Multidatabase Integration using Polymorphic Queries and Views

by

Magnus Werner

March 1996
ISBN 91-7871-687-X
Linköping Studies in Science and Technology
ISSN 0280-7971
Thesis 546
LiU-Tek-Lic 1996:11

ABSTRACT

Modern organizations need tools that support coordinated access to data stored in distributed, heterogeneous, autonomous data repositories.

Database systems have proven highly successful in managing information. In the area of information integration *multidatabase systems* have been proposed as a solution to the integration problem.

A multidatabase system is a system that allows users to access several different autonomous information sources. These sources may be of a very varying nature. They can use different data models or query languages. A multidatabase system should hide these differences and provide a homogeneous interface to its users by means of *multidatabase views*.

Multidatabase views require the query language to be extended with *multidatabase queries*, i.e. queries spanning multiple information sources allowing information from the different sources to be combined and automatically processed by the system.

In this thesis we present the integration problem and study it in an *object-oriented* setting. Related work in the area of multidatabase systems and object views is reviewed. We show how *multidatabase queries* and *object views* can be used to attack the integration problem. An implementation strategy is described, presenting the main difficulties encountered during our work. A presentation of a multidatabase system architecture is also given.

This work has been supported by The Swedish Board for Industrial and Technical Development (NUTEK).

Department of Computer and Information Science
Linköping University
S-581 83 Linköping
Sweden

Acknowledgements

I would like to thank my professor and supervisor Tore Risch for his support, encouragement, and providing a constant flow of new ideas. I would also like to thank all members of EDSLAB and RTSLAB for fruitful discussions and suggestions.

Anne Eskilsson provided excellent administrative service, always in cheerful spirits. I am also grateful to Ivan Rankin who helped me with the various subtleties of the English language.

Last, but not least I would like to thank Pernilla for putting up with me. You have been the best support I could have asked for. I promise that we will spend more time together now.

Magnus
Linköping
March 1996

1 Introduction

In this thesis I will discuss some of the problems with information *integration* existing in *Object-Relational*[1] *multidatabase systems (ORMDBS)* and present solutions to some of these problems. Readers are assumed to be familiar with theoretical and practical aspects of database management systems.

Modern organizations have strong requirements for tools that support coordinated access to data stored in distributed, heterogeneous, autonomous data repositories. Several reasons for this are discussed in [Connors and Lyngbaek, 1988]. Organizations evolve over time, they merge and split. This influences the way that data are managed within the organizations. In fact, the choice of an information management system depends on the application requirements and on the available technology. As these evolve over time, an organization ends up having several, most likely, heterogeneous, information management systems. For economic or practical reasons it is not possible to migrate information from old systems to new ones and yet the information in the old systems must be kept available to new systems. This has become known as the *legacy problem* [Brodie and Stonebraker, 1992]. Performance may also dictate information management policies. To provide adequate performance it may be necessary to maintain the data in different data repositories with different capabilities, structures and organizations. Finally, not all sources of data may belong to the same organization. This is the case, for example, when several contractors work together on some large governmental project. Therefore, applications needing data from several information sources have to bridge the gap among the various systems. There is a need for *integration*.

Different solutions have been proposed over the years. The general concept of *mediators* was first proposed in [Wiederhold, 1992] as a framework for solving the above problem. A mediator is a middle layer software between an application and an information source. Several different types of mediators may exist. They may, for example, translate information between various formats, locate information, or perform filtering of information.

Database systems have proven highly successful in managing information. In the area of information integration multidatabase systems have been proposed as a solution to the integration problem. A *multidatabase system* (*MDBS*) is a system that allows users to access several different autonomous information sources, often referred to as *external data sources* (*EDS*) or *component systems*.

1. An object-oriented database system with a relationally complete query language allowing meta queries over schema information.

These data sources may be of a highly varying nature. They can use different data models or query languages if they have one at all.

In figure 1.1 a multidatabase system is shown with three different external data sources, a file, a *relational database system (RDBMS)*, and an *Object-Oriented database system (OODBMS)*. Files provide no query language at all, relational databases typically provide SQL or a similar declarative query language, and object-oriented databases provide at least navigational access and often also some query language. A multidatabase system should hide these differences and provide a homogeneous interface to its user that allows him/her to make equally powerful queries no matter what EDS the information comes from.

Providing users with multidatabase queries, i.e. allowing users to transfer and combine information, is not enough. Often users want to organize the informa-

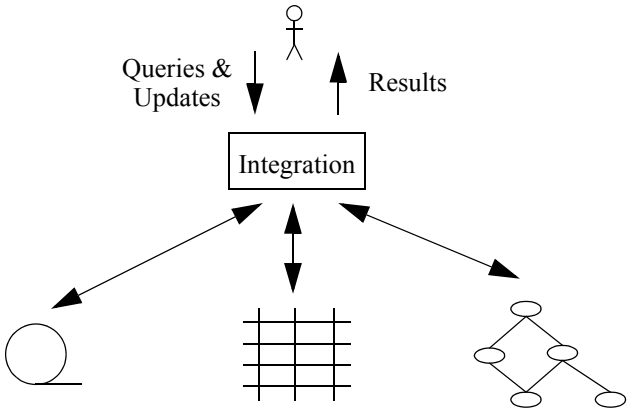


Figure 1.1: Multidatabase system.

tion according to their own preferences. *Multidatabase views* are needed for this [Krishnamurthy et al., 1991], [Litwin and Abdellatif, 1986], and [Litwin et al., 1990]. In relational multidatabase systems this mechanism is fairly simple since we only have to deal with literal data types such as integers or strings. However, in object-relational multidatabase systems things get more complicated since we have to deal with abstract data types and objects that have an identity that is unaffected by values of any properties. We need to extend the view mechanism to handle these difficulties.

The main contributions of this thesis are:

- We provide a presentation of the integration problem.
- We review related work in the area of object views.
- It is shown how the combination of multidatabase queries and objects views can be used to attack the integration problem.
- A presentation of a multidatabase system architecture is given.
- An implementation strategy is described, presenting the main problems with this approach encountered during our work.

The presentation throughout the thesis is kept informal as it is our intention to

present the intuitions behind information integration and multidatabase query processing.

1.1 Outline of the thesis

In section 1.2 the system architecture of AMOS, the database system used in our research, is presented. We then proceed by describing what we mean by integration and some approaches to realize it in section 1.3.

When integrating information, several different problems have to be addressed; some of the typical integration problems are presented in section 1.4. In section 1.5 we then present integration problems typical when integrating object-oriented information.

In chapter 2 a review of related work is given in the area of object views and it is followed in chapter 3 by a presentation of the AMOS data model.

Object views in AMOS are presented in chapter 4 where an example is given presenting how object views and multidatabase queries can be used for integration. The implementation of object views in AMOS is then outlined in chapter 5 and problems encountered during our work are described.

Finally, a discussion and outline of future work are presented in chapter 6.

1.2 The AMOS System

AMOS provides us with the two facilities needed to perform a successful integration, multidatabase queries and object views. Both of them are needed and we will now provide a brief description of the AMOS architecture and how multidatabase queries and object views are used in the system.

The AMOS (Active Mediators Object System) architecture uses the mediator approach that introduces an intermediate level of software between applications and their information sources. This allows new applications to access old information sources while these sources are still accessible from old applications. Certain information processing can also be performed by the middle layer. We call the intermediate modules *active* mediators, since they support active database facilities.

The AMOS architecture is built around a main memory based platform for intercommunicating information bases. Each AMOS server has full DBMS facilities, such as a local database, a data dictionary, a query processor, a transaction manager, and a communication manager. Central to the AMOS architecture is an object-relational query language, AMOSQL, supporting object-oriented abstractions and declarative queries. It is extensible to allow for easy integration with other systems.

If we want to provide access to some *external data source* we link it with an AMOS server as shown in figure 1.2. This AMOS is known as a *translator* (*T*-

AMOS) as the AMOSQL query language is extended with functions providing access to the EDS. These functions will, however, return the information from

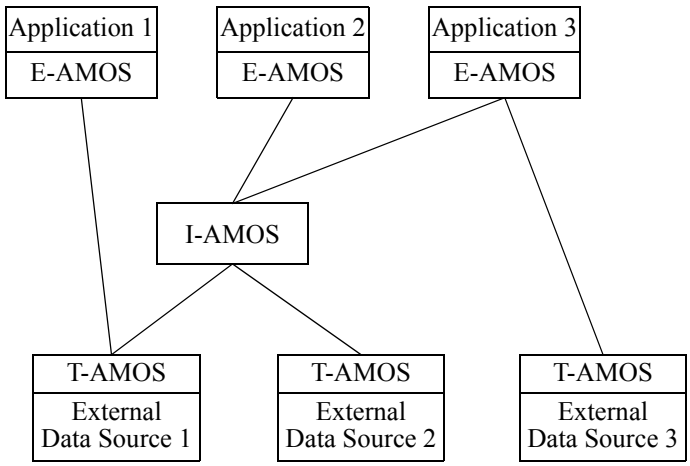


Figure 1.2: AMOS Architecture.

the EDS as literal valued information, i.e. as integers or strings, and usually we want to convert this information into objects, the prime construct in the AMOS data model. To do this we use *mapped types*. Objects as opposed to literals can have properties associated with them and the value of these properties may vary over time without affecting the identity of the object.

Mapped types are defined by a query expression that converts literals to objects and this allows us to provide an object view of information residing in an EDS.

Not only do we want access to EDSs but we want to combine information from various AMOS servers as well. An AMOS server performing this kind of integration is called an *integrator* (*I-AMOS*). AMOSQL provides multidatabase queries that allow us to do this. However, issuing a multidatabase query may render foreign objects as result. Such objects lack local type structure since they are retrieved from another AMOS server. To solve this problem derived types are defined by declarative queries and allow us to give objects a type membership based on the values of some properties.

When combining information from several AMOS servers, information about the same real-world entity may be present in several of the servers. This leads to the undesirable situation where this entity is represented by several database objects. This problem has become known as the *object-equivalence problem* [Tresch and Scholl, 1994], [Kent, 1991], and [Eliassen and Karlsen, 1991]. Mapped types help us in this situation as well by providing facilities to produce new objects representing the equivalent objects. This maintains the relationship where one real-world entity is represented by one database object.

Applications can also be linked directly with AMOS, a so-called *embedded AMOS* (*E-AMOS*). The application may then store its information in the embedded AMOS using the efficient storage structures provided. Also, the embedded

AMOS provides the application with opportunities to share information with other AMOS servers.

Our architecture is very flexible since it is symmetrical, i.e. each AMOS server has some least common functionality compared to other AMOS servers. This allows any AMOS to access any other AMOS acting as a server. Thus we can choose whether we want to hide or not the fact that information comes from various sources. For example, in figure 1.2 Application 2 is not aware of the fact that information is retrieved from two different external data sources since these sources have been integrated by the AMOS in the middle. Application 3, however, performs the integration locally. It accesses both EDS1 and EDS2 through the middle AMOS but it also accesses EDS3 directly. Thus, the local database administrator can decide whether to hide this fact locally or if it should be kept explicit that information comes from different sources.

The symmetry of the solution also means that not only applications issue request and receive answers. It could be the other way around as well. The applications may be questioned and send answers back.

1.3 Integration

When can information be considered as integrated? To us, information is integrated when it is possible to perform some *automatic* processing of it. Thus, displaying information from two different information sources on the same screen is not information integration if a user has to examine the information and combine it by hand to obtain the desired result. However, should the system allow a procedure to be specified to perform the combination automatically, then we can say that the system integrates the information.

Integration allows the user to specify requests in terms of *abstractions* that define complex conceptual structures whose physical representations or implementations may span multiple, heterogeneous information sources [Heiler and Siegel, 1991].

Several different kinds of heterogeneity need to be addressed to successfully integrate different information sources. The sources may be distributed, running on different hardware using different operating systems. Different kinds of data models may be used, *data model heterogeneity*, and even if the data model does not differ the data manipulation languages may, *language heterogeneity*. Even when trying to integrate two information sources employing identical database management systems, we most likely face problems since the same entity can be modelled in several different ways. This is called *semantic heterogeneity*.

These heterogeneities have to be resolved when integrating different EDSs. In [Sheth and Larson, 1990] a five level schema architecture (figure 1.3) is presented that addresses these problems.

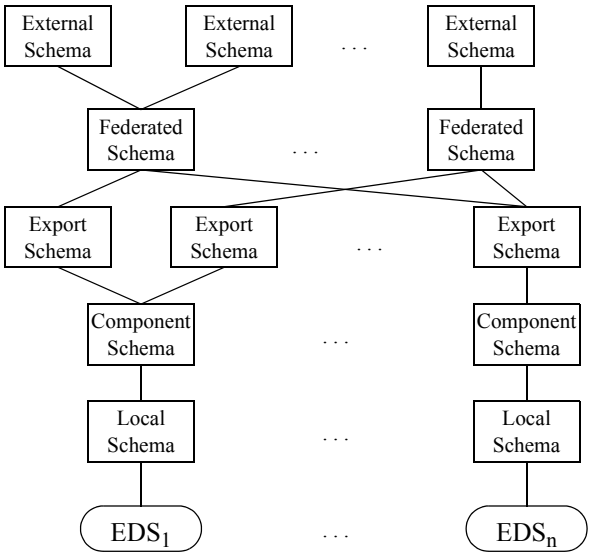


Figure 1.3: Five-level schema architecture of multidatabase systems (from [Sheth and Larson, 1990]).

At the lowest level we have the external data sources that we want to integrate. A *local schema* is defined for each of them. The local schema defines what information is stored in the EDS and how it is organized. The local schema may be either explicit or implicit. For example there is no explicit schema defined for a text file. However, there exists such a schema implicitly as it is possible to retrieve and interpret information from the file. The local schema is often expressed in the native data model of the source. For example, if the EDS is a relational database then the local schema is expressed in the relational data model.

Component schemas map local schemas into a *canonical data model* (*CDM*) thereby resolving data model heterogeneity and language heterogeneity. The component schema contains the same information as the local schemas do. Also, it is possible to perform *semantic enrichment* in the component schema, i.e. semantics that is missing in a local schema can be added to its component schema. The CDM has to be as least as expressive as the most powerful data model used by the EDSs, otherwise we lose information. Having a powerful CDM facilitates semantic enrichment, e.g. in the relational model generalization/specialization cannot be expressed directly. If the CDM is an object-oriented data model then this can be made explicit.

A query to a component schema is translated into queries to the underlying local schema. The results of these queries are then processed to form an answer to the initial query.

For each component schema, one or more *export schemas* may be defined. An export schema represents a subset of the component schema. It defines what

part of the component schema is available to a particular group of users.

A *federated schema* is an integration of multiple export schemas. It makes it possible to access data from multiple external data sources as if it was stored in a single database. A query against a federated schema is translated into queries to the underlying export schemas. The results of these queries are then processed to form an answer to the initial query. All federated schemas are expressed in the CDM. Federated schemas resolve semantic heterogeneities between different export schemas. The process of constructing a federated schema is known as *schema integration* [Sheth and Larson, 1990].

For each federated schema, one or more *external schemas* can be defined. An external schema represents a subset of a federated schema. It can be transformed in various ways to suit the needs of a particular user group. It may even be expressed in a data model other than the federated schema, as described in [Sheth and Larson, 1990].

The purpose of the five-level schema architecture is to hide from users the fact that they are actually accessing several different EDSs. They should perceive the system as if it was one centralized database. Consequently the query language is the same as in the case of a centralized database management system.

It is not always desirable for various reasons, e.g. performance reasons, to hide from users the fact that they are working with several disparate EDSs [Wang and Madnick, 1990] and [Waldo et al., 1994]. Also, creating and maintaining federated schemas may not be possible if the number of EDSs varies or schemas undergo frequent changes [Litwin and Abdellatif, 1986], [Litwin et al., 1990], and [Milliner et al., 1995]. Instead of providing an integrated view the multidatabase system should therefore provide users with a *multidatabase language* powerful enough to allow interdatabase queries to be specified. Users may then themselves select the sources they are interested in and retrieve and combine information from those. In the five-level schema architecture this corresponds to having no federated schema and no export schema.

