation

Our approach to schema integration has been proposed on the basis of a standard. The next step in this process would be to design and develop a full system to implement the ideas introduced in this thesis. As we have mentioned work has been done on a ODL-to-C++ compiler [LS92]. The ODL-M language needs a compiler in order to integrate the mapping statements with the rest of the application. Since the ODL-M language is

relatively close to the EXPRESS-M language we assume it to be a feasible task to develop a compiler for ODL-M. We have ourselves touched upon a simple implementation of the source schemas with simple mappings using the `Ref` feature of ObjectStore (from Object Design), an object-oriented database system claiming conformity to the ODMG standard. In short, the `Ref` feature assigns "loose" pointers to objects that are computed at run-time and this seems to support such a mapping that we have proposed in that we avoid "hard" links to the underlying systems (since they wish to maintain autonomy).

The schema integration process we have developed should have a designers graphical tool to further support the use of the constructs we have presented. The goal is to automate the process as much as possible without the schema integrator losing control and understanding of the process.

Appendices

Appendix A

Object Oriented Concepts

A.1 History

The basis of the Object-Oriented(OO) paradigm was developed through the programming language Simula-67 [DMN70]. The starting point of the language was to provide a mean to describe complex systems, which could be simulated. The developers of Simula had a more philosophical approach than language-oriented which is reflected by their system-oriented definition of object-orientation.

The System-definition: In object-oriented programming an information process is regarded as a system developing through transformations in state. The substance of the process is organized as the system components, called objects. A measurable property of the substance is a property of an object. Transformations of state are regarded as actions by objects. A system is a part of the world that is regarded as a whole consisting of components, each component characterized by properties that are selected as being relevant and by actions related to these properties and those of other components.

A.2 The Principles of Object-Orientation

The more common definition, also known as the American perspective, has four principles that a programming language must support to be classified as an object-oriented language. In traditional programming the emphasis is on data-structures and control structures, where as in object-oriented programming it is on objects and messages.

A.2.1 Encapsulation

Data and operations that logically belong together in a meaningful way are encapsulated. Such an encapsulation is often referred to as an object[1] and could e.g. be used to model a real world object and its properties. The encapsulation of the object ensures that the object can hide its information and share only what it finds necessary. The data of the

[1] Thus the term '*Object-Orientation*'

object represents the state of the objects and the operations of it are the messages or services it can compute. Furthermore the encapsulation of the object provides the outside world with entry points or interface to the objects services. However, it is up to the object(or really the creator of the object) how it implements its services.(A real world example of this hidden implementation could be a car. You, the outside user, do not need to know *how* the engine is built to drive the car. The only thing you must know is how to operate its interface, i.e. throttle pedal etc.) This way, the object can change its implementation of its interface as long as the services remain the same (in the car example the manufacturer can change the engine to a totally different one without the user having to care about it too much). We also say that object can send *messages* to each other and that the object that receives the message executes the appropriate *method* according to the message sent.

A.2.2 Classification

Common objects should be classified. Since the outside world can only access an objects interface we classify objects according to their interfaces. Objects are classified into *classes*. A class defines the interface of a common set of objects and describes how these objects will be implemented in terms of their variables and methods. We say that an object belonging to a class is an *instance* of that class and that its variables are *instance variables*. The class is "responsible" for the creation of new objects as it has the base skeleton for new objects.

A.2.3 Inheritance

Classes can be extended into new classes that share the first classes properties, but also adds some new. We say that it *inherits* from the existing class. The class that inherited is said to be the *sub-class* of the *super-class* it was derived from. The sub-class might add new messages to the super-class' and might also override the current definitions of methods, and do something differently. Another possibility is *multiple inheritance* which means that a sub-class can inherit from multiple super-classes.

A.2.4 Dynamic Binding

Dynamic binding, or *late binding*, is when an object decides which method to execute. Consider a message x_1 defined for class languagesX, and that this message is re-implemented in class Y an Z, Y and Z being sub-classes of X. If the sender knows only that the object it is in contact with can respond to the message x_1, it is an instance of class X or a sub-class of X. Since the message x_1 is re-implemented in the sub-classes Y and Z, it has to be decided at run-time which actual method to invoke(the binding is done at a later time). The notion of dynamic binding is also known as *polymorphism* among objects. An example of this could be that we have a class called GEOMETRIC-OBJECT that has two sub-classes, SQUARE and TRIANGLE. The super-class has a method called DRAW that is re-implemented in its two sub-classes. If the sender invokes a call to the method DRAW of

an object in this hierarchy, the correct method for drawing the corresponding figure will be invoked.(see fig.A.1).

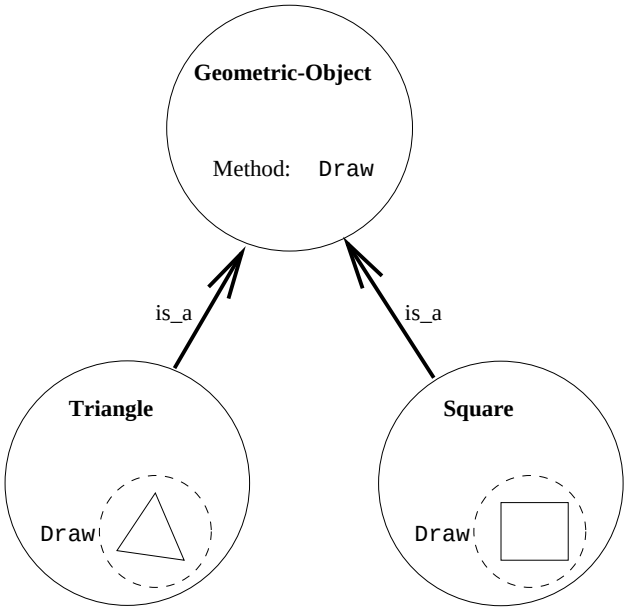

Figure A.1: Dynamic binding - the correct Draw method will be decided at run-time.

A.3 Object-Oriented Programming

As mentioned, the Simula-67 language was early in introducing the object-oriented way of thinking. Later, this paradigm has received a popularity that is still growing and we have seen several new programming languages emerge. Two of the most popular are C++[Str91] and Smalltalk[GR83].

Object-Oriented programming is not just a few new features added to programming . Rather, it is a new way of thinking about the process of decomposing problems and developing solutions. Where traditional programming languages had the emphasis on data-structures and control-structures, the object-oriented languages have emphasis on objects and message passing between them. The objects act like autonomous agents and by the interaction of objects, the computation proceeds. By reducing the interdependency among software components, object-oriented programming permits the development of reusable software systems. Such components can be created and tested as independent units, on isolation from other portions of software application. Reusable software components permit the programmer to deal with problems on a higher level of abstraction. We can define and manipulate objects simply in terms of the messages they understand and a description of the tasks they perform, ignoring implementation details.

Although the object-oriented programming languages support the object-oriented prin-

ciples, it is never the less possible to program the traditional way using most of these languages. It is up to the programmer to alter his way of thinking when developing new software to take advantage of object-orientation.

A.4 Object-Oriented Databases

ODBMSs provide an architecture that is significantly different than other DBMSs. A summary definition of ODBMS could be:

ODBMS is a DBMS that integrates database capabilities with object-oriented programming language capabilities.

Rather than providing only a high-level language such as SQL for data manipulation, an ODBMS transparently integrates database capability with the application programming language. The advantages of this are e.g.:

- One doesn't have to use a separate DML[2], it lies within the programming language.
- The DBMS no longer has to copy and translate data between data and programming language representations(see fig.A.2). This is a good performance advantage.