In AMOS we opt for the multidatabase language approach but provide constructs that allow construction of *multidatabase views*, allowing transparent integration of several EDS as in the five-level schema architecture. It should be noted that we still need a canonical data model since component- and export schemas have to be defined.

1.3.1 The Canonical Data Model

The choice of canonical data model is important for successful integration. In [Saltor et al., 1991] the suitability of different data models as CDM is evaluated. It is observed that to capture the semantics already expressed in native data models of the various EDSs, the CDM must have, depending on the data model, an expressiveness equal to or greater than any of native models of the EDSs.

The following are some desirable features of a good CDM:

- **Classification/Instantiation**. It should support the notion of a *class* (or type) and an *instance*. The class concept is a key feature in object-oriented data models. A class is similar to the notion of an *abstract data type* (*ADT*). Functions and procedures that operate on the class are *methods* of the class. Methods define the behaviour of *objects*. All objects are *instances* of some class and every object is unique and is usually assigned a unique *object identifier* (*OID*). Thus two objects can exhibit the same behaviour and yet be different objects. The *extent* of a class is the set of objects which are currently instances of the specific class.
  In this thesis we will use the term *type* in favour of the term *class* and *function* in favour of *method*.
- **Generalization/Specialization**. *Inheritance* should be supported. It must be possible to organize classes in an inheritance hierarchy of arbitrary depth. The ordering operator used is usually *set inclusion* [Cardelli and Wegner, 1985].
  Inheritance means that if a type *A* is a subtype of type *B* and if *B* has a certain property, *p*, then the type *A* also possesses the property *p*. Properties include functions and variables associated with a type.
  Multiple inheritances and different kinds of specialization are optional, but recommended.
- **Aggregation/Decomposition**. Aggregation means that a new type is created as the Cartesian product of other types. For example the type `Address` is an aggregation of the types `City`, `Street`, and `ZipCode`. Complex objects of this kind are created by applying the *tuple* or *record* constructor to existing objects. *Decomposition* means that we must be able to extract the components of complex objects of this kind.
  There also exists another form of aggregation known as *grouping*. Grouping means that a set of objects of some existing type are gathered together. For example, the `drives` attribute of a person is a set of objects of the type `Car`. Complex objects of this kind are created by applying the *set* or *bag* constructor to existing objects, i.e. the order amongst the members is not important.
- **Operations and integrity constraints**. The CDM should allow definition of new operations and integrity constraints. This allows integration of traditional and non-traditional data sources, where *structural mapping* cannot be used, by extending the functionality of existing methods defined for various local views [Bertino, 1991].
- **View mechanism**. A view mechanism is needed and it should be at least as powerful as a view mechanism using relational algebra.
- **Type hierarchy integration operators**. A CDM should also support implementation of type-hierarchy integration operators as described in “Superviews” on page 17 or it must be possible for the user to achieve the equivalent results within the CDM. Type-hierarchy integration operators are needed to integrate schemas with different structures using inheritance and classification. The operators can then be applied to the schemas to transform them into some common structure.

- **Multiple semantics**. In [Sheth and Larson, 1990] the following example is given to illustrate multiple semantics. Suppose there exist two databases *DB1* and *DB2* containing information about shoes. Two users *userA* and *userB* wish to integrate these two databases; however, their perception of colours differs. UserA sees as cream what is cream in DB1 and what is tan in DB2, whereas userB considers cream what is tan or cream in DB1 and what is tan or white in DB2.
  Multiple semantics means that it must be possible for two users to integrate the underlying sources in different ways.

Object-oriented data models support most or all of the above characteristics and are thus suitable as CDMs.

1.4 Schema Discrepancies

Whether integrating EDSs using the same or different data models, we have to address possible schema discrepancies. Suppose that we have the following two object-oriented databases (fig. 1.4) that we want to integrate. Both of the databases store information about publications.

In the figure an ellipse represents a type and the name of the type is written in the ellipse. Functions associated with the type are written next to it. The name of the function is written in small letters and stands before the colon (:) whereas the range of the function is the type whose name is written after the colon. The domain of the function is the type it is associated with. Arrows represent inheritance. Thus, in `DB1` an instance of `Editor` or `Secretary` is an instance of `Person` as well. This means that `name` may be applied to editors and secretaries as well, since they are persons, too.

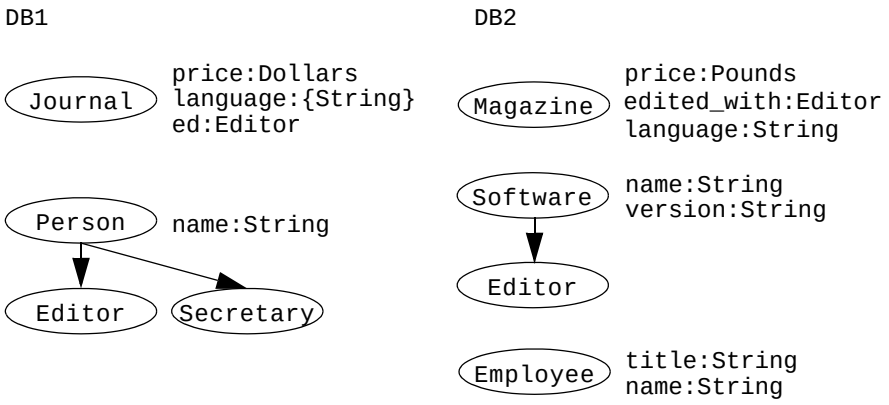


Figure 1.4: Two example databases with schema discrepancies.

The two schemas contain various schema discrepancies which we will take a closer look at now. Before we proceed we want to point out that the classification presented here is not the only possible and many other exist. Also, several

different terms have been used over the years denoting the same concepts. Nor do we claim that the enumeration is complete, we simply want to present some typical problems that have to be addressed.

1.4.1 Naming Conflicts

A common type of conflict is the *naming conflict* where different entities or properties with different semantic meaning may share a name (*homonyms*), or semantically related properties are named differently (*synonyms*).

In the example above we see a typical example of a homonym. Both databases contain a type `Editor` but in `DB1` it is a person whereas in `DB2` it is a program used for word processing. The example also contains an example of synonyms as the terms `Journal` and `Magazine` denote the same concept.

1.4.2 Scaling Conflicts

In different databases, different units of measure may be used measuring the same thing. This has been termed a *scaling conflict*.

An example of scaling conflict is the price of `Journal` and `Magazine` respectively. The former has `Dollar` as the unit for price whereas `Magazine` uses `Pound` as price unit.

Another example could be if prices in one database included VAT whereas the other database recorded price information without VAT.

1.4.3 Structural Differences

The same concept may be represented in two schemas by different modelling constructs. In `DB2` the title of an `Employee` tells us what work the person is doing, i.e. a secretary would have the string *"Secretary"* as value of the property title. In `DB1` the same thing is modelled using types. `Secretary` is a subtype of `Person`.

From an integration point of view a CDM providing only one modelling construct would leave us with no choice as to how we represent a certain concept. Thus, we would avoid structural differences. However, it would be highly impractical since it would force users to model their problem in an unnatural way. Therefore it is important that the data model supports transformations between different modelling constructs [Krishnamurthy et al., 1991] and [Chomicki and Litwin, 1994]. If not, we are not able to resolve structural differences in different component schemas.

Structural differences have also been called *representation conflicts* [Rafii et al., 1991].

1.4.4 Constraint conflicts

Two schemas may impose different constraints for some property associated with a type [Kent, 1988]. For example the key of an entity type may be different in each schema. An example of a *constraint conflict* is shown in figure 1.4 where `language` of `Journal` is set-valued, i.e. a `Journal` is printed in several languages whereas a `Magazine` is only available in one language.

Other terms used for constraint conflicts are *dependency conflicts* [Rafii et al., 1991].

1.4.5 Key conflicts

A key conflict arises when different keys are assigned to the same conceptual entity in different schemas. An example would be if we have two databases DB1 and DB2 each storing information about persons. In DB1 social security numbers are used as keys whereas in DB2 names are used. Thus, names are required to be unique in DB2 but not in DB1. Therefore it is possible to store information about two persons named Joe in DB1, but this cannot be accomplished in DB2.

1.4.6 Value conflicts

Value conflicts arise when the same property for some real world entity is stored in multiple EDSs and the values of the property differ. It should be noted that it cannot be the identifying property (key) that differs since we then would not perceive the different entities as representing the same real world entity.

For example, assume that two relational databases store information about the annual income of persons. If we have two entries, one in each database, where, the social security number (the key) is equal but the recorded income differs then we have a value conflict. Should, however, the social security number differ, then we would have assumed that the two entries concerned different individuals.

1.5 Object-Oriented Integration Problems

The schema discrepancies discussed in section 1.4 may arise during integration of any two schemas irrespective of data model used. To complicate things further, when integrating *homogeneous object-oriented multidatabase systems*, as AMOS, one usually wants to take advantage of types and classes defined in the EDSs to be integrated. In this section we describe some problems that have to be addressed if we intend to integrate such systems.

1.5.1 Behaviour conflicts

This corresponds to integrating two attributes with different domains. Likewise, methods that we wish to integrate may behave differently. Thus we need a

view mechanism that is general enough to resolve this kind of conflict.

1.5.2 Object Equivalence

In [Tresch and Scholl, 1994], [Kent, 1991], and [Eliassen and Karlsen, 1991] the problem of *object equivalence* is discussed. When integrating EDSs it may happen that the same real world entity is *semantically replicated*, i.e. represented by multiple objects in different component databases. Due to local autonomy, OID domains of different EDSs are pairwise disjoint, such that no two objects from different EDSs can be identical. Object integration requires mechanisms to integrate objects that represent the same real world entity, such that the MDBS treat them as a single object in queries.

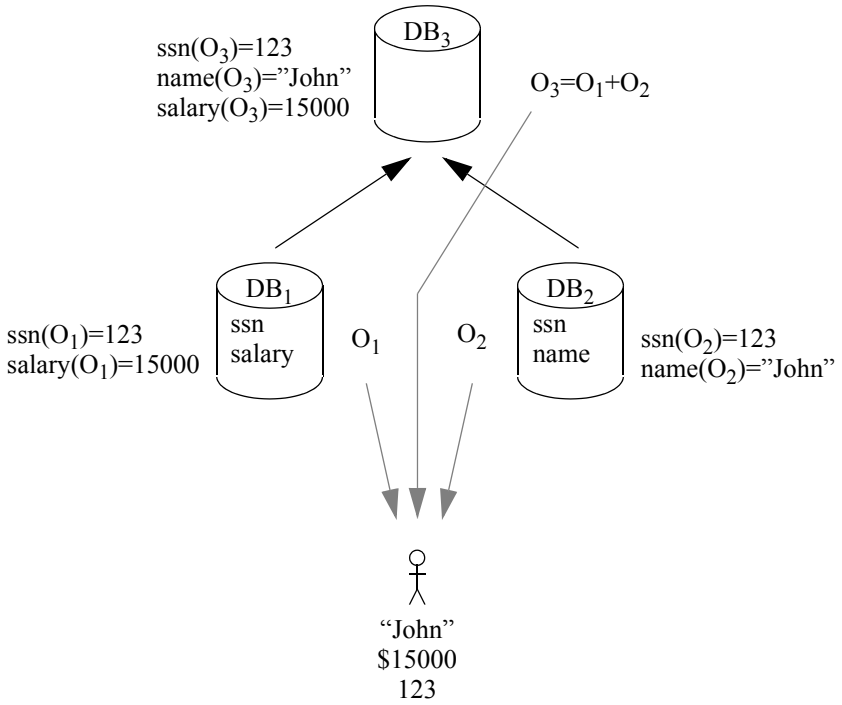


Figure 1.5: The object equivalence problem.

OIDs are not adequate to globally identify objects, since they are internal representations within each EDS as noted in [Tresch and Scholl, 1994]. Global identity must be based on characterizing values.

For example, in figure 1.5 two homogeneous object-oriented databases, DB_1 and DB_2 are integrated in a new database DB_3. In DB_1 we store information about a person’s salary and his social security number. In DB_2 we store the name of a person and the social security number. As we can see, John is represented in both DB_1 and DB_2 by objects O_1 and O_2 respectively. When we integrate DB_1 and DB_2 we do not want to have John represented by several objects;

instead we want a single object (O_3) to represent him. The only way we can tell that O_1 and O_2 represent the same object is by the social security number.

1.5.3 Type Integration

Integration requires the ability to combine and compare information from various EDSs, as stated earlier. To combine and compare information requires that the information is of the same type. In an object-oriented multidatabase system this is a problem since we have to decide on how to merge the different type hierarchies of the EDSs, i.e. it must be possible to state that literals and other types are "equal". There are two different approaches to this problem, the *global schema approach* and the *multidatabase language approach*.

The Global Schema Approach

In [Tresch and Scholl, 1994] multidatabase systems are classified on five different levels according to the amount of integration they provide using the global schema approach.

Level 0 systems provide no integration whatsoever. They only allow transactions spanning multiple EDSs but information from various EDSs may not be compared. Level 4 systems provide the highest degree of integration and correspond to distributed databases, i.e. they do not differ from centralized databases in respect to object identity and type hierarchy. There exists a single domain of OIDs and a single global type hierarchy. Level 1 to level 3 systems provide various degrees of integration and we will examine them more closely.

- **Level 1 integration** or *schema composition*. Names of all schema elements from EDSs are just imported and made globally available. Type and class systems of local databases are combined, without establishing connections between composite systems. As an anchor, basic data types of component systems are assumed to be identical. This ensures that at least values of elementary data types can be compared between component systems. Local object type and class hierarchies of the EDSs are put together by defining a new global top type and a new global top class.
  Schema composition makes it possible to formulate queries that involve multiple EDSs. For example, it is now possible to compare names of persons stored in different databases. We are, however, limited to comparisons of literals.

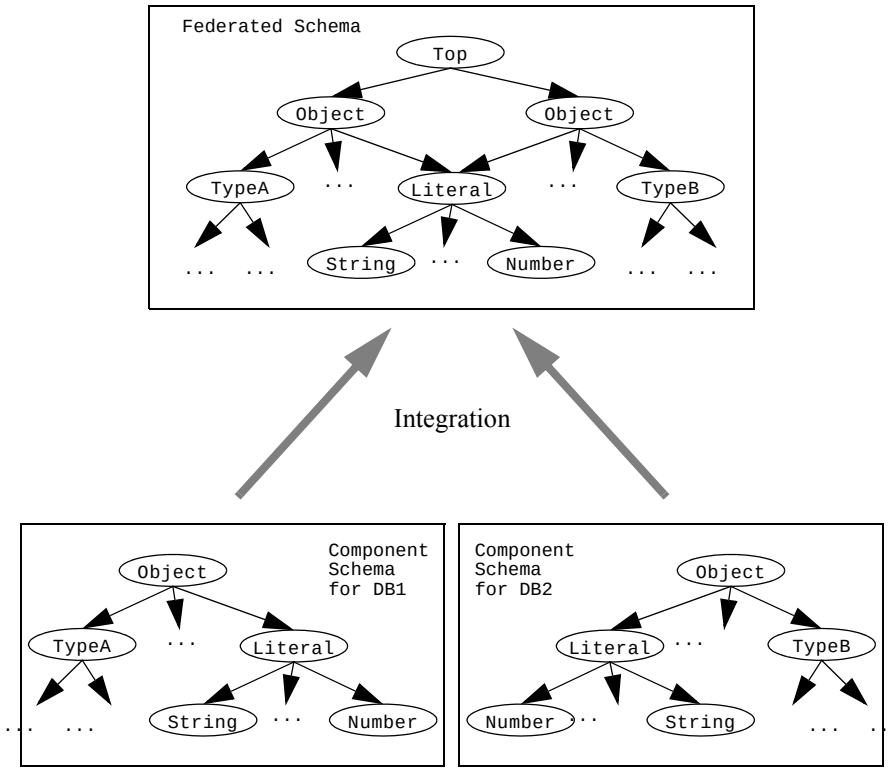


Figure 1.6: Level 1 integration.

- **Level 2 integration** or *virtual integration* provides *multidatabase views* providing a uniform, virtual interface over multiple databases. Multiple objects representing the same real world object can also be integrated at level 2, i.e. the various objects representing the entity are *merged* and can be treated as one object. This requires a view mechanism more powerful than the relational one, as noted in section 1.3.1. In many object-oriented data models, types and functions are objects as well. This provides a uniform way to provide integration of objects, types and functions. Any two types can then be merged in the same way as the literal types were in schema composition integration.
- **Level 3 integration** or *real integration*. While virtual integration provides us with the ability to merge objects, types and functions, it does not allow interdatabase functions to be created, i.e. a function with a signature involving types from more than one EDS. For example we could not create a function `foo:TypeA->TypeB` since `TypeA` resides in `DB1` and `TypeB` in `DB2`. This is, however, allowed at this third level of integration.

A fundamental property of the global schema approach is that it does not modify the component schemas of the various EDSs. All access has to be through the federated schema if we want to have the integrated view. Of course we

could still access the various EDSs through their component schemas, but then we lose the integration.

An alternative way, instead of having a federated schema, would be to modify the component schemas of the various EDSs so that each component schema would reflect the current integration. However, this affects all component schemas being integrated. The various EDSs are thus made aware of the integration, i.e. a component schema no longer reflects only the local schema of an EDS but reflects parts of other EDSs local schemas as well.

One of the problems with the global schema approach is that it will not scale to the proportions needed in most multidatabase environments [Bright et al., 1992]. Since *one* global schema is used, it is not feasible to build a multidatabase systems having a *large* number of EDSs. The global schema must integrate all the export schemas of the EDSs and its size might be prohibitive.

Maintenance of the global schema may be a time-consuming task. As the EDSs are autonomous, they may at any time change their export schemas and this may require modifications to the global schema. Furthermore, EDSs may join and leave the multidatabase as they wish. Should the number of participating EDSs be dynamic as can be expected in certain situations [Litwin et al., 1990], then it will not be possible to maintain the global schema.

The global schema approach does have some advantages. Since integration is centralized, the system appears to users as a *single integrated database*. Thus, it is obvious where to look for information. The global schema approach also makes it easier to enforce and maintain integrity constraints on interdatabase dependencies and relationships. Furthermore, integration efforts are shared which means that a certain integration only has to be done once and all users can then benefit from it.

In our approach to integration we want to provide the equivalent of level 3 integration. Currently level 2 integration is provided but our work should be extensible to provide level 3 integration. However, instead of the global schema approach we opt for the multidatabase approach described next.

The Multidatabase Language Approach

It has been argued that multidatabase language systems are the only reasonable solution in a large multidatabase environment where the number of EDSs is large and/or varies dynamically.

In the multidatabase language approach no federated schema is created and, thus, the bottleneck of the previous approaches has been eliminated. However, by eliminating federated schemas we have to extend the query language to support multidatabase operations since there no longer exists a *single* integrated database that users can access.

Each user or EDS should build their own schema incorporating information of interest. Thus users at one EDS may incorporate schema information from other EDSs in the system and these EDSs may not be aware that they are used;

at least it is not reflected in their schemas. This can cause some anomalies as we shall see.

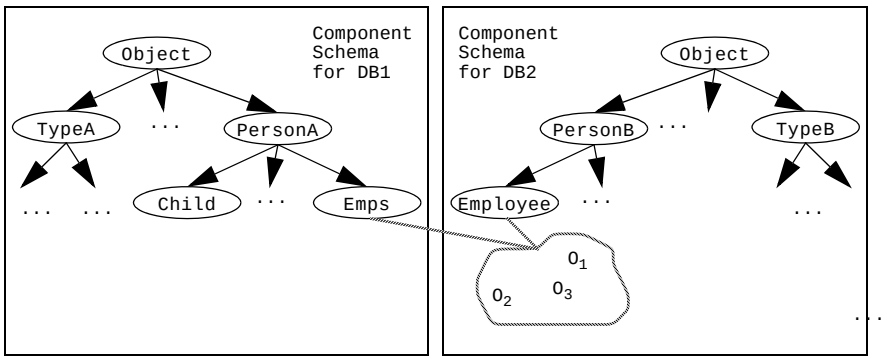


Figure 1.7: Instance sharing.

Suppose that two databases are being integrated as depicted in figure 1.7. Both databases contain information about persons. `DB1` contains information about children (`Child`) and employees (`Emps`) in particular. `DB2` also contains information about employees (`Employee`). The designer of `DB1` does not wish to create an extent for `Emps` himself but would like to use the extent defined for `Employee` in `DB2` (O_1, O_2 and O_3). This can easily be done by defining a derived type (class) involving a query retrieving the objects of interest from `DB2`.

Note: the component schema of `DB2` contains no information about the extent of `Employee` being used as the extent of `Emps`. Suppose that we use the meta data function `typesof:Object->Type` to retrieve all the types of an object. In `DB1` the types of O_1 are `Emps`, `PersonA`, and `Object`; in `DB2` the types of O_1 are `Employee`, `PersonB`, and `Object`. The same object has different types in different databases. Thus users must be aware that the same meta data function may return different answers for the same input depending on where it is executed.

Trying to define a formal model and to study the various anomalies similar to the one described above that can occur and how to best resolve them pose interesting questions for research.

2 Object Views and Queries for Integration of Heterogeneous Data Sources

Object views have been proposed as solutions to a number of problems such as *authorization* [Rabitti et al., 1991], *schema evolution* [Ra and Rundensteiner, 1995], and *integration* [Motro, 1987] and [Rundensteiner,1992]. In this chapter we examine some of the proposals as to how object views can be used for integration. However, we have limited ourselves to approaches similar to ours.

2.1 Superviews

A formal approach to integration is taken in [Motro, 1987]. A formal framework is defined and ten integration operators are defined. The operators can be divided into class hierarchy manipulation operators and attribute[1] manipulation operators. A good CDM, as noted in section 1.3.1, should support these operators or should be equally powerful allowing for similar restructuring of types, classes and functions.

Superviews provide a federated schema over different component schemas. Users define the federated schema in an interactive process where the integration operators are applied to the different component schemas. The sequence of operators applied is recorded and the system uses this information to transform queries to the federated schema to subqueries to the different component schemas. The answers are then combined to form the final answer to the global query.

Superviews organize objects in *classes*. Each class has a *type* which is the set of functions applicable to the instances of the class. Each class has also a *key* which is a subset of the type. As in all integration the literals are considered identical even if they come from different data sources.

1. Attributes in this case are stored or computed values associated with an object, i.e. attributes or methods in the object-oriented terminology. We will also use the term *function* in this section to denote the same concept.

Class Hierarchy Operators

The **meet** operator produces a common generalization of two classes. A common generalization is possible only when the two classes have a *common* key. By a common key the authors mean that the number of attributes constituting the key must be the same for both the classes. Furthermore, the corresponding attributes must be equally named and share the same domain.

Suppose that we have two types `Faculty` and `Student` that record information about faculty members and students, respectively, and that we want to generalize these two classes into a third class `Person`. The `Student` class has an `ssn` attribute, a `name`, and a `gpa` attribute. The `Faculty` class also has an `ssn` attribute, a `rank`, and a `name` attribute. We have depicted this in figure 2.1[2].

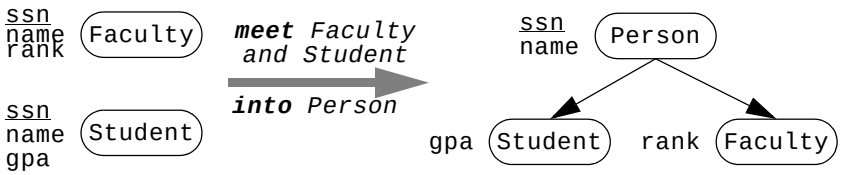


Figure 2.1: Example of the **meet** operator.

The key for each class has been underlined in the figure. As we can see `Faculty` and `Student` share the same key, namely `ssn`[3]. Apart from `ssn` they also share the attribute `name`. The **meet** operation raises all common attributes to the new superclass `Person`. Thus `ssn` and `name` will be associated with the new class `Person` whereas `gpa` will still be associated with `Student` and `rank` will still be associated with `Faculty`. Members of `Student` and `Faculty` sharing the same `ssn` are considered as the same person. Should a person be a member of both `Student` and `Faculty` then all common attributes are also required to have identical values; if not, the attribute is assigned a distinguished value *not consistent*. Members of `Person` will be the union between `Student` and `Faculty`.

While the **meet** operator constructs superclasses, the **join** operator constructs subclasses. Thus, **join** `Student and Faculty` **into** `Assistant` will add `Assistant` as a subclass to `Student` and `Faculty`. `Assistant` will now

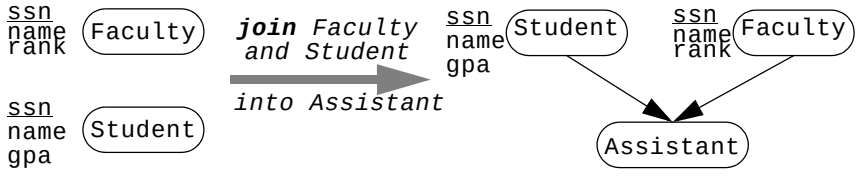


Figure 2.2: Example of the **join** operator.

2. Rounded boxes will be used to depict classes.
3. The domain for the different `ssn` attributes also has to be the same, e.g. ten-digit integers.

inherit from both `Student` and `Faculty`; the type for `Assistant` will thus be {`ssn`, `name`, `gpa`, `rank`}, i.e. `Assistant` inherits the union of the attributes of the superclasses. As in the case of **`join`**, the key has to be common and shared attributes have to have the same value. This avoids the problem associated with multiple inheritance, namely inheriting equally named functions from several superclasses with different implementations or values. If we inherit equally named attributes from the two superclasses we also know that they are bound to have the same values and therefore it does not matter which one we choose. Should the value differ for some multiple-inherited attribute, it is assigned the distinguished value *not consistent* as in the case of the **`meet`** operator.

While **`meet`** and **`join`** add new classes, **`fold`** removes classes. **`fold`** allows a class to absorb a subclass. With **`fold`** the class `Student` may be absorbed by the more general class `Person`. Any attribute associated with `Student` will be carried over to `Person`. Since the attributes acquired from the absorbed class

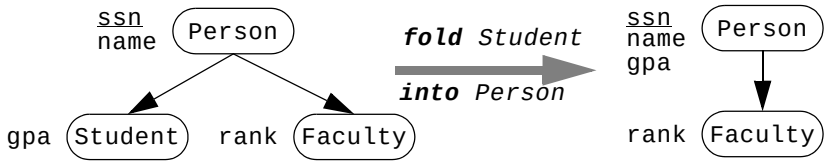


Figure 2.3: Example of **fold** operator.

are only meaningful for instances that once belonged to the absorbed class, other instances that acquired the attribute will have *null* as value for the acquired attributes. For example, in figure 2.3 the attribute `gpa` is acquired by `Person`. Only previous members of `Student` have a meaningful value for `gpa`. However, all `Person` members (and `Faculty` members) acquire the `gpa` attribute and, since they lack a meaningful value for the attribute, the value will be *null*.

There may be functions that had the absorbed class as their domain. Since it no longer exists we have to change the domain of the functions to the absorbing class instead. For example, if there existed a function `f` with `Student` as the domain before **`fold`**, it will have `Person` as domain after **`fold`**.

During integration it will sometimes be necessary to rename a class or an attribute; this is the task of the **`rename`** operator. **`rename`** `S` **`to`** `T` assigns the new name `T` to class or attribute `S`.

Two compound operators, **`combine`** and **`connect`**, are also defined. When two classes have identical types, **`combine`** merges them into a single class. When the type of one class S is contained in the type of another class T, i.e. the key of S is a subset of the key of T, then a **`meet`** followed by a **`fold`** can **`connect`** them into one class.

Formally:

- **combine** S and T into U is defined as
  meet S and T into U
  fold S into U
  fold T into U
- **connect** S to T is defined as
  meet S and T into U
  fold T into U
  rename U to T

Attribute Operators

As well as restructuring the class hierarchy, it may be useful to perform operations on the attributes of a certain class. To do this we use the attribute operators.

The operator **aggregate** replaces a number of attributes of a given class with a new attribute. An example illustrates the point. Given a class Person that

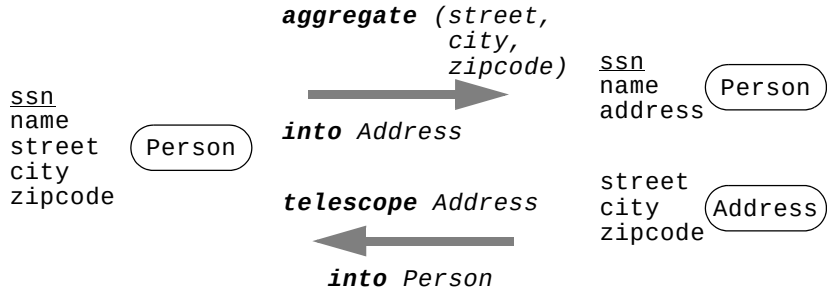


Figure 2.4: Example of the **aggregate** and **telescope** operator.

among other attributes has the attributes street, city, and name, aggregating these attributes into a new class Address removes them from Person and instead associates them with the new class Address. Person acquires the new attribute address whose value is a member of Address. Thus, to find out what city a person lives in, we take the value of address and then retrieve the value of city from the value of address.

Of course there exists an inverse to the **aggregate** operator, which is called **telescope**. Thus it is possible to reverse the effect of the aggregation performed in figure 2.4. **telescope** Address **into** Person removes the class Address and substitutes the attribute address for the attributes street, city, and zipcode for the Person class. The author requires Address not to be used as domain of any attribute not associated with the class it is being telescoped into, i.e. there may not exist another class Company that also has an address.

The final two operators are **add** and **delete**. New attributes are added to a class by the **add** operator. This can be very useful as whenever identical struc-

tures from different databases are combined, loss of information may result. Consider two library databases both with a class `Book` = {`book-no`, `title`, `author`}. If these classes are combined, the information on where each book is shelved would be lost. Using **`add`**, this implied knowledge can be added to each class as a new attribute `library`. **`add`** `library(1)` **`to`** `Book` and **`add`** `library(2)` **`to`** `Book` would add the attribute to the `Book` classes. The value for the `library` attribute is also specified in the **`add`** statement. The type of the two classes would now be {`book-no, library, title, author`}. Integrating them into a single class requires the `library` attribute to be added to the key since `book-no` may only be unique within one library. This is accomplished by **`add`** `library` **`to key of`** `Book`.

The **`delete`** operator removes attributes from classes. For example, **`delete`** `name` **`from`** `Person` would remove the name attribute from class `Person` in figure 2.4.

Summary

Superviews use a functional model and as a consequence structural difference cannot occur, thus simplifying the integration task. We have, however, chosen to present the concepts in terms of an object-oriented model. We also note that the approach taken by Superviews is not altogether satisfactory in an object-oriented model where a distinction is made between notions such as classes, types, functions (methods and attributes), and objects. However, in an object-oriented model it must be possible to perform restructurings similar to those of Superviews.

The problems of scaling conflicts, key conflicts, and constraint conflicts are recognized by the authors but not addressed.

The authors have concentrated primarily on integration and interrogation of integrated databases and not on updates. A short discussion is given concerning updates but the authors conclude that more research is needed.

The work is mainly theoretical and the authors make no suggestions on how to realise their work. One practical detail that has to be solved is how to merge members from different classes into a single member of a new class.

2.2 MultiView

MultiView [Ra and Rundensteiner, 1995] and [Rundensteiner,1992] is an implementation of object views on top of the GemStone OODBMS [Kuno and Rundensteiner, 1993]. It maintains multiple views of a global schema.