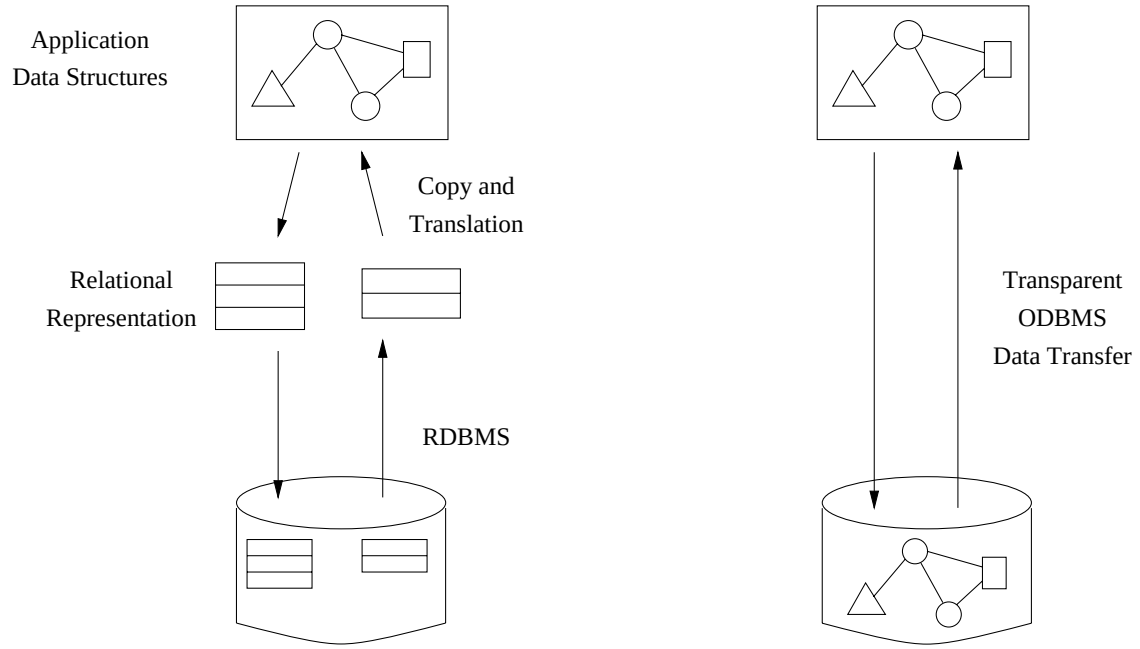

Figure A.2: Comparison of RDBMS and ODBMS architectures

[2]Data Manipulation Language

A.5 What are the Benefits of OO?

There are several benefits of adapting OO. The following benefits, although subjective, are considered by many to be good reasons for adopting OO.

- OO modeling reflects reality better than traditional modeling
- The model is more stable than functionality
- Subclassing and virtuals improve the reusability of code

Appendix B

ODMG-93 - The Object Database Standard

In this appendix we will describe the ODMG-93 [Cat94] object database standard. This is a standard proposed by the Object Database Management Group(ODMG). The ODMG is a consortium of object-oriented database management system (ODBMS) vendors and interested parties working on standards to allow portability of customer software across ODBMS products.

B.1 Introduction

The object-oriented view in computer science has become more and more popular the later years.[1] *Simula-67* [DMN70] was an early object oriented language developed at The Norwegian Computing Center(NR)[2] in Oslo. The most widespread object-oriented language however is C++ [Str91]. Following the program languages that have complied to this paradigm are the database systems. The relational model has had success for some time now, but here we also see that the object-oriented thinking has entered the scene. Several vendors have developed object-oriented database management systems. But a group of vendors saw that the importance of a standard to ensure portability and endorsement of the approach was crucial to meet the customers requirements. This group founded ODMG and have been working on the object database standard. This work is an ongoing work and in 1996 the group will release their next version of their work with the latest extensions.

B.2 Goals

The ODMG group has a primary goal to define a standard that allows an ODBMS user to write portable applications, i.e. applications that could run on more than one standard

[1] See Appendix A for an introduction to the object-oriented way of mind
[2] Norsk Regnesentral

compliant ODBMS product. It is also a hope that the standard proposal will be helpful in allowing interoperability between the ODBMS products. In the context of this thesis, this could e.g. be used for heterogeneous distributed databases communicating through the OMG[3] Common Object Request Broker[OMG92].

An important goal is also to try to bring programming languages and database systems to a new level of integration. Using the relational model in database systems has shown to have a mismatch between the application language and the database systems internal representations. The object model described here has had this in thought and it is defined closer to the programmers application language(object-oriented), making it possible for an ODBMS to transparently integrate database capabilities with the programming language.

All the participating member companies are committed to support the standard, thus they hope this proposal will become a de facto standard for the industry. The participating vendors have already released products that are compliant with the standard. Two of these are ObjectStore OODBMS from Object Design, and Versant OODBMS from Versant.

B.3 Architecture

The architecture of the ODMG-93 standard has four major components:

1. Object Model
2. Object Definition Language
3. Object Query Language
4. Language Binding

These components will be described closer in the following sections.

B.3.1 Object Model

The common data model has used the OMG Object Model [Sol90] as a basis. Components have been added to support the intended needs of the ODMG group.

The Object Model is simply summarized as:

- The basic modeling primitive is the *object*.
- Objects that exhibit common behavior and have a common range of states are categorized into *types* or *object types*.
- The behavior of objects is defined by a set of *operations* or *messages* that can be executed on an object of the type[4].
- An object has a set of properties that can be either *attributes* of the object itself or *relationships* between the object and one or more objects. The state of an object is defined by the value it has for its properties.

[3] Object Management Group

[4] E.g. you can "draw" an object of type Circle

B.3.1.1 Types and Instances

A type has one *interface* and one or more *implementations*. The interface of the type defines the external interface supported by all the instances of the type. An implementation defines *data structures* in terms of which instances of the type are physically represented and the *methods* that operate on those data structures to support the externally visible state and behavior defined in the interface.

B.3.1.1.1 Inheritance Types may be organized into a hierarchy of subtypes and supertypes. The subtype inherits all of the characteristics(properties and operations) of its supertype. In addition it can define its own characteristics that apply only to its instances(or subtypes). This way a subtype can be treated as an instance of its supertype because it has its characteristics, but not vice versa, thus the subtype supports all the state and behavior of the supertype as well as new state and/or behavior unique to its more specialized nature.

Some types are termed *abstract* which means that they do not define an implementation and therefore can not be instantiated. They must be subtyped and their subtypes must define an implementation for the inherited characteristics.

B.3.1.1.2 Extent The set of all instances of a type is called the *extent* of the type. There is a direct correspondence between the intentional notion *type* and the extensional notion *extent*. If an object is an instance of type A, then it will automatically be a member of the extent of A. In a similar way the extent of B will be a subset of the extent of A if B is a subclass of A.

B.3.1.1.3 Implementations and Classes A type has one or more implementations. An implementation of an object type consists of a *representation* and a set of *methods*. The representation is a set of data structures and the methods are procedure bodies. The methods implement the external operations, but there may also be internal methods that have no associated operation.

Implementations are named uniquely within the scope defined by a type.

The combination of the type interface specification and one of the implementations defined for the type is called a *class*.

In comparison with e.g. the C++ definition of a class the ODMG model is richer in that it allows multiple implementations for a given interface. Which implementation an object uses is specified at object creation time and it is not possible to dynamically change the implementation of an object at a later time.

B.3.1.2 Objects

Objects have state and behavior and also identity. Their identity is intrinsic in and of themselves and not based by the objects characteristics.