This is a problem relevant in multidatabase environments as well, since several export schemas may have to be defined for a given component schema as shown in figure 1.3 on page 6, or several external schema may have to be defined for a given federated schema.

Anything with distinct existence in objective or conceptual reality is represented as an *object*. Each object has associated with it a number of *instance variables* which hold the state of the object and *methods* that represent the behaviour of the object. A method consists of a *signature* (method name and argument and result types) and an *implementation*, a block of code specifying the behaviour of the method. Two methods in MultiView are considered equivalent if they share the *same* block of code.

A *type* is, as in Superviews, the library of methods and instance variables available to a given object. In MultiView, a *class* is composed of both a type and an *extent* (the set of all the object instances with that type). Every object posses at least one type, and is thus an instance of at least one class.

Classes can be categorized in two categories, *base classes* and *virtual classes*. The extent for base classes is explicitly stored whereas the extent for virtual classes is defined by a query expression.

MultiView organizes both base classes and virtual classes into a single class hierarchy. A class C_{sub} is a subclass of another class C if C *subsumes* C_{sub}, i.e. all of the methods and attributes contained in the type description of C must be included in the type of C_{sub}. Furthermore, the extent of C_{sub} is a *strict* subset of the extent of C. Each instance of the subclass is considered to be an instance of the superclass, too. Thus, in each context where an instance of the superclass is required, an instance of the subclass is also permitted.

Users have to explicitly declare subclass relationships for base classes whereas it is automatically derived by MultiView for virtual classes.

Subsumption in MultiView is based on implementation of methods and not signatures, i.e. if two classes C_1 and C_2 share some common property then they must ultimately have inherited it from the same superclass. If instead signatures were used, we would have to compute whether or not two different implementations model the same behaviour, which is clearly not computable.

The single class hierarchy in which both base classes and virtual classes are organized is called the *global schema*. This schema can become quite unwieldy as new classes are defined. Typically, users are interested in different subsets of the global schema. Such subsets can be defined and are called *view schemas*. A view schema is not affected by changes made to other view schemas and as the global schema evolves, the system will maintain a correct mapping between the different view schemas and the global schema.

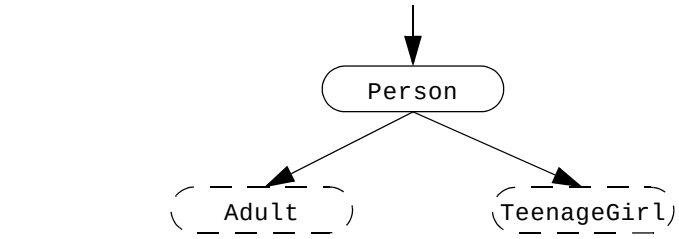
The query language used when defining virtual types is based on an object-preserving algebra, i.e. queries do not generate new objects. The algebra is set-oriented and has six operators.

- **Select**. The select operator defined by (**`select`** `<class>` **`where`** `<predicate>`) returns a subset of the input set of objects, `<class>`, namely those satisfying the predicate expression. The type of the resulting set is unchanged, i.e. it is equal to the type of `<class>`. As the extent of the virtual class is a subset of the extent of `<class>`, the virtual class will become

a subclass of <class>.

- **Hide**. The hide operator defined by (**hide** <properties> **from** <class>) removes properties listed in <properties> form the set of objects <class> while preserving all other properties defined for the type of <class>. The type of the output set is a supertype of the input type, as fewer functions are defined on the output. All objects of the input set are also members of the output set. This means that the virtual class will become a superclass of <class>.
- **Refine**. The refine operator defined by (**refine** <property-defs> **for** <class>) returns a set with the same objects as the input, but with a new type, a subtype of the input type, as all the old properties plus the new one are defined for it. It is required that each property in <property-defs> must be different from all existing properties of the <class>. Since the type of the virtual class is a subtype of the type of <class> and the extents are equal for the two classes, the virtual class will become a subclass of <class>.
- **Union**. The union operator defined by (**union** <class1> **and** <class2>) returns the union of the extent for the two classes. The criterion for duplicate removal is object identity. No restrictions are needed on the operand types since ultimately everything is an object. The resulting type, however, depends on the input types. For union it is the lowest common supertype of the input types. Since the type of the virtual class is a supertype of the types for both <class1> and <class2> and the extent of the virtual class is a superset of the extents for <class1> and <class2>, the virtual class will become a superclass of <class1> and <class2>.
- **Intersection**. The intersection operator defined by (**intersection** <class1> **and** <class2>) returns the intersection of the extents for the two classes. The resulting type of the virtual class will be the greatest common subtype of <class1> and <class2>. The virtual class will become a subclass of both <class1> and <class2> as the extent of the virtual class is a subset of the extents for <class1>and <class2>.
- **Difference**. The difference operator defined by (**diff** <class1> **and** <class2>) returns the set difference between the extent of <class1> and <class2>. The resulting type of the virtual type will be the same as the type of <class1> but the extent will be a subset. Thus, the virtual class will become a subclass of <class1>.

Let us look at an example. In figure 2.5[4] a global schema is depicted. It con-




```
class Adult =
  select (p:Person) where
     p.age>21;
```

```
class TeenageGirl =
  select (p:Person) where
     p.age<21 and p.age>=13 and
     p.sex=female;
```

Figure 2.5: Global schema.

tains one base class, `Person`, and two virtual classes, `Adult` and `TeenageGirl`. Now suppose a user defines two virtual classes `Minor` and `TeenageBoy` as

```
class Minor = select (p:Person) where p.age<21;
class TeenageBoy =  select (m:Minor)
                    where m.age>=13 and m.sex=male;
```

Example 1: Creating two virtual classes.

MultiView will then modify the global schema and insert the two virtual classes at their proper places in the class hierarchy as shown in figure 2.6. The global

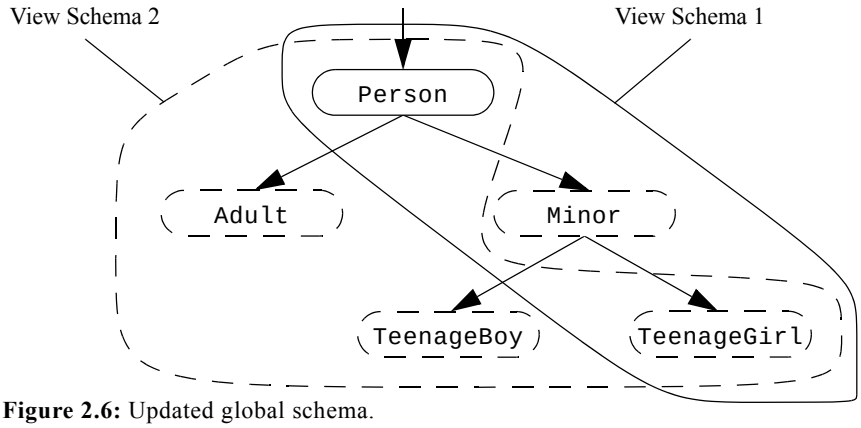


Figure 2.6: Updated global schema.

schema contains all the classes. However, users can select interesting classes for view schemas. A user only specifies what classes are of interest. MultiView then organizes the classes in a view schema. For example, for view schema 1 the classes `Person`, `Minor` and `TeenageGirl` have been selected. View

4. Rounded boxes with dashed borders depict virtual classes.

schema 1 will contain only these classes and they will be organized as in the global schema. However, class `Minor` has been excluded from view schema 2. MultiView will then make `TeenageBoy` and `TeenageGirl` direct subclasses of `Person`.

Since classes are shared between different schemas, updates of the extent of one class in one schema will be noticed in other schemas using this class. This allows applications to cooperate and the schema can evolve by means of views without interfering with this cooperation.

2.3 COOL*

In [Tresch and Scholl, 1994] the functional object database language COOL [Laasch and Scholl, 1993] using the object-oriented data model COCOON [Scholl et al., 1992], is extended with multidatabase facilities and renamed COOL*.

The COOL* language aloows users to choose what level of integration to use, up to level 3 integration, See “The Global Schema Approach” on page 13.

In the COCOON model information is modelled using *objects*, *functions*, *types* and *classes*. *Objects* are instances of *abstract object types* (*AOTs*). *Data* are distinguished from objects and are instances of *concrete data types*.

Functions model the AOT-specific operators. They are abstractions of side-effect free retrieval functions, called *properties*, and update *methods*[5] with side-effects. Properties can be either *stored* or *computed*. The computed properties can be subdivided into *derived* and *foreign* properties where derived properties are defined by a COOL* expression and foreign properties are defined in some general-purpose programming language.

Types are separated into concrete data types and abstract object types. *Concrete data types* are either primitive (e.g. integer, real, string) or constructed (e.g. tuple, set, function). *Abstract object types* describe the common interface of all instances of that type, the set of applicable functions. Types are organized in a type hierarchy according to the same principles as in MultiView.

Classes are typed containers for objects and they are organized in a class hierarchy. Again, the same rules apply to classes as in MultiView. Derived classes can be defined and are called *views*. An object-preserving algebra with the same operators as in MultiView is used when defining derived classes. The same rules also apply as to what type a derived class gets and where it should be placed in the class hierarchy.

Let us now see how integration is performed in COOL*.

Level 1 integration or *schema composition* is achieved by combining the schema of the different EDSs. Type and class systems of the EDSs are com-

5. Collectively called *functions*.

bined, without establishing connections between composite systems. As an anchor, concrete data types of EDSs are assumed to be identical. This ensures that at least values of elementary data types can be compared. Local object and type hierarchies of the EDSs are put together by defining a new global top type and a new global top class. COOL* has a type lattice; therefore, a new bottom type is also created that is made a common subtype of all local bottom types.

In [Tresch and Scholl, 1994] the following example is given. In a university environment information about students is stored in database *StudDB* and information about what books a student has borrowed is recorded in a library database, *LibDB*.

The following COOL* statement composes the three schemas into a new global schema *UnivDB*.

```
define database UnivDB
  import LibDB, StudDB
end
```

Example 2: Composing two schemas.

Queries can now be formulated that involve multiple EDSs. For example, since composition made basic data types and name spaces globally available, we can now compare the names of customers (from *LibDB*) with names of students (from *StudDB*).

```
select[∅≠select[name(c)=name(s)](s:Students)](c:Customers)
```

Example 3: Comparing name of students with name of customers.

The syntax for the select statement in COOL* is `select[bool-expr](set-expr)` where `bool-expr` is a boolean expression returning *true* or *false* and `set-expr` is a COOL* expression returning a set.

The above query retrieves all those students who are also known as customers. It would be convinient if we instead could have expressed it as

```
select[c∈Students](c:Customers)
```

Example 4: Selecting all students who are also customers.

Unfortunately, since objects of class `Students` are of type *Student* and the type of `c` is *Customer* and the two types are not yet related, the selection predicate would be rejected by the type checker. We first need to relate objects of the two classes to each other which requires level 2 integration.

Level 2 integration or *virtual integration* allows us to specify that certain objects are the *same*. This is accomplished by the use of *same*-functions. A same-function is partial, injective, single-valued function with the signature:

$$\texttt{same}_{\texttt{i,j}}\texttt{:object}_{\texttt{i}}\texttt{->object}_{\texttt{j}}$$

where `i` and `j` denotes databases. Global identity $=_{gl}$ of objects is then defined as

$$o1 =_{gl} o2$$
$$\Leftrightarrow$$
$$(\exists i{:}object_i(o1) \wedge object_i(o2) \wedge o1 =_i o2)$$
$$\vee$$
$$(\exists same_{i,j}{:}object_i \text{->} object_j{:}\ object_i(o1) \wedge object_j(o2) \wedge o2 =_j same_{i,j}(o1))$$

Two objects are the same if they stem from the same EDS or if they have been defined to be the same using *same*-functions.

If we wish to answer the query in example 4 on page 26 we have to say that students and customers are the same. We *extend* the type of the `Student` class with a *same*-function.

```
define view Students as
extend[6][same_StudDB,LibDB :=
        pick[7](select [name(c)=name(s)]
                (c:Customers))]
     (s:Students)
```

Example 5: Integrating objects of class `Students@StudDB`[8] with objects of class `Customers@LibDB`.

Customers and students with the same name will now be regarded as the same object. The type checker will now not reject the expression in example 4 on page 26.

Since functions are also objects they can be merged in the same way.

```
define view Functions@StudDB as
extend[sameStudDB,LibDB:=
        pick(select[fname[9](f)="NAME" and fname(g)="NAME"]
                (g:Functions@LibDB))]
     (f:Functions@StudDB)
```

Example 6: Unifying function `name@StudDB` and function `name@LibDB`.

Level 3 integration or *real integration* allows us to create functions whose domain and range are types from different databases. For example, we could define a function `favourite_book` as

```
define function
  favourite_book: Student@StudDB[10] -> book@LibDB
```

Example 7: Defining an inter-database function.

In level 2 integration we were not allowed to defined stored inter-database functions, though *derived* inter-database functions were allowed. Function

6. Corresponds to `refine` in MultiView.
7. The `pick` operator does a set collapse, returning the objects from a singleton set.
8. `@` is used as qualifier specifying which database the information is to be retrieved from.
9. `fname` returns the name of a function.
10. This is a type, *not* a class.

$\texttt{same}_{\texttt{StudDB,LibDB}}$ was an example of a derived inter-database function; it compares objects from two different databases.

Stored *same*-functions can also be useful. In [Tresch and Scholl, 1994] the authors discuss what should be done if a user tries to add a type to an object. In COOL* there is generic update operation `gain[`*t*`](`*o*`)` that adds type *t* to object *o*. If both *t* and *o* stem from the same database, then `gain` works as in a centralized database. However, should *t* and *o* come from different databases, the semantics becomes unclear. One realization of this `gain` operation would be to create a *same*-object *o'* of *o* in the database where type *t* is defined and a local `gain` operation is performed, making *o'* an instance of *t*. This realization maps the multi-database `gain` operation to a sequence of operation, that can be executed within one single EDS. Since an object *o'* of DB_j is assigned to be the same object as *o* of DB_i, stored *same*-functions are needed.

2.4 Pegasus

Pegasus [Ahmed et al., 1991a], [Ahmed et al., 1991b], [Ahmed et al., 1993], and [Rafii et al., 1991] is a prototype of an object-oriented multidatabase system developed at Hewlett-Packard. Pegasus is like AMOS based on the Iris data model [Fishman et al., 1989] and [Lyngbaek et al., 1991] and the HOSQL language is an extension of OSQL used in Iris. It has been extended to address integration at schema and data levels, and to deal with multidatabase operations.

The main difference between Pegasus and AMOS is that Pegasus is a "centralized" multidatabase system. A single Pegasus system integrates a number of EDSs. In AMOS we need not have a single AMOS server that integrates all the EDSs rather we can have a number of AMOS servers that integrate different EDSs and where the different AMOS servers interact with each other.

The Pegasus system manages databases. Two different kinds of databases are distinguished, *native databases* and *imported databases*. Native databases are created and managed by the Pegasus system. Information stored in native databases resides in the Pegasus system. Imported databases represent some external data source, such as a file system or a database managed by some other system.

Import

Creating an imported database involves, among other things, making the schema of the external data source available as a Pegasus schema. Pegasus provides two data definition facilities to do this: *imported producer types* and *imported functions*.

An *imported producer type* defines the existence of its instances according to a rule based on some identifying literal-valued property for each entity in an external data source. There will be one instance of the producer type for each

unique occurrence of the identifying property in the external data source. OIDs are fabricated for instances of produce types and OIDs generated for instances of two producer types will be distinct even if the same value for their identifying property occurs for both types.

```
create imported producer type Student
imported from Relational System
datasource SDB
relation Students
producing by (StudID)
functions (studentid Integer as identifier;
           ssnum Integer as map to SSNo);
```

Figure 2.7: Example of imported producer type.

In figure 2.7 an example of an imported producer type is given. We are constructing a Pegasus schema for a relational database called `SDB` containing information about students. In `SDB` there exists a relation `Students` that we wish to represent as a type `Student` in Pegasus. The primary key for `Students` is `StudID` and thus we use it as the identifying property.

In the process of defining the `Student` type we also define two functions. The function `studentid` is called an *identifier function*. It takes as argument the OID of a student and returns the integer value of his studentid. This value is a member of the producer set. The other function `ssnum` is an *imported function*.

Imported functions are mapped to properties or relationships in external data sources. For example `ssnum` takes a student as argument and returns his social security number found in the `SSNo` column of the `Students` relation in the `SDB` database.

Integration

As discussed in chapter 1 it is not enough to import external data sources; they need to be integrated as well. Pegasus addresses this issue as well by distinguishing between *underlying objects* and *unifying objects*.

Underlying objects result from the import process and faithfully reflect the external data sources, including any discrepancies. Unifying objects present an integrated view for the end user.

Suppose that we are integrating different databases containing student information. A given student may be represented in more than one of these databases. For each database we would create a distinct student type in Pegasus and the student would then be represented by several distinct objects of different student types in Pegasus. We would then define a *unifying type* where all these different objects were represented by the same *image object.*

```
create unifying producer type Student
over underlying types EStudent, WStudent
functions (ssnum Integer as identifier);
```

Figure 2.8: Example of a unifying type.

The form in figure 2.8 creates `Student` as a unifying type as well as a producer type, and designates existing types `EStudent` and `WStudent` as underlying types. The producer expression is *implicitly* defined by the system as:

```
producing by HOSQL(
  select ssnum(x) for each EStudent x
  union
  select ssnum(x) for each WStudent x);
```

Students occurring in both `EStudent` and `WStudent` will thus be represented by a single image object in `Student` since the producing expression is a union over `ssnum`. If `EStudent` and `WStudent` had been disjoint, then we would not have had to specify `Student` as a producer type since no unifying objects would need to be created. Instead it would have sufficed to define `Student` as a unifying type only and not as a producer type also.

Every type has exactly one unifying type, usually the type itself. Every instance object has at most one unifying object (or *image*), usually the object itself, given by the system function `image`. The `image` function is used to transform an object of an underlying type to the unifying object. By allowing the administrator of the Pegasus system to define suitable `image` functions, desired mappings can be obtained.

In figure 2.8 we integrated existing information but sometimes it is the other way around. We already have a type that we want to unify with new imported types. Suppose that a `Student` type already exists and is populated in the Pegasus system. We then want to integrate it with the two imported types `EStudent` and `WStudent`.

```
add underlying types EStudent, WStudent
under Student
(EStudent.Image(x) as stored)
(WStudent.Image(x) as HOSQL(
  select s for each Student s);
```

Figure 2.9: Integration of an already existing type.

Figure 2.9 illustrates how this is done. `WStudent.Image` is defined as

```
create function image(WStudent y)->Student z as
  HOSQL select z where ssnum(z)=ssnum(y);
```

For `EStudent` `image` is defined as a stored function, i.e. the image of an `x` in `EStudent` may be undefined (`null`) until a corresponding instance of `Student` is assigned. The image of an instance of `WStudent` is that instance of

`Student` having the same social security number. There might be none. Thus, we note that image mappings might be algorithmically or manually maintained.

Pegasus also addresses the problem of naming conflicts (section 1.4.1) by providing a construct that allows a user to rename a function. To make integration easier Pegasus provides *unifying inheritance*.

Unifying inheritance means that the unifying type inherits all the functions defined for the underlying types thus enabling the user to apply functions given the same names as the original ones to unified objects. However, *overload ambiguity* arises when a unified function resolves to more than one underlying function, i.e. two or more of the underlying types have equally named functions. This situation is resolved by an *ambiguity reconciler* defined by the user.

In [Ahmed et al., 1991b] and [Rafii et al., 1991] it is noted that it is also possible to specify equivalence classes of objects similar to COOL* (section 2.3). This obliterates the need to create producer types if we already have objects of different classes that we want to integrate. It would then suffice to given an expression defining the equivalence between the different objects representing the same real world entity. However, in later papers [Ahmed et al., 1993] this possibility is no longer mentioned; instead integration is achieved as described in this section.

The need for higher order views is recognized in [Ahmed et al., 1991b] and suggestions similar to those described in section 2.5 are made. This enables Pegasus to handle structural differences, section 1.4.3.

Key conflicts, section 1.4.5, as we understand it, are handled by providing a suitable **`producing by`** expression when specifying the producer type.

Behaviour conflicts, section 1.5.1, are handled by defining suitable functions resolving the conflicts.

It has not been possible to verify to what extent the various features described have been implemented in Pegasus. The different concepts have been presented in various ways in different papers. However, we do know that an implementation exists and that it is probably one of the more advanced multidatabase prototypes in existence.

2.5 Parameteriezed Views

In [Chomicki and Litwin, 1994] the problem of structural differences is addressed, see “Structural Differences” on page 10. They discuss extensions of the object-oriented database language, OSQL, needed to resolve these differences as they are inevitable in a multidatabase environment.

In the paper, the authors distinguish between four different modelling constructs, *literals*, *object*, *types*, and *functions* defined in the same way as in section 3.1. Literals are self-identifying objects such as integers and strings.

Since information may be modelled in different databases using any of these

four constructs, users must be able to map between them. Thus, 16 different mappings are described.

The concept of derived types is defined similarly to derived classes in Multi-View and COOL*. However, this concept is then generalized into *type schemas*. A type schema defines a family of derived types.

For example, if we have a database containing information about employees and their titles then the following type schema would define as many derived types as there are titles.

```
create type EmpPos[String s] as
  select e for each Employee e
  where title(e)=s;
```

Example 8: Defining a type schema `EmpPos`.

Assuming that two employees existed with title `"Secretary"` and `"Boss"` respectively, two derived types named `EmpPos["Secretary"]` and `EmpPos["Boss"]` would be created. Type schemas themselves are not types, they only create types called *instance types*.

Type schemas provides great flexibility as users do not have to define a new derived type whenever a new title is added. The system will automatically create a new type.

The correspondence to *producer types* in Pegasus (section 2.4) and *mapped types* (section 4.3) in AMOS are called *object schemas*.

To complete the framework the authors define *functions schemas*. Function schemas will map values, objects, types, and functions into new functions.

Assume the existence of a function `Stock:Date x String->Price` that records the closing price for companies each day. We can then create a function schema defining functions returning the closing price for each company given a date.

```
create function closing_price[String c](Date d)->Price as
  select p for each Price p
  where stock(d,c)=p;
```

Example 9: Defining function schema closing_price.

As in the case of type schema function schemas are not functions but they define functions called *instance functions*.

Function schemas and type schemas enable us to ask higher order queries such as

```
select nm
  for each String nm, Function closing_price[nm], Date d
  where closing_price[nm](d)>$200;
```

Example 10: Retrieving the name of all companies whose stocks have ever closed above $200.

Function schemas and type schemas causes some problems. For example, it is possible to define an infinite number of types.

```
create type YoungerThan[Integer k] as
  select e for each Employee e
  where age(e)<k;
```

Example 11: A type schema with an infinite number of instance types.

This and a number of other problems described in [Chomicki and Litwin, 1994] make it impossible to maintain an explicit type hierarchy and thus partially obstructs inheritance.

2.6 Concluding Remarks

We have given a brief review of related work in the area of integration using object views. We first looked at *Superviews*. Superviews provide ten operators for information restructuring. Five operators for restructuring the class hierarchy and five operators working on attributes. Using these operators users can merge classes, create new superclasses, and create new subclasses of existing classes. Also, it is possible to create new attributes of existing classes and hide already existing attributes.

Next we examined *MultiView* an object view management system built on top of GemStone. It allows users to change *view schemas* defined over a global schema without affecting other view schemas. The system will maintain the correct mappings as changes are made.

In MultiView we can do more than we could in Superviews. Superviews did not let us create virtual classes where the class contained only a subset of some other class, as in example 1 on page 24. However, compared to Superviews we have to perform more operations. For example, if we want to merge two classes into one, we can do it by defining a common superclass. We then have to create a new view schema excluding the two classes that we merged, while including the common superclass. In Superviews a special operator was available for doing this.

*COOL** can be seen as an extension of MultiView for a multi-database environment. The same operators are defined in COOL* for defining virtual classes as in MultiView. However, in COOL* the problem of object-equivalence is addressed. This problem has to be solved in an object-oriented multidatabase system. In COOL* it is possible to define *same*-functions stating when two objects are to be treated as the same object.

Superviews and Multiview concentrate on reorganizing classes of existing objects while *Pegasus* also lets users define classes that generate new objects, so-called *producer types*. This allows users to create object views of non-object-oriented data. The same could be accomplished in COOL* by means of stored *same*-functions and an operation for creating objects. However, in [Tresch and Scholl, 1994] this is not discussed.

Finally we reviewed a proposal for parameteriezed views. In a multidatabase environment it is crucial that we can map between the different constructs used in the canonical data model, or we will not be able to resolve all structural differences that can occur. Parameteriezed views let us do this. However, they cause some problems as it is possible to define an infinite number of types and functions.

The ideal object view management system in a multidatabase environment would combine the view mechanism of MultiView with *same*-functions from COOL*. Finally we would add parameteriezed views.

3 AMOS Data Model

In this chapter we will present the data model of AMOS and AMOSQL [Flodin et al., 1996], the query and data manipulation language of AMOS. The data model of AMOS is strongly influenced by the functional data model OODAPLEX [Dayal, 1989] and by the Iris data model [Fishman et al., 1989] and [Lyngbaek et al., 1991]. We also present a multidatabase extension of AMOSQL.

3.1 Types, Functions and Objects

The AMOS data model has four basic modelling constructs; *objects*, *types*, *functions* and *rules*. The relationship between objects, types and functions can be seen in figure 3.1.

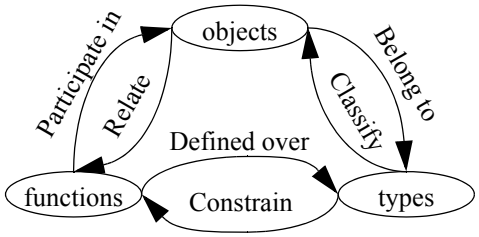


Figure 3.1: Functions, types and objects.

Objects are used to model entities in the domain of interest. *Types* are used to classify objects and act as containers for their instances. All objects are instances of some type. *Functions* are constrained to accept only objects that are instances of the declared argument type of any subtype thereof. They are used to model properties of objects and relationships between objects. Finally, *rules* are used to define constraints.

3.1.1 Types

Types can be divided into *literal* types and *surrogate* types based on their extent. The extent of a literal type is fixed (often innumerable), and instances of literal types are self-identifying, i.e. no object identifiers are needed to distinguish between instances of a literal type. However, notice that literals are objects as well. Examples of literal types are integers, reals, and strings. Instances of surrogate types are created by the system or by users and they are uniquely identified by an immutable system generated *object identifier (OID)*.

In multi-database AMOS surrogate types can be further divided into three categories; *stored*, *derived* and *mapped* types. Stored types have an extent explicitly stored in the database whereas the extents of derived and mapped types are expressed by declarative queries. In this section we will only look at stored types. Derived and mapped types are multi-database extensions of AMOS that will be presented in section 4.2, and section 4.3, respectively.

Types are organized in a type hierarchy as shown in figure 3.2. The most general type is `Object`; all other types are subtypes of `Object`. User-defined types are subtypes of a special type called `UserTypeObject`.

All objects representing types are instances of type `Type`. User-defined types are also instances of the type `Usertype`.

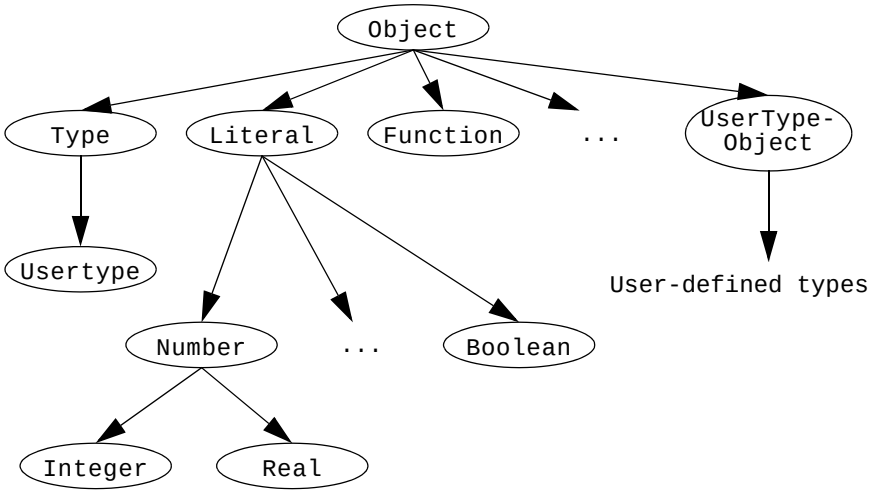


Figure 3.2: Part of the AMOS type hierarchy.

Organizing types in a hierarchy lets us define the notion of *comparable* and *incomparable* types. Comparable types are related via a sub-/supertypes relationship whereas incomparable types are not. For example, in figure 3.2 types `Number` and `Real` are comparable whereas `Number` and `Boolean` are not.

AMOS supports *inclusion polymorphism* [Cardelli and Wegner, 1985], i.e. types are ordered in the hierarchy using *set inclusion*. This means that an object *o* instance of a type *t* is also an instance of any supertype, t_{sup}, of *t*. Consequently, all functions defined for t_{sup} are also applicable to instances of *t*. We could also say that *t* inherits all functions defined for t_{sup}. The following statement expresses this and we require that it always holds[1]:

$$extent(t) \subseteq extent(t_{sup}) \ \wedge \ applicablefns(t_{sup}) \subseteq applicablefns(t)$$

Let us look at an example to clarify this. In figure 3.3 the type hierarchy is

1. extent(t) denotes the extent of type t and applicablefns(t) denotes the set of functions that are applicable to objects of type t.

shown for a database recording information about people and their employment. Next to the types, functions for each type are shown. For type `Person` two functions, `name` and `age`, are defined. However, `name` and `age` are applicable to any instances of `Employee`, `Secretary` and `Boss` as well, as they are persons, too.

The notion of being able to use an object, *o*, of type *t* in any context specifying that an object of type t_{sup} should be used has been termed *substitutability* [Shaw and Zdonik, 1989].

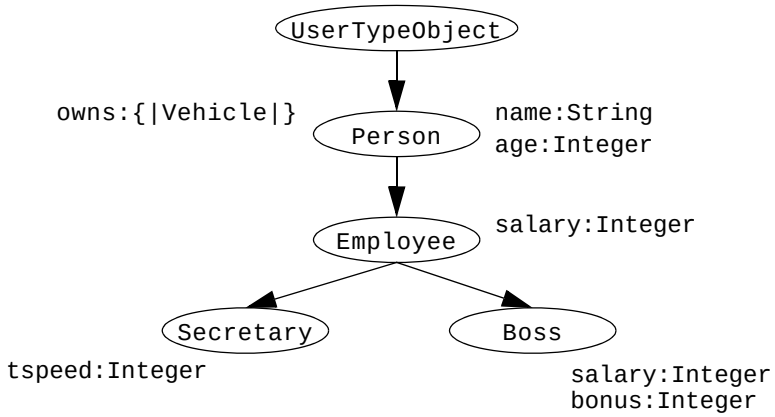


Figure 3.3: Example type hierarchy.

We also note that AMOS supports *multiple inheritance*, i.e. a type *t* can have several *immediate supertypes*.

3.1.2 Functions

There exist three different kinds of functions in AMOS; *stored*, *derived* and *foreign functions*. As with types, functions have an extent, which is the relationship between arguments and results of a function. For stored functions, the extent is stored directly in the database. A derived function uses the AMOSQL query language to calculate the extent, whereas foreign functions are implemented in a general programming language, such as C or Lisp.