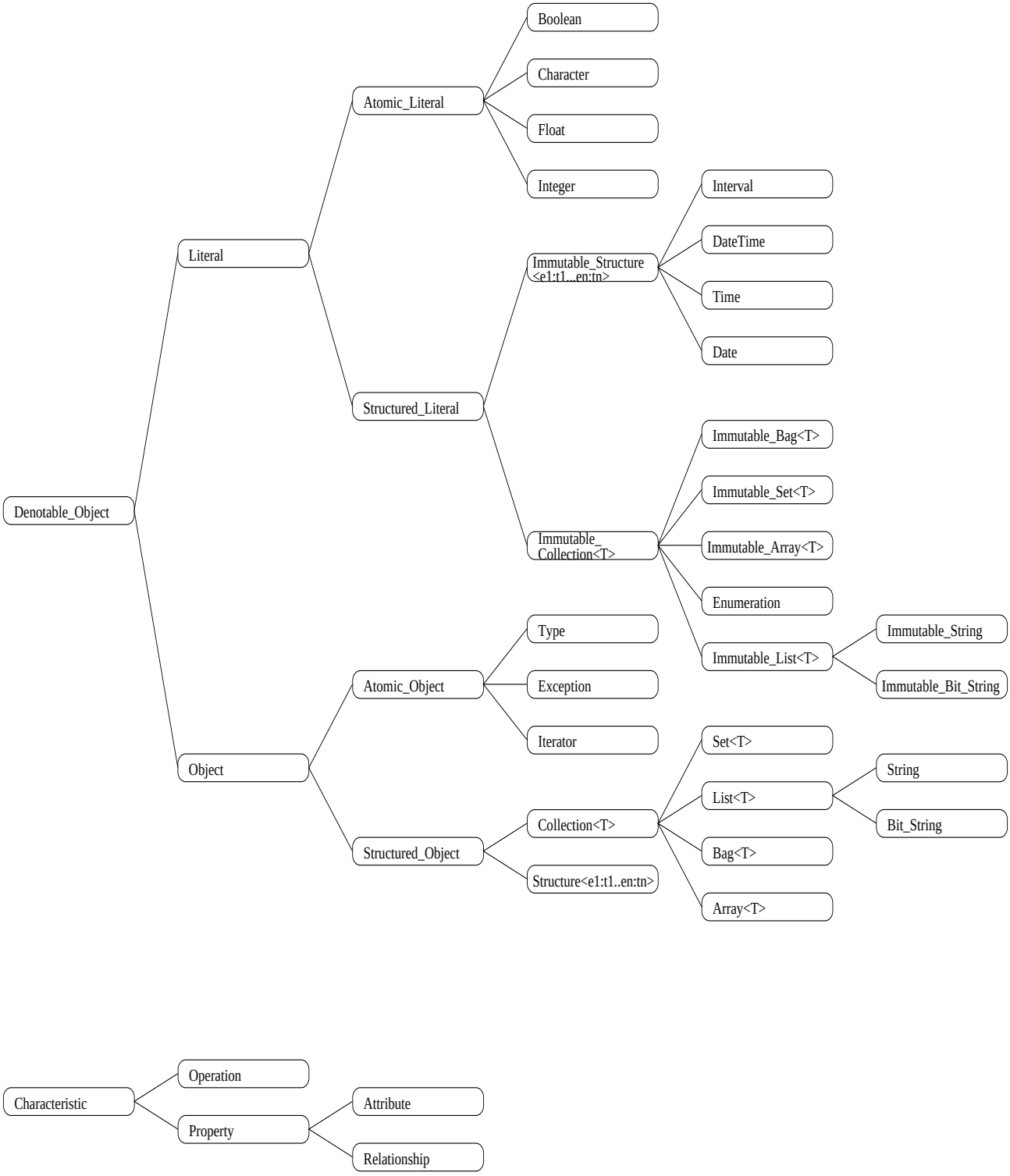

Figure B.1: The full type hierarchy

B.3.1.2.1 The Denotable_Object The hierarchy of object types is rooted at the type Denotable_Object(see fig.B.1). As we can see the Denotable_Object is decomposed into two categories, Object(mutable) and Literal(immutable) which are further decomposed into Atomic and Structured branches. These first to decompositions represent the two orthogonal lines over which Denotable_Object can be decomposed.

B.3.1.2.2 Type Object All denotable object have a unique identity however the internal representation of these differs between the objects and the literals. The literals are typically identified by their bit pattern, but the object representation is referred to what we call *object identifier*, or OID for abbreviation. The OID is a specially constructed bit pattern generated only for the purpose of uniquely identifying a particular object(the actual structure of the bit pattern is not defined by the Object Model – this is considered a representation issue). The OIDs remain unchanged over an objects lifetime. Individual objects may however be given names meaningful to the programmer. A name must refer uniquely to a single object within the scope of the name[5].

An object may be defined as a subtype of one or more other types. If objects type B is declared to be a subtype of A, then any operations defined on A are also available on instances of B, all attributes defined on A are also defined on B, and any relationships defined on A are also available on instances of B.

The following built-in properties are defined on type Object and thus inherited to all subclasses:

- **has_name?**:Boolean
- **names**:Set<String>
- **type**:Type

The following built-in operations are defined on type Object:

- **delete**()
- **same_as?**(oid: Object_id) → b:Boolean

There is also an explicit **create** operation that creates an object, assign an Object_id and returns the id as the value of the operation. The **delete** operation removes the object from the database and thereby removing it from any relationships in which it participated. This does not mean that it recursively deletes any other objects related to it. The ODBMS may however be responsible of removing the object from the maintained extensions. The Object_id is not reused.

[5]From the application programs point of view, the database adds a new out-most scope to those defined in the program.

B.3.1.2.3 Literals Literals are objects whose instances are immutable. The hierarchy(see fig.B.1 defines two subtypes of literals:

1. Atomic_Literals – e.g. numbers and characters
2. Structured_Literals – e.g. date and time

There is no explicit `create` operation defined on atomic literals; they implicitly pre-exist. It follows that they do not have unique OIDs, but nevertheless have unique identity. The structured literals are further decomposed into two sub-categories:

1. Immutable_Collection
2. Immutable_Structure

They are analogous to their counterpart Structured_Object types, Structure and Collection, but are immutable.

B.3.1.3 Modeling State – Properties

An object type defines a set of *properties*. Two kinds of property are defined in the model:

1. Attribute
2. Relationship

They are described in the following.

B.3.1.3.1 Attributes Attributes are defined on a single object type and take literals as their values. They do not have OIDs. An attribute takes as its value a literal or a set of literals.

The following built-in operations are defined on attributes:

- **set_value**(new_value:Literal)
  The **set_value**() operation gives the attribute a new value, replacing whatever value it currently has.
- **get_value**() → existing_value:Literal
  The **get_value**() operation will return the literal supplied by the argument to the previous **set_value**(). If it was not set, the return value will be the default value if one was set at object type definition, otherwise nil.

Attributes define abstract state. They therefore appear within the interface definition of an object rather than in the implementation. It is not necessary that the attribute is implemented as part of a data structure. E.g. a call to a **get_value**() operation on an age attribute of a type Person could be implemented as a method deriving the person's age from a date_of_birth attribute.

Attributes can not be added subtype specific operations nor participate in relationships. However it is possible to override the **set_value**() and **get_value**() operations

allowing the type definer to have better control over attribute settings and access, e.g. by doing constraint evaluation on the invocation of a **set_value**() operation.