Each function has a *name* and a *signature*. The name need not be unique but the signature has to be. For example, the function `salary`, defined for type `Employee`, has the signature `salary:Employee->Integer`[2]. As can be seen in figure 3.3 the function name is not unique as there exists another `salary` function for type `Boss`.

AMOS allows functions to be *overloaded*. Overloading means that there exist

2. Employee.salary->Integer is the equivalent of salary:Employee->Integer. The first notation is used in the current AMOS implementation. We will use the two notations interchangeably.

several implementations for a given function name. Inheritance combined with overloading allows us to *override* inherited functions. Consider again the `salary` function defined for `Employee`. The signature of the salary function applicable to secretaries is `salary:Employee->Integer`. However, for type `Boss` another `salary` function is defined that overrides the inherited salary function. This new function has the signature `salary:Boss->Integer`.

When an overloaded function name is used, the right function is chosen by the system by looking at the types of its arguments and results. This is called *function name resolution* and the chosen function is called the *resolvent*. It should be noted that functions can be overloaded on all argument and result types. Thus, the coexistence of two functions with the following signatures is allowed: `exchange_rate:Dollar->Franc` and `exchange_rate:Dollar->Mark`.

Multiple inheritance combined with overloading causes some problems. If a type inherits two or more functions with the same name but different signatures, then we do not know which resolvent is the correct one given just the function name. Users then have to explicitly state which resolvent to use by giving the signature of the resolvent, i.e. *explicit disambiguation* [Ameli and Dujardin, 1995].

Inheritance combined with function overriding necessitates the use of *late binding*. As stated above, the extent of a type t also contains all instances of any subtype t_{sub} of t. Thus querying the objects of the `Employee` type in figure 3.3, means querying all instances of the `Secretary` type and the `Boss` type as well. Suppose that we want to know the name of each employee. Since there exists only one definition of the function `name` the right resolvent can be chosen before execution. This a priori selection is possible since the definition is applicable to all objects that are instances of the types being queried. This is called *early binding* or *compile-time type resolution*.

If, on the other hand, there exists more than one definition of a particular function for the queried types, the definition to be used is dependent on the type of the object that the function is applied to. Thus, the function definition cannot be selected until the time of application. This is called *late binding* or *run-time type resolution*. An excellent treatment of late binding in AMOS can be found in [Flodin, 1996].

For example, if we had asked for the salary of all employees, the `salary` function would be late bound since we would have to know the type of the object before knowing whether to apply `salary:Employee->Integer` or `salary:Boss->Integer`.

Functions can be *single-valued* or *multi-valued*. A single-valued function can only take on a single result whereas a multi-valued function may return many. All functions in figure 3.3 but `owns` are single-valued. We use the $\{|\ t\ |\}$ notation to denote multi-sets. The t denotes the element type of the collection.

3.1.3 Procedures

AMOS also allows users to create procedures. The procedural language consists of the data manipulation language, the query language, and control flow constructs. The control flow constructs are a block construct, which allows sequencing of AMOSQL statements, and an if-statement. This guarantees that the ordering of the statements will not be changed by the optimizer in AMOS. There also exists a result-statement which allows us to specify a value to be returned by a procedure. The return statement, in contrast to return statements in most programming languages, returns a value but does not return from the procedure. By specifying several return statements in a procedure it can return several values. Conceptually we return from a procedure when we reach the end of it.

3.2 The Data Manipulation Language

In this section we will present the *data manipulation language* (*DML*) of AMOS. We will show how we define the example database whose type hierarchy is depicted in figure 3.3 on page 37. In section 3.3 we will then illustrate AMOSQL, the query language of AMOS, using the database defined in this section.

Let us first look at how to create stored types. Suppose that we start by defining the `Employee` and `Secretary` type. For each employee we should record the salary and for secretaries we should record the typing speed too. This is accomplished by the statements below[3].

```
create type Employee properties (salary Integer);
create type Secretary subtype of Employee
  properties (tspeed Integer);
```

Example 12: Creating two types.

We now have created two types with associated properties. A property is simply a function that takes as argument an object of the type being created and returns an object of the result type specified. So in the case of the `Employee` type the function `salary:Employee->Integer` was created. Thus, it is possible to add properties to types after the creation of the type by using the `create function` statement described below. This is needed since using the `properties` construct only allows us to specify stored properties (functions). If we need a derived property (function) then we have to specify it afterwards. This is the case for the `salary` function for type `Boss` as we will see.

Note that `salary` will be inherited by `Secretary` as `Secretary` is a subtype of `Employee`.

As well as supporting specialization, as we saw above, the `create type` statement supports generalization. We want the more general type `Person` as super-

3. The typing speed is recorded by `tspeed`.

type of `Employee`. This is accomplished by;

```
create type Person supertype of Employee
  properties (name String, age Integer);
```

Example 13: Creating a more general type.

If no subtypes or supertypes are specified for the new type when it is created, AMOS will also create the new type as a subtype of `UserTypeObject`. If, when adding a new supertype to existing types as in the case of `Person`, no supertypes are specified for the new type, then AMOS will insert the new type as a subtype of the most specific common supertypes of the specified subtypes. An example is shown in figure 3.4.

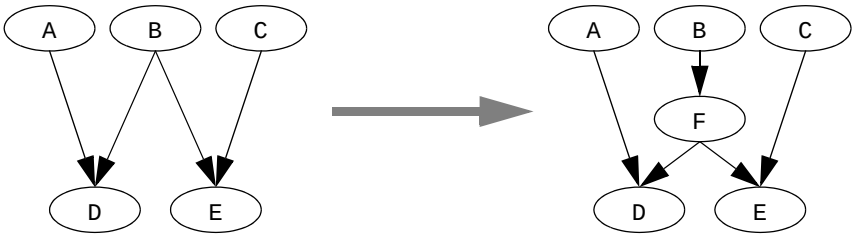


Figure 3.4: Effect of **`create type F supertype of D,E;`**

Should the user desire another placement of the new type, (s)he has to specify this explicitly using the full syntax of the `create type` statement shown below.

```
create type <type name>
  [ subtype of <subtype list> ]
  [ supertype of <supertype list> ]
  [ properties <property list> ]
  ( initializor <initializor decl.> )*[4]
  ( constructor <constructor decl.> )*;
```

Example 14: `create type` statement syntax.

Apart form defining the type, the `create type` statement also creates two functions, a *type extent function* and a *type predicate function*. The type extent function is a function with no arguments that returns the extent of the type as in DAPLEX [Shipman, 1981]. The type predicate function is a function of one argument that returns `true` if the argument is an instance of the type, otherwise it fails. Thus, for type Person above the type extent function has the signature `person:->Person` and the type predicate function has the signature `personp:Object->Boolean`.

Let us now create the remaining types to complete the schema of our example database.

```
create type Boss subtype of Employee;
```

4. * means zero or more occurrences.

```
create type Vehicle properties (model String);
```

Example 15: Creating the last two types.

When we created type Boss we did not specify any properties. Thus, we have to add them afterwards using the `create function` statement. The salary for a boss is the salary that (s)he has as an employee plus a bonus. Thus, `salary:Boss->Integer` is a derived function. Let us first create the stored function `bonus`.

```
create function bonus(Boss)->Integer as stored;
```

Example 16: Creating a stored function.

The `as stored` key words specify that the extent of the function will be explicitly stored in the database. Derived functions are created in the same manner.

```
create function salary(Boss b)->Integer as
  select Employee.salary->Integer(b)+bonus(b);
```

Example 17: Creating a derived function.

As pointed out in section 3.1.2 the extent of a derived function is not explicitly stored in the database but computed by the query expression. Note that we specify that the function `salary:Employee->Integer` should be used in the definition of `salary:Boss->Integer`. If we had not done this, but instead written just `salary(b)+bonus(b)`, then AMOS would have assumed that `salary(b)` should be resolved to `salary:Boss->Integer` because `b` is declared to be of type `Boss`. We would then have defined a recursive function that unfortunately would not terminate. AMOS does allow recursive functions but it is the responsibility of the user to guarantee termination of such functions.

Having created the schema of the database, we can now populate it. To create instances of a type we use the `create instances` statement.

```
create Person(name,age) instances
  ('Joe', 67), :george('George', 55);
```

Example 18: Creating instances.

The objects created will be instances of the specified type (and all its supertypes). In example 18 two instances of type `Person` are created. Immediately after the type name a property list can be specified. This gives us the opportunity to provide initial values for the specified properties. We should also note the use of an interface variable, `:george`, in the example. It will be assigned to the object representing George and gives us a handle that we can use in queries or when performing other operations.

For example, suppose that we want to promote George to boss. We can accomplish this with the `add type` statement:

```
add type Boss(Emloyee.salary->Integer, bonus) to
  :george(25000, 15000);
```

Example 19: Adding a type to an instance.

We next use an interface variable to specify which object we wish to add the type to. As in the case of the `create instances` statement we can specify a property list and initial values for the properties. Again we have to specify which salary resolvent function to use or AMOS would try to set the value of the derived function `salary:Boss->Integer`[5].

Having made George a boss, all the functions applicable to instances of type `Boss` are now applicable to George.

We also have the ability to remove a type from an object. This is done with the `remove type` statement. Suppose that we have created a boss `:john` as:

```
create Boss(name, age, Employee.salary->Integer, bonus)
  instances :john('John', 33, 17000, 10000);
```

Example 20: Creating a boss.

We now want to demote John to just an employee. We then write;

```
remove type Boss from :john;
```

Example 21: Removing a type from an object.

Having removed type Boss from `:john`, `bonus` is no longer applicable to the object nor is `salary:Boss->Integer`. However, all functions associated with type `Employee` are still applicable.

When adding a type *t* to an object *o* using the `add type` statement, *o* is made an instance of *t* and all supertypes of *t*. When removing a type *t* from an object *o* using the `remove type` statement, *o* loses type *t* and all subtypes of *t*.

Let us now define a secretary:

```
create Secretary(name, age, tpseed) instances
  :alice('Alice', 26, 150);
```

Example 22: Creating an employee.

When we created Alice in example 22 we forgot to set her salary. We can do this afterwards by using the `set` statement. The `set` statement updates the value of a function. If we want to give Alice a salary of 13000 we write;

```
set salary(:alice)=13000;
```

Example 23: Setting the salary.

All examples of functions so far have been single-valued. However, AMOS allows multi-valued functions as well. We have yet to create the `owns` function for type `Person`. This is an example of a multi-valued function.

5. AMOS allow updates of some derived functions. However AMOS would not be able to infer the correct update action in this case.

```
create function owns(Person)->bag of Vehicle as stored;
```

Example 24: Creating a stored multi-valued function.

The key words `bag of` means that given an argument the result may be *non-unique*. If the `bag of` is omitted then AMOS requires that each argument corresponds to *at most* one result[6].

Let us now create some vehicles for people to own.

```
create Vehicle(model) instances :rolls('Rolls Royce'),
  :bmw('BMW'), :saab('SAAB'), :volvo('VOLVO');
```

Example 25: Creating vehicles.

Setting the value of a multi-valued function is somewhat different. If we just want a person to own one vehicle we use `set` as usual;

```
set owns(:alice)=:saab;
set owns(:john)=:volvo;
```

Example 26: Assign a single result to a multi-valued function.

However, if we wish to add more results then we have to use the `add` statement. The first result of the function is assigned with `set` whereas the rest of the results are assigned with `add`.

```
set owns(:george)=:rolls;
add owns(:george)=:bmw;
add owngs(:george)=:saab;
```

Example 27: Assigning multiple results to a multi-valued function.

If we wish to remove the result from single valued and multi-valued functions we use the `remove` statement.

```
remove owns(:george)=:saab;
```

Example 28: Removing a result from a stored function.

3.2.1 Constructors and Initializors

When adding new instances to a type one might want certain actions performed, e.g. some properties should be given default values. *Constructors* and *initializors* allow us to specify what actions to perform.

Constructors and initializors prove useful in combination with mapped and derived types as they make it possible to create instances of these kind of types, See "Creating Instances" on page 66. Constructors and initializors also facilitate adding new foreign data structures as types in AMOS.

Constructors are called by the `create instances` statement and they are responsible for creating a new object and making the object an instance of the

6. Functions are partial since no result is returned unless the function has been given a value for the specified argument.

specified type. Constructors are implemented using the procedural part of AMOSQL.

Initializors are called by the `add type` statement and they are responsible for making the specified object become an instance of the specified type. Adding a new type to an object is very similar to creating a new instance of the type using the `create instances` statement. The only difference is that the object already exists when we use `add type` whereas it has to be created when `create instances` is invoked. Most likely we want similar actions performed in both cases.

Constructors are a well-known construct in object-oriented languages such as C++ [Strostrup, 1991]. The database community has begun to realize their importance and in the ongoing standardisation work on SQL3, abstract data types with constructors are proposed. However, initializors have not been proposed as far as we know, most likely because most object-oriented languages used lack the `add type` statement.

Whenever a type is defined AMOS provides a *default constructor* for the type, and a *default initializor*. Let us return to type `Employee` defined in example 12 on page 39. That declaration is equivalent to the following one;

```
create type Employee properties (salary Integer)
  initializor (Object o) as
      begin
          make_instance("Employee", o);
      end
  constructor () as
      begin
          declare Object o;
          set o=create_object();
          add type Employee to o;
          result o;
      end;
```

Example 29: Type declaration with explicit default constructor and initializor definition.

The default constructor takes no argument and creates an object of the specified type. It does so by calling the `create_object:->Object` function which creates a new object. However, the new object needs to acquire type `Employee` as well. This is accomplished by using the `add type` statement. The `add type` statement will invoke the default initializor for type `Employee`, and the default initializor will add type `Employee` to the object. The default initializor receives the objects through its only parameter, `o`, and makes `o` an instance of `Employee` by calling `make_instance` that actually adds the object to the extent of `Employee`. However, `make_instance` first calls all default initializors of supertypes of `Employee` beginning with the initializor of the most general supertype and ending with the least general one.

Note that the initializor does not return the object. It merely performs side effects. The constructor must, however, return the object since the object is created by it. For every stored type a default constructor and a default initializor of this kind are provided by the system.

If users need to declare new initializors or redefine initializors for existing types, then this is accomplished through the `create initializor` statement.

```
create initializor Employee (Object o) as
  begin
      make_instance("Employee", o);
      set salary(o)=13000;
  end;
```

Example 30: Redefining the default initializor for type `Employee`.

For example, the new default initializor for type `Employee` above would automatically set `salary` for any new instance of `Employee` to 13000. Since initializors perform side effects, such as setting the value of certain properties, it is important that the initializor for a type *t* is invoked on an object *o* only if *o* is not an instance of *t* already. If we were to invoke the initializor anyway, we might overwrite the value of some property.

A `create constructor` statement also exists with similar syntax.

Note that property values specified in the `create instances` statement and `add type` statement as in example 18 on page 41 and example 19 override any default values assigned by constructors and initializors.

Observe that we redefined the default initializor in example 30 and not the default constructor. Had we redefined the constructor to set `salary` then only employees created by the `create instances` statement would have had a default salary assigned. By redefining the default initializor both employees created using the `create instances` statement and employees created through the `add type` statement get a default salary.

Constructors and initializors can be parameterized and any number of constructors and initializors can be specified for a type. Each constructor and initializor for a type has a unique signature, i.e. specifying a constructor with identical signature as an already existing constructor means that the old constructor will be replaced by the new one, likewise for initializors.

Constructors and initializors can be overloaded. Suppose that we have defined the types in figure 3.5. Furthermore suppose that type `Child` is defined as shown in example 31.

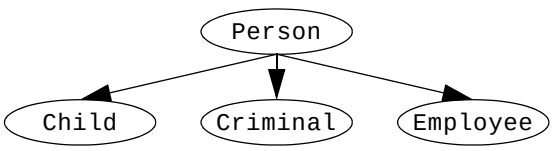


Figure 3.5: Example type hierarchy.

```
create type Child properties (behaviour String,
                              parent bag of Person,
                              age Integer)
  initializor (Object o, Person p1, Person p2) as
      begin
          make_instance("Child", o);
          set behaviour(o)="Good";
          set parent(o)=bag(p1,p2);
      end
  initializor (Object o, Criminal c1, Criminal c2) as
      begin
          make_instance("Child", o);
          set behaviour(o)="Bratty";
          set parent(o)=bag(c1,c2);
      end
  constructor (Person p1, Person p2) as
      begin
          declare Object o;
          set o=create_object();
          add type Child to o[p1,p2];
          result o;
      end;
```

Example 31: Definition of type `Child`.

It is a well-known fact that children that have criminal parents are brats. Another well-known fact is that if at least one parent is not criminal then the child will be well-behaved. Suppose that we have four different persons `:e`, `:c1`, `:c2`, and `:c3`. Person `:e` is an instance of `Employee` whereas `:c1` to `:c3` are criminals. We now create two children, `:child1` and `:child2` (example 32). The `create instances` statement will invoke the non-default constructor of `Child` since arguments are supplied. Arguments to the constructor are supplied within square brackets[7]. The constructor creates an object and then makes it an instance of `Child`, as indicated by the `add type` statement. The `add type` statement will invoke a non-default initializor as arguments are supplied within square brackets[8].

```
create Child(age) instances
  :child1[:e,:c1](7),
  :child2[:c2,:c3](9);
```

Example 32: Invoking non-default constructors

When invoking a non-default initializor AMOS looks at the types of the arguments supplied to perform initializor resolution[9]. When creating `:child1`,

7. If no arguments are specified using the square-bracket notation, then the default constructor is invoked.
8. If no arguments are specified using the square-bracket notation, then the default initializor is invoked.
9. Constructor and initializor resolution is performed in the same manner as function resolution.

`initializor(Object,Person,Person)` will be invoked since `:e` is not a criminal but both `:e` and `:c1` are persons. For `:child2 initializor(Object,Criminal,Criminal)` will be invoked since both `:c2` and `:c3` are criminals. After the completion of the statement in example 32 `:child1` will be aged 7, will be well-behaved and have one employee and one criminal as parents whereas `:child2` will be aged 9, behave brattily and have two criminal parents.

Note that `create_object` and `make_instance` exist primarily to allow definition of constructors and initializors. They are not supposed to be called by the user directly. If a user wants to create new instances of a type or add a type to an already existing object, then (s)he should use the `create instances`, and `add type` statements, respectively.

3.3 AMOS Query Language

The query language is syntactically similar to SQL but has a different semantics. Functions can be called directly without being embedded in a select statement. This is called *navigational access* and is one of the two different ways in which object-oriented databases can be accessed.

For example if we want to retrieve the salary of John we simply write;

```
amos 1>[10] salary(:john);
17000
```

Nested function calls have DAPLEX semantics, i.e. when a function is called with a bag as argument, the function is applied to all the members of the bag (one at a time). The result of the function call is the union[11] of all the results of applying the function to the different bag members. Consider the following example;

```
model(owns(:george));
```

Example 33: Example of nested function calls.

The result of applying owns to `:george` is a bag of two tuples:

```
{|<:rolls>, <:bmw>|}
```

When `model` is called with this bag as argument, it is first applied to `:rolls` which gives the bag `{|<"Rolls Royce">|}` as result and then to `:bmw` which gives the bag `{|<"BMW">|}` as result. The final result of the nested function call therefore is:

```
{| <"Rolls Royce">, <"BMW">|}
```

which in turn will be printed by AMOS as

```
"Rolls Royce"
```

10. This is the prompt of the AMOS system.
11. Bag union, i.e. duplicates of the same element are preserved.

```
"BMW"
```

Aggregation operators have a different semantics. When called, an aggregation operator is applied once on all the members of the bag, not once for each member. Examples of aggregation operators are `sum`, `max`, `min`, and `unique`. `sum`, `max` and `min` have their usual meaning. `unique` removes duplicates from a bag, i.e. it turns a bag into a set.

A function is given this kind of semantics if the type if its argument is declared as '`bag of ...`'. For example:

```
create function sum(bag of Integer)->Integer as ...
```

For more general queries we have to use the `select` statement with the following syntax:

```
select <result>
  for each <type declaration for local variables>
  where <condition>
```

The `select` statement provides *declarative access*, which is the second way to access object-oriented databases. Using declarative access a user specifies *what* information should be retrieved, not *how* it should be done.

The semantics of `select` is, for each possible binding of variables declared in the `for each` clause, to evaluate the `<condition>`. If `<condition>` evaluates to *true*, then issue `<result>`. Types with infinite extent are allowed in `for each` clauses.

`<condition>` is an expression built from function calls and the logical connectives `and` and `or`. Infix operators such as `+`, `=`, and `<` are ordinary functions that the parser will translate to prefix form, e.g. `4+5` will become `plus(4,5)`.

Suppose that we want to retrieve the name of all persons who earn 17000. The following query would appear thus:

```
select name(p) for each Person p where salary(p)=17000;
```

Example 34: Declarative access.

This query would return `"John"`. Observe that `salary` has to be bound late since we have two possible resolvents to choose from. The example also illustrates that functions can be used in the *backward direction*. When we earlier retrieved John's salary, we used the `salary` function in the *forward direction*. Now we have specified the result of the `salary` function and we want to know what arguments produce this result.

Negation in AMOSQL is handled through the aggregation operator `notany`. `notany` succeeds if it is given the empty bag as argument. For example, suppose that we want to retrieve all persons who do not own a vehicle. In AMOSQL we express it as:

```
select p for each Person p where notany(owns(p));
```

Example 35: Query containing negation.

This query would return `:joe` given our example database.

3.3.1 Multidatabase Extensions

AMOSQL also contains extensions allowing users to access information residing in other database than the local one. All function calls and type identifiers can be postfixed by `@` and an AMOSQL expression evaluating to an instance of type `Datasource` (fig. 3.6).

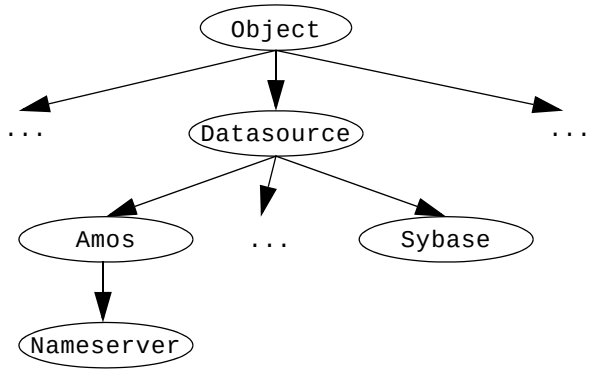


Figure 3.6: Data source subtype hierarchy.

Typing in OIDs for various data sources is not something that users are likely to do. They prefer some simpler method to denote a certain data source. Again interface variables come in handy. When a user finds a data source (s)he wants to use frequently, the object representing the data source is simply stored in an interface variable.

Suppose that we want to know all types defined in the AMOS denoted by `:EmpDB`. We would then submit the following query;

```
select tp for each Type@:EmpDB[12];
```

Example 36: A remote query.

The result returned would be all types defined at `:EmpDB`.

It is now possible to combine information from different sources, i.e. perform *multidatabase joins*. For example, if we have recorded study information about students in a database `:StudDB` and information about employees in `:EmpDB`, we can now retrieve the major for each student who is also an employee by submitting the following query[13];

12. `@` this syntax is currently not implemented.
13. Function `ssn` is the social security number for a person.

```
select name(e)@:EmpDB, major(s)@:Stud
  for each Employee@:EmpDB e, Student@:StudDB s
  where ssn(e)@:EmpDB=ssn(s)@:StudDB;
```

Example 37: Multidatabase query.

Having to type the database-identifying expression all the time could be tiresome; therefore the `select` statement is augmented to allow a default database to be specified. If no database is explicitly denoted then the default database will be used. So, the above query could be restated as shown below.

```
select@:EmpDB name(e),major(s)@:Stud
  for each Employee e, Student@:StudDB s
  where ssn(e)=ssn(s)@:StudDB;
```

Example 38: Equivalent multidatabase query.

Observe that we have used the `ssn` to decide whether two objects are equal or not, i.e. we are using *value-based identity* (cf. "Object Equivalence" on page 12).

3.4 Rules

Rules are used to define constraints in AMOS [Sköld, 1994]. They can also be used by applications to monitor specific events in the database. A rule has a *condition part* and an *action part*. The condition part of the rule is a Boolean expression. If some event in the database changes the value of the condition to *true*, then the rule is marked as triggered. If something happens later in the transaction which causes the condition to become *false* again, the rule is no longer triggered. This ensures that we only react to logical events.

In the *check phase* (usually done before committing the transaction), The actions are executed for each triggered rule.

Let us look at an example;

```
create function previous_salary(Employee)->Integer
  as stored;
create function set_salary(Employee e, Integer i)->Boolean
  as begin
      set previous_salary(e) = salary(e);
      set salary(e)=i;
  end;
```

Example 39: Procedures for updating the salary.

Then we define procedures for what to do when a salary is decreased.

```
create function compensate(Employee e)->Boolean[14]
  as set salary(e)=previous_salary(e);
```

Example 40: Employees' salaries cannot be decreased.

14. Boolean is used as result type for functions not returning any results, i.e. procedures.

```
create function compensate(Boss)->Boolean;
```

Example 41: Dummy procedure, managers are not compensated.

Finally we define the rule to detect decreasing salaries for all employees.

```
create rule no_decrease() as
  when for each employee e
  where salary(e)<previous_salary(e)
  do compensate(e);
```

Example 42: Rule to monitor salary decreases.

Since Boss is a subtype of Employee, the rule is overloaded for managers (because the functions salary and compensate are overloaded).

The following example would lower the salary for George but not for John.

```
set_salary(:george, 15000);
set_salary(:john, 15000);
commit; Commit a transaction. John's salary will be reset.
```

Example 43: Trying to lower salaries.

After the execution of the above statements George's salary has been reduced from 40000 to 30000 whereas John still has a salary of 17000.

3.5 AMOS Data Model as CDM

If we compare the AMOS data model with the requirements listed in "The Canonical Data Model" on page 7, we see that it is well suited as a CDM.

- **Classification/Instantiation.** Objects are classified by types. Objects have a unique object identity and are instances of one or more types.
- **Generalization/Specialization.** Types are organized as subtypes/supertypes and subtypes inherit all functions defined for the supertypes. Multiple inheritance and even different kinds of specialization are supported through derived types.
- **Aggregation/Decomposition.** Aggregation is supported through the use of types and functions. For example, the aggregation *address* described in section 1.3.1 under aggregation, is created by defining a new type `Address`, and the functions `city:Address->City`, `street:Address->Street`, and `zipcode:Address->ZipCode`.
- **Operations and integrity constraints.** AMOS supports the definition of new operations as we can define new derived functions. We can also customize functions through overloading and overriding. When importing data from an EDS, foreign functions are used which allow for arbitrary customization. The rule mechanism allows us to define constraints.
- **View mechanism.** AMOS supports a view mechanism, through derived functions, mapped and derived types, stronger than the view mechanism available in relational algebra.
- **Integration operators.** Integration operators are not fully supported yet. This thesis is a step towards completely supporting them but the complete realization will be future work.
- **Multiple semantics.** Through the view mechanism AMOS supports multiple semantics. Each user can customize, by using views, how (s)he prefers to view information.

4 Object Views in AMOS

In this chapter we will introduce the kinds of object views that exist in AMOS and in section 4.3 we will use object views to perform integration.

Object views are somewhat more complicated than relational views. In relational database systems, a view is traditionally defined to be a named, persistent query, i.e. a virtual relation. Relational conceptual schemata are concerned with tables as distinct units (in that tables are independent and related only by means of foreign keys); hence it is trivial to incorporate a virtual table into the schema in a relational system by simply adding it to the set of already existing tables. To some extent we have a similar situation in object-oriented systems. Derived functions, as presented in section 3.2, are examples of object views. To incorporate them into the schema is straightforward; simply add the derived function to the set of known functions. We can then treat it like any other function except for updates.

Updates of views are a well-known problem referred to as the *view update problem*. Assume that we have defined a type `Boss` with properties `salary` and `bonus`. We then define a derived function `annual_income` as;

```
create function annual_income(Boss b)->Integer as
   select 12*(salary(b)+bonus(b));
```

Suppose that we want to set the annual income of a certain boss to 200 000 by submitting the following statement;

```
set annual_income(:boss)=200000;
```

How should we perform this update? Should `salary` be affected or `bonus`? Perhaps both `salary` and `bonus` should be changed. There is no way a DBMS can deduce the correct action to take in this case. The typical solution is that the user has to define the update procedure for derived functions.

However, derived functions are not the only kind of possible object view; *virtual types* are another example. Virtual types have no extent explicitly stored in the database. Their extent is defined in terms of stored types and other virtual types. Unlike stored types virtual types let us organize local and non-local information according to our own preferences. If we use information stored in some non-local database then we cannot usually modify the schema of that database, i.e. we have to accept the way information is organized. However, locally we may reorganize the information. This is where virtual types can help us.

As in the case of derived functions we wish to treat virtual types no differently

from stored types. We would like virtual types to be part of the type hierarchy and we want to be able to define functions ranging over them. We also want to be able to create instances of virtual types which means that we have to address the view update problem. Constructors and initializors (section 3.2.1) let us do this.

Virtual types can be divided into two categories, those that create new objects and those that do not. We need both types to be able to perform a successful integration. Virtual types that only are concerned with existing objects are called *derived types*, whereas virtual types generating new objects are called *mapped types*. Mapped types are used when we have to integrate non-local information. Derived types allow us to organize the integrated information according to our preferences.

We next discuss the semantics and problems related to derived and mapped types. The implementation issues concerning derived and mapped types are presented in chapter 5.

4.1 An Integration Example

We introduce an example that we will use in this chapter to clarify the use of virtual types.

Let us return to our person database in figure 3.3 on page 37. We remodel it slightly so we can show how virtual types can help us. The example may seem contrived but it has to be small for reasons of clarity. However, the techniques described in this chapter are equally applicable to real-world problems.

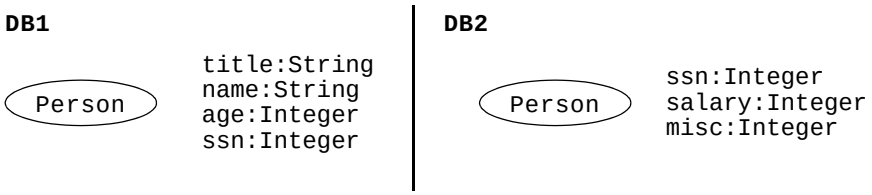


Figure 4.1: Modified person database schema.

We add a new property `ssn` (social security number) that allows us to identify persons uniquely. Furthermore, information is stored in two separate databases, `DB1` and `DB2`.

In `DB1` we store the `name` and `age` of a person. We also store the title which tells us what work a person does. Only persons employed have a title. In `DB2` we store the salary for employees. Property `misc` records the typing speed for secretaries and the bonus for bosses. Note that we do not know how to interpret the information in `misc` without accessing the value of `title` in `DB1` for an employee. If `title` in `DB1` says `"Secretary"` then we know that `misc` contains the typing speed. Furthermore, only in the case where `misc` has no value can we tell the salary of an employee. Should `misc` contain a value, then we

must know if the employee is a boss or not before we can calculate the salary, since the salary for a boss is the sum of `salary` and `misc`. It is evident that we need to integrate `DB1` and `DB2`.

OID[a]	ssn	title	name	age
$_{DB1}O_1$	123	"Boss"	"George"	55
$_{DB1}O_2$	234	"Secretary"	"Alice"	26
$_{DB1}O_3$	345	"Engineer"	"John"	33
$_{DB1}O_4$	456	NULL	"Joe"	67

Table 4.1: Extent of properties for `Person@:DB1`.

a. Object identifier for object representing a person.