The programmer will seldom see or need the **set_value**() or the **get_value**() operations directly. They will rather be woven in to the programmers normal environment as assignments(object.att=literal) or default value settings(new Person(33) – 33 being an initial age setting). The Preprocessor or compiler will further translate this syntax to the appropriate Object Model operations.

B.3.1.3.2 Relationships are defined between mutable object types. The base model doesn't support n-ary relations, only binary relations. But it does support one-to-one, one-to-many and many-to-many relationships.

The relationships themselves have no names, instead named *traversal paths* are defined for each direction of traversal. An example of a many-to-many relationship could be that a student *takes* a course, conversely, a course *is_taken_by* a set of students. Each name is defined within the interface definitions of the respective object types that participate in the relationship. To tie the relationships defined in the two objects together we indicate that they are *inverses* of each other(see fig.)

```
interface Student
   { ...
      takes: Set<Course> inverse Course::is_taken_by
   }

and

interface Course
   { ...
      is_taken_by: Set<Student> inverse Student::takes
   }
```

Figure B.2: Interface definitions of the relationships between Student and Course

Relationships maintain referential integrity; if an object that participated in relationship is deleted, a subsequent attempt to traverse the relationship will raise an exception. Also relationships do not have OIDs, they are uniquely identified by the object instances that participate in them.

B.3.1.4 Modeling Behavior – Operations

Instances of an object type have a defined behavior and it is specified as a set of *operations*. For each operation, an *operation signature* is included in the object type definition by the type programmer. The signature includes the argument names and types, exceptions

potentially raised, and types of the values returned, if any. Operations are always defined on a single object type, never on two or more object types nor are they ever defined independently of an object type.

Operations are only uniquely defined within a single type definition which means that operations defined on different types may have the same name. This raises some problems in schema integration as we have mentioned in chapter 4(homonyms).

We have the following built-in operation on type Operation:

- **invoke**()
- **return**()
- **return_abnormally**(e:Exception)

These operations can usually not be directly invoked by the programmer. The occurrence of an operation name within a statement of the programming language is instead compiled into the code which invokes the named operation.

The object model supports exception handling with the root type Exception provided by the ODBMS. It includes an operation to print out a message noting that an unhandled exception of some type has occurred and to terminate the process. The root type can be subtyped into a supertype/subtype hierarchy.

B.3.1.5 Structured Objects

As we can see from the full type hierarchy(fig.B.1), the Structured_Object has two subtypes – Structure and Collection.

Structures have a fixed number of named slots of which contains an object or a literal and these slots can be referred to directly to modify them, e.g. address.zip_code.

Collections, on the other hand, contain an arbitrary number of elements. They do not have named slots and their elements are all of the same type, which is not a requirement for Structures.

B.3.1.5.1 Collections A collection is an object that groups other objects. They may be defined over any instantiable subtype of type Denotable_Object. Individual collections are instances of collection types, collection types are instances of collection type generators, also called parameterized types. A parameterized type can be instantiated to generate a new type, e.g. the parameterized Stack<T> can be instantiated to produce Stack<Customer> by supplying it with the element type Customer. The type checking of parameterized types is done at runtime.

Each collection has an immutable identity, just like any other object. This means that one can insert, delete or modify an element in a collection and it will still be identified as the same collection. Also, two collections having the same elements are not the same collection.

Insertion into collections is based on one of two alternatives:

1. Absolute position within the collection, at the beginning or the end.

2. Point established by a *cursor*

Retrieval is based on one of three alternatives:

1. Absolute position(as in insertion)

2. Current cursor-relative position(as in insertion)

3. A predicate that uniquely selects an element from the collection based on the value(s) the sought object carries for one or more of its properties.

The object model supports both ordered and unordered collections, where the order is defined either by the sequence in which objects are inserted or by the value of one of the properties of the objects that are members of the collection. The same object may be allowed to be present in the collection more than once(bag) or it may not be allowed.

Iteration over the elements in a collection is done by defining an *iterator* or *cursor* that maintains a current position within the collection to be traversed. The type Iterator has four basic operations:

- `first()`
- `last()`
- `next()`
- `more?()`

that can be used to step through the elements.

Collection<T> is an abstract type and can not be instantiated. It has a number of properties and operations that are inherited by its subtypes. The object model defines a standard set of built-in type generators:

Set<T> Sets are unordered collections and do not allow duplicates.Its defined operations are common set operations, e.g. union(), intersection() and is_subset().

Bag<T> Bags are unordered collections that allow duplicates. Bag<T> defines the following operations in addition to its inherited ones; union(), intersection() and difference().

List<T> Lists are ordered collections that allow duplicates. The order is based on the order of their insertion. It defines list specific operations such as insert_element_after(), remove_last_element and retrieve_last_element().

Array<T> Arrays are dimensional arrays of varying length. Its initial size is specified at creation time, but it can be changed both implicitly(by inserting beyond the current end) and explicitly(by the resize() operation).

B.3.1.5.2 Structures A structure is an unnamed group of elements. Each element is a (name, value) pair, where the value may be any subtype of type Denotable_Object and thereby also other structures as members of its elements. Since they are immutable they remain unchanged after their creation and since they are literals they do not have OIDs.

The operations defined on $Structure < e_1 : T_1, ...e_n : T_n >$ are:

- create([<initializer-list]) → s:Structure
- delete()
- get_element_value(element) → value: Denotable_Object
- set_element_value(element, value:Denotable_Object)
- clear_element_value(element)
- clear_all_values()
- copy() → s: Structure

B.3.1.5.3 Structured Literals Structured literals have two subtypes:

1. Immutable_Collection
2. Immutable_Structure

parallel to the structured object.

The built-in subtypes of Immutable_Collection mirror those of Collection and are:

- Immutable_Set
- Immutable_Bag
- Immutable_List
- Immutable_Array

The immutable collections behave just like their mutable counterparts apart from that they can not be modified. Immutable sets are the basis for the extensional treatment of sets that is common in mathematical logic.

There are no defined subtypes of Immutable_Structure. Immutable structures may be used to capture update constraints on the values of a property and are often returned as the results of queries — cutting out the interesting parts of objects rather than having to walk through the object for interesting parts afterwards.

B.3.1.6 Transactions

Persistent data is data that survives the process that creates it. Programs that use persistent data are organized into *transactions*. Transactions are data referrals or modifications that have the three following properties:

1. **Atomicity** – which means that the transaction either happens as a whole or not at all. If the transactions succeeds(commits), the transactions changes are permanent and visible to other user of the database. If it aborts, than the database is unchanged as if the transaction never happened.

2. **Consistency** – which means that database users will always see the database in a consistent state. A transaction will bring the database from one consistent state to another.

3. **Integrity** – which means that committed transaction are ensured to never be lost, surviving process abortions and operation system failure.

The object model supports nested transactions like in fig.B.3.

```
Transaction::begin() → t:Transaction
...
   Transaction::begin() → a:Transaction
   ...
      Transaction::begin() → b:Transaction
      ...
      b.commit()
   ...
   a.commit()
...
t.commit()
```

Figure B.3: Nested Transactions

In this scheme, if t aborts, then changes made by x and y will be aborted, whether or not they had already committed. The commit of a nested transaction is only relative to the commit of its containing parent transaction. If a nested transaction aborts, this does not cause abort of the containing transaction.

B.3.1.7 Type Database

A database provides storage for persistent objects of a given set of types. Each database has a *schema*, which consists of a set type definitions. The database may contain instances

of the types defined in its schema. Each database in an instance of type Database. The type Database[6] has the following operations:

- open()
- close()
- contains_object?(oid:Object) → b:Boolean
- lookup_object(oid:Object) → b:Boolean

The names of the types in the schema and their associated extents are global to the database, and become accessible to a program once it has opened the database. A database may also contain named objects, often called "root objects", that can be referenced by a program. Type names, extent names and root object names are the three kinds of global names that serve as entry points into the database allowing the programmer to do initial navigation from.

B.3.2 Object Definition Language

The Object Definition Language (ODL) is a specification language to define the interfaces to object types that conform to the ODMG Object Model. The ODMG group has had a primary objective with the ODL to facilitate portability of database schemas across conforming ODBMSs. ODL is not intended to be a full programming language nor is it meant to be programming-language dependent. It is a specification language for interface signatures. It defines the characteristics for types, including their properties and operations, but is does not address the definition of the methods that implement those operations. Further, ODL provides a context for integrating schemas from multiple sources and applications. These source schemas may have been defined with any number of object models and data definition languages, and they may all be translated to ODL as a common basis (see fig.B.4). This common model then allows the various models to be integrated with common semantics. An ODL specification can be realized concretely in an object programming language like C++ or Smalltalk (see. B.3.4 and fig.B.4).

B.3.2.1 Specification

A type is defined by defining its interface in ODL. In a type definition the characteristics of the type itself appear first followed by the definitions of the properties and operations of the types interface. The top-level BNF[7] for ODL is described in fig.B.5. Any list may be omitted if it is not applicable for the types interface.

[6] The type `Database` is really a proposed type which is meant to be included in the next version of the ODMG Database Standard

[7] Bachus Naur Format

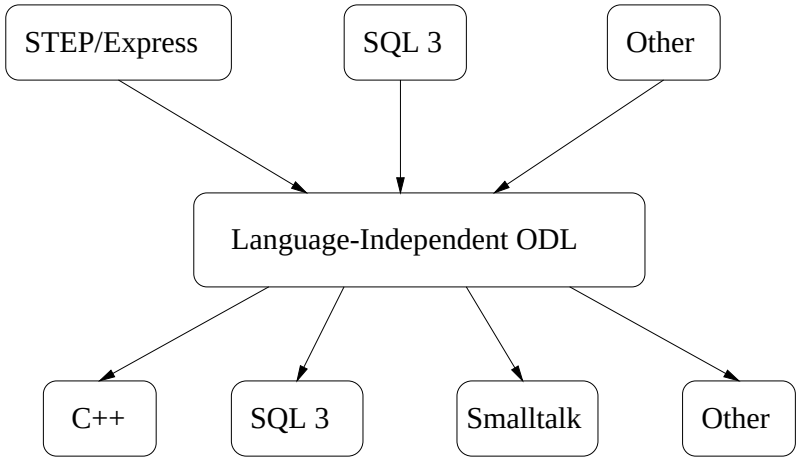

Figure B.4: Mapping from other models to ODL, and from ODL to other languages

```
<type definition>  ::= interface <type_name>[:<supertype_list>]
                       {
                       [<type_property_list>]
                       [<property_list>]
                       [<operation_list>]
                       };
```

Figure B.5: Top-level BNF for ODL

B.3.2.2 Type characteristics

Type characteristics are the characteristics that apply to the type itself, and not directly to its instances. From the top-level BNF for types the type characteristics are <type_name>, <supertype_list> and <type_property_list>. The BNF for these are described in fig.B.6.

```
<type_name>           ::= <string>
<supertype_list>      ::= <supertype> | <supertype>, <supertype_list>
<supertype>           ::= <type_name>
<type_property_list>  ::= <type_property>;
                          | <type_property><type_property_list>
<type_property>       ::= extent <extent_name> | key[s] <key_list>
<extent_name>         ::= <string>
<key_list>            ::= <key_spec> | <key_spec>, <key_list>
<key_spec>            ::= <property_name> | (<property_list>)
<property_list>       ::= <property_name> | <property_name>, <property_list>
<property_name>       ::= <attribute_name> | <traversal_path_name>
<attribute_name>      ::= <string>
<traversal_path_name>::= <string>
```

Figure B.6: BNF for type characteristics

Each supertype must be specified in its own type definition. The supertype, extent and key definitions may appear in any order in the type property list and furthermore there should not be more than one extent or key definition.

B.3.2.3 Instance Properties

The type's instance properties are the attributes and relationships of its instances. These properties are specified in attribute and relationship specifications. the BNF for the instance properties are described in fig.B.7

B.3.2.4 Operations

ODL is compatible with IDL[8] for specification of operations. The BNF for the <operation_list> is described in fig.B.8

B.3.3 Object Query Language

The object query language (OQL) for the ODMG data model will be described in the following. The ODMG group designed the OQL with the following principles and as-

[8]IDL is the Interface Definition Language from the OMG core Object Model

```
<property_list>    ::= <property_spec>; | <property_spec><property_list>
<property_spec>    ::= <attribute_spec> | <relationship_spec>

<attribute_spec>   ::= [attribute]<domain_type>[[<size>]]<attribute_name>
<domain_type>      ::= <atomic_literal> | <structured_literal> |
                       <collection of objects or literal>
<size>             ::= <integer>

<relationship_spec>::= [relationship]<target_of_path><traversal_path_name_1>
                       inverse <inverse_traversal_path>
                       [{order_by<attribute_list>}]
<traversal_path_name_1> ::= <string>
<target_of_path>   ::= <target_type> | <collection_type><target_type>
<target_type>      ::= <target_name>
<inverse_traversal_path> ::= <target_type> :: <traversal_path_name_2>
<traversal_path_name_2>  ::= <string>
<attribute_list>   ::= <attribute_name> | <attribute_name>, <attribute_list>
```

Figure B.7: BNF for instance properties

```
<operation_list>    ::= <operation_spec>; | <operation_spec>, <operation_list>
<operation_spec>    ::= <return_type><operation_name>
                        ([<argument_list>])[<exceptions_raised>]
<return_type>       ::= <type_name>
<operation_name>    ::= <string>
<argument_list>     ::= <argument> | <argument>, <argument_list>
<argument>          ::= <role> [<argument_name>:]<argument_type>
<role>              ::=  in | out | inout
<exceptions_raised> ::= raises(<exception_list>)
<exception_list>    ::= <exception> | <exception>,  <exception_list>
<exception>         ::= [[...]]
```

Figure B.8: BNF for operation specification

sumptions:

- OQL is not computationally complete. It is a query language which provides easy access to an object database.
- OQL provides declarative access to objects.
- OQL relies on the ODMG object model.
- OQL has an abstract syntax.
- The formal semantics of OQL can easily be defined.
- OQL has one concrete syntax which is SQL-like, but it is easy to change the concrete syntax. Other concrete syntaxes are defined for merging the query language into programming languages (e.g. a syntax for preprocessed C++ and a syntax for Smalltalk)
- OQL provides high-level primitives to deal with sets of objects but does not restrict its attention to this collection construct. Thus, it also provides primitives to deal with structures and lists, and treats all such constructs with the same efficiency.
- OQL does not provide explicit update operators but relies on operations defined on objects for that purpose.
- OQL can be easily optimized by virtue of its declarative nature.

OQL can be a stand alone language or it can be embedded into a programming language. The query language supports both types of objects, mutable and literals, depending on the way these objects are constructed or selected.

Creating objects with an identity is achieved by using a type name constructor as in:

```
Person(name:"Peter", birthdate:"3/28/56", salary:100000)
```

Here we have initialized certain properties of the object. The object can, however, have additional properties which are given default values.

A literal might be created using the literals name in a similar way:

```
struct(a: 10, b:"Peter")
```

creating a structure with two valued fields.

When using OQL embedded in a programming language, objects are created with the constructs of this (extended) language.

An extraction expression may return a number of different object types depending on its nature:

- A collection of objects with identity, e.g. `select x from x in Persons where x.name="Peter"` returns a collection of persons whose name is Peter.

- An object with identity,
  e.g. `element(select x from x in Persons where x.passport_number=1234567)`
  returns the person whose passport number is 1234567.

- A collection of literals, e.g. `select x.passport_number from x in Persons where x.name="Peter"` returns a collection of integers giving the passport numbers of people named "Peter".

- A literal, e.g. `Chairman.salary`

Therefore the result of a query is an object with or without object identity: some objects are generated by the query language interpreter, and others are produced from the current database.

B.3.4 Programming Language Bindings

The standard describes language bindings for both C++ and Smalltalk. The programming language-specific bindings for ODL/OML for C++ and Smalltalk are based on one basic principle: The programmer should feel that there is one language, not two separate languages with arbitrary boundaries between them.

B.3.4.1 C++ binding

The most important programming language for ODBMSs has proven to be C++. The C++ binding of ODL is expressed as a class library and an extension to the standard C++ class definition grammar. The class library provides classes and functions to implement the concepts defined in the ODMG object model.

The C++ to ODBMS language binding approach described by this standard is based on the smart pointer or "Ref-based" approach. In the Ref-based approach, the C++ binding maps the Object Model into C++ by introducing a set of classes that can have both persistent and transient instances. These classes are distinct from the normal classes defined by the C++ language, all of whose instances are transient. For each database class `X`, an ancillary `class Ref<X>` is automatically defined by the ODL preprocessor. Instances of database classes are then referenced using parameterized references, e.g.:

1. `Ref<Professor> profP;`
   (Comment: declares the object `profP` as an instance of the automatically defined type `Ref<Professor>`)

2. `Ref<Department> deptRef;`
   (Comment: declares deptRef as an instance of the automatically defined type `Ref<Department>`)

3. `profP → grant_tenure();`
   (Comment: invokes the `grant_tenure()` operation defined on class `Professor`, on the instance of that class referred to by `profP`)

4. `deptRef = profP → dept`
   (Comment: assigns the value of the `dept` attribute of the professor referenced by `profP` to the variable `deptRef`)

B.3.4.2 Smalltalk Binding

Smalltalk Images provide a form of object persistence, but are not the same as databases. Smalltalk implements its own memory management and expects all Smalltalk objects to exist within its object space. A Smalltalk object can not refer to memory outside this space via a direct pointer. Thus Smalltalk cannot directly reference objects within an ODBMS cache. This means that in all likelihood an ODMG Smalltalk binding will be implemented through external procedures.

The ODMG Smalltalk binding is based on the Smalltalk Object and Class instance protocols, along with new classes DatabaseGlobals and Session.

The Smalltalk binding for ODL has a syntactic style that is consistent with the declarative aspects of the Smalltalk language. Instances of these classes can be manipulated using Smalltalk and the Smalltalk OML. Figure B.9 shows a simple object type declaration including property type declarations and operation type declarations. As Smalltalk is a dynamic language, operations need not be specified at object type declaration time.

```
Object subclass:'Professor'
instVarNames:#('age','name','salary','universityId','dept','advisees')
classVars:#()
poolDictionaries:#()
inDictionary:ADictionary
constraints:#(#('age',SmallInteger),
   #('name',String),
   #('salary',Money),
   #('universityId',Integer),
   #('dept',Department,'inverse','professors'),
   #('advisees',StudentSet,',inverse','advisor','orderedBy','studentId'))
```

Figure B.9: Smalltalk Sample Object Type Declaration

We use the Smalltalk class definition facilities directly. The `constraints:argument` array contains type definitions for implementations of both attributes and relationships. The class compiler detects these constraint types and generates appropriate methods to support the attribute and relationship semantics.

The only types that can be embedded as objects within a class are `Char` and `SmallInteger`. All other types are treated be Smalltalk as first-class objects.

B.4 Status

Currently(spring-96) the ODMG group have released the 1.2 version and are expecting to release version 2.0 soon.

Appendix C

ODL-M Syntax

The syntax of the constructs introduced in chapter 6 is described in the following.

C.1 ODL-M Keywords

The following are the ODL-M keywords:

ALIAS	AS	BEGIN	BOOLEAN
BUILD	BY	CASE	COPY
ELSE	END	END_BUILD	END_CASE
END_IF	END_MAP	END_PRUNE	END_SCHEMA_MAP
ENUMERATION	IF	INTEGER	MAP
PRUNE	SCHEMA_MAP	STRING	THEN
WHERE			

C.2 Schema Map

```
<sche_map_decl>      ::= <sche_map_head> <sche_map_body> END_SCHEMA_MAP;
<sche_map_head>      ::= SCHEMA_MAP <schema_id> {',' <schema_id>} '<-'
                         <schema_id> {',' <schema_id>}
<sche_map_body>      ::= {<interface_spec>} {<global_decl>} {<instansiate_clause>}
                         {<sche_map_component>}
<sche_map_component> ::= <map_decl> | <external_function_ref> | <no_map_decl> |
                         <external_call_decl> | <function_decl> | <procedure_decl>
```

C.3 Object Type Interface Map

```
<map_decl>           ::= <map_head> <map_body> END_MAP;
<map_head>           ::= MAP <target_group> '<-' <source>;
<target_group>       ::= <target_factor> {<target_operator> <target_factor>
```

```
<source>              ::= <qualified_type> {AND <qualified_type>}
<map_body>            ::= [<prune_clause>] {<map_body_component>}
<map_body_component> ::= <statement> | <attribute_map> | <local_decl>

<target_factor>       ::= <qualified_type> | <oneof_factor> | <optional_factor>
<target_operator>     ::= AND | OR | ',' | ANDOR
<oneof_factor>        ::= ONEOF '(' <target_group> ')'
<optional_factor>        ::= OPTIONAL '(' <target_group> ')'

<attribute_map>       ::= <qualified_attribute> ':-'
                             [<cast>] (<qualified_attribute | <simple_expression>)
```

C.4 Build

```
<build_decl>       ::= <build_head> <where_clause> <build_body> END_BODY;
<build_head>       ::= BUILD <entity_id> {<build_operator> <entity_id>} '<-'
                         <entity_id> ',' <entity_id>;
<build_operator>   ::= ',' | AND
<build_body>       ::= <map_body>
```

C.5 Copy

```
<copy_decl>        ::= COPY <copy_target> '<-' <copy_source>;
<copy_source>      ::= (<entity_id> {AND <entity_id>}) |
                                  (<function_id> '(' <entity_id> ')' )
<copy_target>      ::= <entity_id> {AND <entity_id>}
```

C.6 Object Type Instantiation

```
<instance_clause>  ::= <instance_id> '=' (<simple_instance> |
                                           <complex_instance>) ';'
<simple_instance>  ::= <entity_id> '(' <attribute_instance>
                                        {',' <attribute_instance> } ')'
<complex_instance> ::= '(' <simple_instance> { <simple_instance> } ')'
<instance_id>      ::= '#' <simple_id>
<attribute_instance> ::= <instance_id> | ''' <string_literal> ''' |
                         <integer_literal> | <float_literal> | <hex_literal> |
                         '.' <logical_literal> '.' | '.' <boolean_literal> '.' |
                         '.' <enumeration_id> '.' | <aggregate_instance> | '$'
<aggregate_instance> ::= '(' <attribute_instance> {','<attribute_instance>}')'
```

C.7 Object Type Pruning

```
<prune_clause>    ::= PRUNE <prune_list> ';' [<objtype_list>]
                                          [<attribute_list>] END_PRUNE ';'
<prune_list>      ::= <prune_id> [',' <prune_id>] |
                      ['('] <prune_element> <prune_op> <prune_element> [')']
<prune_element>   ::= <prune_list>
<prune_id>        ::= <simple_id>
<prune_op>        ::= AND | OR | XOR
<objtype_list>    ::= <objtype_id> {',' <objtype_id> } ';'
<attribute_list>  ::= <attribute_id> {'.' <attribute_id> } ';'
```

C.8 Type Mapping

```
<type_map_decl>   ::= <type_map_head> <type_map_body> END_MAP_TYPE ';'
<type_map_head>   ::= MAP_TYPE <type_id> '=' <type_id> | <simple_expression> ';'
<type_map_body>   ::= <defined_type_map_body> | <enumeration_type_map_body>

<defined_type_map_body>     ::= {<type_id> [<bound_spec>] '='
                                ([<bound_spec>]) | <simple_expression> ';'}

<enumeration_type_map_body> ::= <enumeration_map> {<enumeration_map>}
<enumeration_map> ::= <enumeration_id> ':=' <enumeration_id>
                                          {',' <enumeration_id> } ';'
```

Appendix D

ODL-Description of the Case

Schema 1:

```
interface Under_Grad{
   extent under_grads;
   key ssn;

   string name;
   integer ssn;
   string major;
   string address
   }

interface Restricted_Course{
   extent restricted_courses;
   key cno;

   string cname;
   integer cno;
   string major;
   }

interface Employee{
   extent employees;
   key ssn;

   string name;
   integer ssn;
   string position;
   }
```

```
interface Faculty{
   extent faculties;
   key ssn;

   string name;
   integer ssn;
   string dept;
   string rank;
   }

interface Enroll{
   extent enrolls;
   keys cno, fssn, sssn;

   string cno;
   integer fssn;
   integer sssn;
   float grade;
   }

interface Emp_Other{
   extent emp_others;
   key ssn;

   integer ssn;
   integer age;
   integer wt_in_lb;
   integer ht_in_in;
   integer salary;
   float   bonus;
   integer tax;
   integer bracket;
   }
```

```
interface Course{
   extent courses;
   key cno;

   string cname;
   integer cno;
   }
```

Schema 2:

```
interface Grad_Student{
   extent grad_students;
   key sssn;

   string sname;
   integer sssn;
   string major;
   float gpa;
   string fname;
   integer fssn;
   string frank;
   string thesis_title;
   }
```

Schema 3:

```
interface Student{
   extent students;
   key ssn;

   string lastname;
   string firstname;
   integer ssn;
   string type;
   string major;
   }

interface Address{
   extent addresses;
   keys ssn,street,city,zip;

   integer ssn;
   string street;
   string city;
   string zip;
   }

interface Enroll{
   extent enrolls;
   keys cno, fac_ssn, stud_ssn;

   string cno;
   integer fac_ssn;
   integer stud_ssn;
   float grade;
   }
```

```
interface Graduate_Info{
   extent graduate_infos;
   key ssn;

   integer ssn;
   integer advisor_ssn;
   }

interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   }

interface Thesis{
   extent theses;
   keys title, ssn;

   string title;
   integer ssn;
   float grade
   }
```

```
interface Faculty{
   extent faculties;
   key ssn;

   string lastname;
   string firstname;
   integer ssn;
   string dept;
   string rank;
   }

interface Course_Restriction{
   extent course_restrictions;
   key cno;

   string cno;
   string major;
   string prereq_cno;
   }

interface Employee{
   extent employees;
   key ssn;

   string name;
   integer ssn;
   string position;

   }
```

```
interface Emp_Personal{
   extent emp_personals;
   key ssn;

   integer ssn;
   integer age;
   integer wt_in_kg;
   integer ht_in_cm;
   }
```

```
interface Emp_Tax{
   extent emp_taxes;
   key ssn;

   integer ssn;
   integer salary;
   float bonus;
   integer tax;
   string bracket;

   }
```

Schema 4:

```
interface Student{
   extent students;
   key ssn;

   string name;
   integer ssn;
   string major;
   float gpa();
   }
```

```
interface Gradstudent:Student{
   extent gradstudents;
   key ssn;

   Set<Faculty> advisor;
   }
```

```
interface Employee{
   extent employees;
   key ssn;

   string name;
   integer ssn;
   string position;
   Employee supervisor;
   }
```

```
interface Faculty:Employee{
   extent faculties;
   key ssn;

   string dept;
   string rank;
   }
```

```
interface Admfaculty:Faculty{
   extent admfaculties;
   key ssn;

   string position;
   }
```

```
interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   Set<Course> prereq;
   }
```

```
interface Enroll{
   extent enrolls;
   keys course, fssn, sssn;

   Course course
   integer fssn;
   integer sssn;
   float grade;
   }
```

```
interface Thesis{
   extent theses;
   keys title, author;

   string title;
   Gradstudent author;
   string status;
   }
```

Schema 5:

```
interface Student{
   extent students;
   key ssn;

   string fname;
   string lname;
   string ssn;
   string major;
   float gpa();
   }
```

```
interface Gradstudent:Student{
   extent gradstudents;
   key ssn;

   Faculty advisor;
   Set<Faculty> committee;
   }
```

```
interface Employee{
   extent employees;
   key ssn;

   string name;
   string ssn;
   Employee supervisor;
   }
```

```
interface Faculty:Employee{
   extent faculties;
   key ssn;

   Department dept;
   string rank;
   }
```

```
interface Course{
   extent courses;
   key cno;

   string cname;
   string cno;
   Set<Course> prereq;
   }
```

```
interface Enroll{
   extent enrolls;
   keys course, fssn, sssn;

   Course course;
   string fssn;
   string sssn;
   float grade;
   }
```

```
interface Thesis{
   extent theses;
   keys title, author;

   string title;
   Gradstudent author;
   string status;
   }
```

```
interface Department{
   extent departments;
   key name;

   string name;
   string chairperson;
   }
```

Bibliography

[AAD+93] R. Ahmed, J. Albert, W. Du, W. Kent, W. Litwin, and C.Shan. An Overview of Pegasus. In *Proc. IEEE Workshop on Research Issues on DE: Interoperability in Multidatabase Systems, Vienna*, pages 273–277, April 1993.

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

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

[Bai95] Ian Bailey. *EXPRESS-M Reference Manual.* CIMIO Ltd., Brunel Science Park, Englefield Green, Surrey, TW20 0JZ, ENGLAND, August 1995.

[BCD+93] Omran Bukhres, Jiansan Chen, Weimin Du, Ahmed K. Elmagarmid, and Robert Pezzoli. InterBase: An Execution Environment for Heterogeneous Software Systems. *IEEE Computer*, 26(8):57–69, August 1993.

[BGMS95] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. *Transaction Management in Multidatabase Systems*, chapter 28, pages 573–591. ACM Press, 1995.

[BGN+88] E. Bertino, R. Gagliardi, M. Negri, G. Pelagatti, and L. Sbattella. The comandos Integration System: an Object Oriented Approach to the Interconnection of Heterogeneous Applications. In *Advances in Object-Oriented Database Systems 2nd International Workshop on ObjectOriented Database Systems*, pages 213–218, New York, September 1988. Springer-Verlag.

[BGN+89] E. Bertino, R. Gagliardi, M. Negri, G. Pelagatti, and L. Sbattella. Integration of Heterogeneous Database Applications through an Object-Oriented Interface. *Information Systems*, 14(5):407–420, 1989.

[BHP92] M.W. Bright, A.R. Hurson, and S.H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *IEEE Computer*, 25(3), March 1992.

[BHR+95] Arne-Jørgen Berre, Frode Høgberg, Magnus Rygh, David Skogan, and Jan Øyvind Aagedal. SISIP – A Systems Integration Platform based on Distributed Persistent Objects. Technical report, Department of Informatics, SINTEF, 1995.

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

[BOH+92] A. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A Transaction Model for Active Distributed Object Systems. In *Transaction Models for Advanced Database Applications*, chapter 5, pages 123–158. Morgan-Kaufmann, 1992.

[Cat94] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, 1994.

[CHS91] C. Collet, M. Huhns, and W-M. Shen. Resource Integrating Using a Large Knowledge Base in Carnot. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 24(12):55–63, December 1991.

[Cod70] E. F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–390, June 1970.

[DBT71] CODASYL DBTG. Report of the CODASYL DataBase Task Group. *ACM Computing Surveys*, April 1971.

[Dit86] Klaus R. Dittrich. Object-Oriented Database Systems: The Notion and the Issues. Technical report, Forschungszentrum Informatik (FZI) an der Universität Karlsruhe, 1986. Preface in 2nd International OODBMS Workshop 1988.

[DKH92] P. Drew, R. King, and D. Heimbigner. A Toolkit for the Incremental Implementation of Heterogeneous Database Management Systems. *VLDB*, 1(2):241–284, October 1992.

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

[DMN70] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language.* Norwegian Computing Center, Oslo, 1970. NCC Publication S-52.

[GCS93] M. García-Solaco, M. Castellanos, and F. Saltor. Discovering Interdatabase Resemblance of Classes for Interoperable Databases. In *Proceedings of the 2nd International Workshop on Interoperability in Multidatabase Systems*, pages 26–33, 1993.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983.

[HJK+92] M. N. Huhns, N. Jacobs, T. Ksiezyk, W. M. Shen, M. P. Singh, and P. E. Cannata. Enterprise Information Modeling and Model Integration in Carnot. In *Enterprise Integration Modeling, Proceedings of the First International Conference*, pages 290–299, Cambridge, Mass., 1992. The MIT Press.

[HM85] Heimbigner and McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office IS*, 3(3), July 1985.

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

[KCGS95] Won Kim, Injun Choi, Sunit Gala, and Mark Scheevel. *On Resolving Schematic Heterogeneity in Multidatabase Systems*, chapter 26, pages 521–550. ACM Press, 1995.

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

[KFM+96] Wolfgang Klas, Peter Fankhauser, Peter Muth, Thomas C. Rakow, and Erich J. Neuhold. Database Integration using the Open Object-Oriented Database System VODAK. In Omran A. Bukhres and Ahmed K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, chapter 14, pages 472–532. Prentice-Hall, 1996.

[Kim95] Won Kim. Introduction to Part 2: Technology for Interoperating Legacy Databases. In Won Kim, editor, *Modern Database Systems – The Object Model, Interoperability, and Beyond*, chapter 25, pages 515–520. ACM Press, 1995.

[Kor94] Espen Frimann Koren. Semantic Proximity in Object-oriented Data Models. Master's thesis, Department of Informatics, University of Oslo, May 1994.

[KS91] Won Kim and Jungyun Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, 22(3):183–236, December 1991.

[LDS92] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed Object Management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), August 1992.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.

[LS92] Petter Lowzow and Per Solberg. COOM ODL Compiler. Master's thesis, University of Trondheim, May 1992.

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

[MHG+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie. Distributed Object Management. In A. K. Elmagarmid, editor, *International Journal of Intelligent and Cooperative Information Systems*, volume 1, pages 5–42, March 1992.

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

[MY95] Weiyi Meng and Clement Yu. *Query Processing in Multidatabase Systems*, chapter 27, pages 551–572. ACM Press, 1995.

[Neb88] Bernhard Nebel. Computational Complexity of Terminological Reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, April 1988.

[NSGS89] Navathe, S.B., Gala, and S.K. A Federated Architecture for Heterogeneous Information Systems. In Yu, editor, *1989 Workshop on Heterogenous Databases*, December 1989.

[OMG92] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 1.1 edition, 1992.

[Øre92] Ole Øren. *Proving the Equivalence of Databases and Database Schemas.* PhD thesis, Institute for Informatics, University of Oslo, September 1992.

[PBE95] Evaggelia Pitoura, Omran Bukres, and Ahmed Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2):141–195, June 1995.

[PSH91] G. Pathak, B. Stackhouse, and S. Heiler. EIS/XAIT Project: An Object-Based Interoperability Framework for Heterogeneous Systems. *Computer Standards and Interfaces*, 13(1–3):315–319, October 1991.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[SCG91] F. Saltor, M. Castellanos, and M. García-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. *ACM SIGMOD Record*, 20(4):44–48, December 1991.

[SCG94] F. Saltor, B. Campderrich, and M. García-Solaco. On Architectures for Federated DB Systems. In *Sixth ERCIM Database Research Group Workshop on Deductive and Interoperable Databases.* ERCIM, Barcelona, November 1994.

[SDS96] S. Y. W. Su, A. Doshi, and L. Su. HKBMS: An Integrated Heterogeneous Knowledge Base Management System. In Omran A. Bukhres and Ahmed K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, chapter 17, pages 589–616. Prentice-Hall, 1996.

[SG89] A. P. Sheth and S. K. Gala. Attribute Relationships: an Impediment in Automating Schema Integration. In Yu, editor, *1989 Workshop on Heterogenous Databases*, December 1989.

[SK92] Amit Sheth and Vipul Kashyap. So far (Schematically) yet so near (Semantically). In David K. Hsiao, Erich J. Neuhold, and Ron Sacks-Davis, editors, *IFIP DS-5 Semantics of Interoperable Database Systems*, pages 272–301, Lorne, Victoria, Australia, November 1992.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Sol90] Richard M. Soley, editor. *Object Management Architecture Guide*. Object Management Group(OMG), Frameington, MA, November 1990.

[SSG+91] A. Savasere, A. Sheth, S. Gala, S. Navathe, and H. Marcus. On Applying Classification to Schema Integration. In *International Workshop on Interoperability in Multidatabase Systems, Kyoto*, April 1991.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 2 edition, 1991.

[TK78] D. Tsichritzis and A. Klug, editors. *ANSI/X3/SPARC DBMS Framework*. AFIPS Press, 1978.

[TL76] D. Tsichritzis and F. Lochovsky. Hierarchical Database Management: A Survey. *ACM Computing Surveys*, 8(1), March 1976.

[TLM+92] C. Tomlinson, G. Lavender, G. Meredith, D. Woelk, and P. Cannata. The Carnot Extensible Service Switch (ESS) – Support for Service Execution. In *Enterprise Integration Modeling, Proceedings of the First International Conference*, pages 493–502, Cambridge, Mass., 1992. The MIT Press.

[WCH+93] D. Woelk, P. Cannata, M. Huhns, W. Shen, and C. Tomlinson. Using Carnot for Enterprise Information Integration (Synopsis). In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 133–136, San Diego, CA, January 1993.

[WSHC92] D. Woelk, W. Shen, M. N. Huhns, and P. E. Cannata. Model-driven Enterprise Information Management in Carnot. In *Enterprise Integration Modeling, Proceedings of the First International Conference*, pages 301–309, Cambridge, Mass., 1992. The MIT Press.