In Table 4.1 and Table 4.2 the extents of the different properties defined for Person in `DB1` and `DB2` are shown.

OID	ssn	salary	misc
$_{DB2}O_1$	123	25000	15000
$_{DB2}O_2$	234	13000	150
$_{DB2}O_3$	345	17000	NULL

Table 4.2: Extent of properties for `Person@:DB2`.

We need to define one more database that we will use in this section. The database is maintained by a company, Mobile Inc., producing vehicles and contains information about boats and cars. For both boats and cars we will record the weight and the model.

Figure 4.2: Mobile Inc. database schema.

Next we will see how derived types can be used to restructure information in DB3. In section 4.3 we will show how mapped and derived types can be used to integrate these three databases.

4.2 Derived Types

Derived types are useful when doing integration since they allow us to reorganize information according to our own preferences; in particular they can be used to address structural differences, See “Structural Differences” on page 10. In section 4.3 we will see an example of this.

A derived type is a type whose extent is defined in terms of stored types and other virtual types. Derived types are created by the `create derived type` statement.

```
create derived type <type name>
  [ subtype of <type list> ]
  [ supertype of <type list> ]
  [ properties <property list> ]
  ( initializor <function definition> )*[1]
  ( constructor <function definition> )*
  as <type expression>;

<type expression>::=
  <simple type expression> |
  <type expression> and <type expression> |
  <type expression> or <type expression> |
  '(' <type expression> ')'

  <simple type expression>::=
  '[' <type name> [ <variable>
                [ [ <for each clause> ]
                    <where clause> ] ] ] ']'
```

Example 44: Syntax of the `create derived type` statement.

The extent of a derived type will be exactly those elements selected by the `<type expression>`.

Suppose that we would like to record the allowance for all persons older than 65 years. Let us call such persons seniors. We could then create a subtype of `Person` named `Senior` and give it a property `allowance`. However, an ordinary stored type would not suffice. People age, i.e. at some point in time they grow old enough to become seniors. If `Senior` is a stored type, then we would have to go through all persons every now and then and add type `Senior` to persons old enough. A better solution can be achieved if type `Senior` somehow picked out the persons of interest. This is exactly what a derived type does. Let us therefore define `Senior` as:

```
create derived type Senior
  properties (allowance Integer) as
  [Person p where age(p)>65];
```

Example 45: Example of a derived specialization type.

1. Zero or more occurrences.

If we now asked for the name of all seniors, AMOS would respond with `"Joe"` since he is the only one whose age is greater than 65. However, should we change the age of another person to something greater than 65 and ask the same question again, we would get one more name in the result.

While `Senior` is a derived type, we can still define properties as we would do if it was a stored type. Views that allow definition of new attributes or methods have been named *capacity augmenting views* [Ra and Rundensteiner, 1995]. Capacity augmenting views are required when trying to integrate several information sources or we would lose information as shown in "Attribute Operators" on page 20.

Types denoted by `<type name>` in `<simple type expression>` (example 44) will be called *defining types* since they are used to define a derived type. In the case of `Senior`, `Person` is the defining type.

4.2.1 Specialization, Intersection, and Union Types

A problem with derived types is where to place them in the type hierarchy. As we want derived types to resemble stored types as far as possible, we also want them to be incorporated in the type hierarchy. However, we have to take the type membership of instances of a derived type into consideration when inserting the derived type in the hierarchy. Three base cases can be identified: *specialization types*, *intersection types*, and *union types*.

- The first case involves *derived specialization types*. A derived type is a specialization type if the `<type expression>` select objects of *one type only*. An example is our type `Senior`. All instances of `Senior` are persons with an age greater than 65, i.e. we have formulated a restriction for instances of type `Person` for participation in `Senior`. Since all seniors also are persons, any function applicable to persons is also applicable to seniors. Let F_P be the set of functions applicable to `Person` objects, *applicablefns(`Person`)*, and F_S the set of functions that are applicable to `Senior` objects, *applicablefns(`Senior`)*, we then have:

  $$extent(Senior) \subseteq extent(Person) \wedge F_P \subseteq F_S$$

  From this we can deduce that the proper placement of `Senior` would be as subtype of `Person`. This is also what our intuition tells us. In figure 4.3 we see the resulting type hierarchy[2].

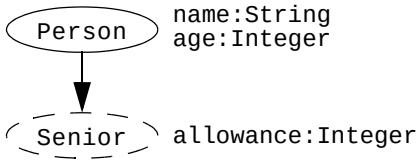


Figure 4.3: Type hierarchy with derived specialization type.

Now suppose that a user specifies type `OldSenior` as in example 46. Intuitively `OldSenior` should be a subtype of `Senior` since `Senior` subsumes `OldSenior`. Currently the user has to specify this explicitly or `OldSenior` will become a sibling of `Senior`. However, the problem of subsumption deserves to be studied closer as it not only allows us to perform a more accurate placement of types in the hierarchy, but it can also improve the optimization of queries. In the general case the subsumption problem is undecidable but many cases can be handled.

```
create derived type OldSenior as
  [Person p where age(p)>90];
```

Example 46: Derived specialization type subsumed by `Senior`.

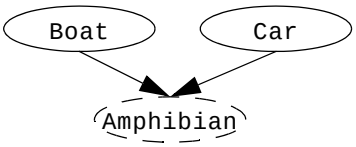
- The second case involves *intersection types*. An intersection type is a derived type where the type expression defining the type select objects of multiple incomparable types. Intuitively, all instances of an intersection type are required to be members of the intersection between the type extents of a number of incomparable types.

  Let us return to our example. Suppose that some creative engineer working for Mobile Inc. has created an amphibian. As no type existed for amphibians, he solved the problem using the following statements.

```
create Car(weight,model) instances :froggy(7,'Froggy');
add type Boat(weight,model) to :froggy(7,'Froggy');
```

  Example 47: Creating an amphibian.

  When the manager discovered this, he realized that it would be very nice to be able to reason about amphibians so he defined type Amphibian as in figure 4.4.




```
create derived type Amphibian as
  [Boat] and [Car];
```

Figure 4.4: Example of an intersection type.

  All instances of `Amphibian` in figure 4.4, are required to be both of type `Boat` and of type `Car`, which Froggy is indeed. Also, `Boat` and `Car` are incomparable since they are not related to each other via a sub/supertype relationship.

  `Amphibian` was made a subtype of `Car` and `Boat`; we shall now justify this. Assume that objects selected by `<type expression>`, for an intersection

2. We use dashed ellipses to denote virtual types.

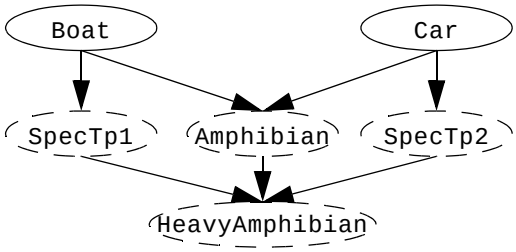
type t_{int}, are required to be of type t_1 to t_n. Furthermore let F_1 to F_n be the set of functions applicable to instance of t_1 to t_n respectively, and F_{int} the set of functions applicable to instances of t_{int}, then the following holds;

$$extent(t_1) \cap ... \cap extent(t_n) = extent(t_{int}) \wedge$$
$$\forall t(extent(t_{int}) \subseteq extent(t) \wedge t \in \{t_1,...,t_n\}) \wedge$$
$$\forall f(f \subseteq F_{int} \wedge f \in \{F_1,...,F_n\})$$

Following the same chain of reasoning as in the case of derived specialization types, we see that the intersection type t_{int} should be a subtype of types t_1 to t_n.

Since `Amphibian` is a subtype of `Boat` and `Car`, it inherits the properties defined. Thus, it is possible to use the functions `model` and `age` on an instance of `Amphibian`. However, since we inherit model and age from both `Boat` and `Car` we have to state which resolvent to use or AMOS will complain, i.e. we have to say `Car.model->String` or `Boat.weight ->Integer`.

It is also allowed to create intersection types where restrictions have been applied to the defining types. `HeavyAmphibian` in figure 4.5 is an example of this. Only boats and cars that weigh more than 10 tons are considered, i.e. some boats and cars may have been excluded, whereas in the case of `Amphibian` the whole extents of `Boat` and `Car` were considered. This is a combination of specialization types and an intersection type. AMOS handles the situation by creating two specialization types and then creating an intersection type of these two specialization types. The resulting type hierarchy is depicted in figure 4.5.




```
create derived type HeavyAmphibian as
   [Boat b where weight(b)>10] and
   [Car c where weight(c)>10];
```

Figure 4.5: Intersection type of implicit specialization types.

Assuming that type `Amphibian` has been defined as in figure 4.4, `HeavyAmphibian` could be defined in two different ways; either as in figure 4.5 or as in example 48. If we choose the definition in example 48, then `HeavyAmphibian` would simply have become a specialization type of `Amphibian`

and the two system-generated specialization types `SpecTp1` and `SpecTp2` would not have been created.

```
create derived type HeavyAmphibian as
      [Amphibian a where Car.weight->Integer(a)>10];
```

Example 48: Alternative definition of `HeavyAmphibian`.

Observations
We note that derived specialization types could be viewed as a special case of intersection types. As the schema evolves when users create or delete types, specialization types may gain new supertypes and become intersection types. Likewise intersection types may lose supertypes and become specialization types.

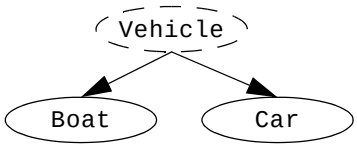
The observant reader may have asked what the difference is between defining `Amphibian` as in figure 4.4 and as in example 49.

```
create type Amphibian subtype of Boat,Car;
```

Example 49: Normal Amphibian subtype.

To answer this question let us go back to example 47 on page 58. If we define Amphibian as in example 49, Froggy would not become an amphibian unless we explicitly added type `Amphibian`. But using the definition in figure 4.4 Froggy automatically becomes an amphibian.

- The third case involves *union types*. Intuitively an instance of a union type is required to be an instance of at least one of several incomparable types. At Mobile Inc. the manager soon realised that a `price` property was needed for both boats and cars. However, instead of defining the property for both `Boat` and `Car`, he decided to create a new type `Vehicle` as a supertype of `Boat` and `Car`.




```
create derived type Vehicle
  properties (price Integer)
  as [Boat] or [Car];
```

Figure 4.6: Example of a union type.

Let t_1 to t_n be incomparable types. F_1 to F_n are the set of functions applicable to instances of t_1 to t_n respectively. Then, for the union type t_U and the set of functions, F_U, applicable to instances of t_U the following must hold;

$$extent(t_1) \cup ... \cup extent(t_n) = extent(t_U) \wedge$$
$$\forall t(extent(t) \subseteq extent(t_U) \wedge t \in \{t_1,...,t_n\}) \wedge$$
$$\forall f(F_U \subseteq f \wedge f \in \{F_1,...,F_n\})$$

Since $extent(t_U)$ is a superset of $extent(t_i)$ for any i, t_U should become super-

type of t_1 to t_n. We also note that since t_U is the supertype, any property defined for it will be inherited by t_1 to t_n.
In figure 4.6 all instances of `Vehicle` are either a car or a boat, or both car and boat.

Having defined type `Vehicle` as in figure 4.6 we can ask for the price of a certain vehicle. However, it would be convenient if we could also ask what model or weight a vehicle has. For example, suppose that the user submits the following query;

```
select weight(v) for each Vehicle v;
```

Example 50: Retrieving the weight of all vehicles.

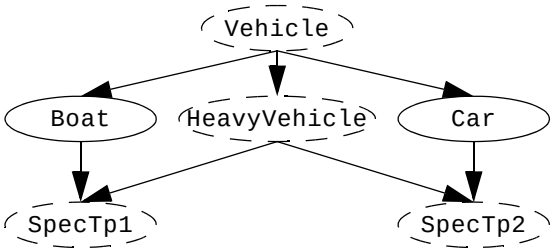
In this case AMOS would issue an error since property `weight` is not defined for type `Vehicle`. However, there exists a solution to our problem by defining the following properties.

```
create function weight(Vehicle)->Integer;
create function model(Vehicle)->String;
```

Example 51: Creating dummy properties for a union type.

These properties are defined just to allow us to utilize late binding. We will never store any information in `Vehicle.weight->Integer` or `Vehicle.model->String`. They are just needed to avoid getting an error in example 50. Since each vehicle is also a boat or a car, `weight` will be late bound in the example and the appropriate function will be invoked.

As in the case of intersection types we may create union types where restrictions have been applied to the defining types. An example of this is shown in figure 4.7. The extent of `HeavyVehicle` is a subset of the union of the




```
create derived type HeavyVehicle as
  [Boat b where weight(b)>10] or
  [Car c where weight(c)>10];
```

Figure 4.7: Union type of implicit specialization types.

extents for `Boat` and `Car`. Thus `HeavyVehicle` cannot be a supertype of `Boat` and `Car`, rather it should become a subtype of `Vehicle`, the union type of `Boat` and `Car`. AMOS creates two system generated, implicitly defined,

specialization types of `Boat` and `Car` and makes `HeavyVehicle` their union type.

Observations
The observant reader wondering about the differences between `Amphibian` as defined in figure 4.4 and example 49 may also ask if there is a difference between defining `Vehicle` as in figure 4.6 and as in example 52.

```
create type Vehicle supertype of Boat, Car;
```

Example 52: Ordinary Vehicle supertype.
The answer is *yes*. All boats and cars automatically become instances of `Vehicle` no matter what definition is chosen due to the semantics of the `create instances` statement. However, when `Vehicle` is defined as in example 52 there may exist vehicles that are neither boat nor car. With the definition in figure 4.6 all vehicles have to be either boat or car.

AMOSQL does not restrict users to the three base cases described, when defining derived types. Arbitrarily complex derived types can be defined by combining types using `and` and `or` in `<type expression>`. AMOS will then construct implicitly defined types as needed.

4.3 Mapped Types

Mapped types, in contrast to derived types, create new so-called *mapped objects*. There exist two primary uses for mapped types. The first is to create objects when we map external data sources into the AMOS data model. A thorough discussion of using mapped types to provide object views of relational data is given in [Fahl, 1994]. The second use is to solve the problem of object equivalence; see "Object Equivalence" on page 12.

Let us now see how we can use mapped and derived types to perform integration of the three databases presented in section 4.1. We want to combine information from the three databases in a way that the resulting schema of our integrated database is similar to the schema in figure 3.3 on page 37.

We begin by integrating databases `DB1` and `DB2`, creating a single `Person` type where the information present in the two databases is combined. This requires us to address the object equivalence problem, section 1.5.2, since the same physical person may be represented in both `DB1` and `DB2`.

```
create mapped type Person converter p2db1p, p2db2p as
  select p1,p2 for each Person@:db1 p1, Person@:db2 p2
  where ssn@:db1(p1)*=ssn@:db2(p2)[3];
```

Example 53: Creating a mapped `Person` type.

3. *= denotes left outer join.

The extent of `Person` will contain one *mapped object* for each *unique* result tuple produced by the `select` statement. We call these tuples *keys*. Elements of a key are called *defining objects*.

The type `Person` will contain one mapped object for each instance of `Person@:db1`. To relate instances of `Person@:db1` with instances of `Person@:db2` we use social security number (`ssn`) as the identifying property. Note that we have to use left outer join since there exist persons in `DB1` not represented in `DB2`. If we used an ordinary join, the extent of `Person` would only have three objects instead of four.

In Table 4.1 the relationship between mapped objects, keys and defining objects is shown.

OID for mapped object *mo*	key	p2db1p(*mo*)	p2db2p(*mo*)
O_1	$<{}_{DB1}O_1, {}_{DB2}O_2>$	${}_{DB1}O_1$	${}_{DB2}O_1$
O_2	$<{}_{DB1}O_2, {}_{DB2}O_2>$	${}_{DB1}O_2$	${}_{DB2}O_2$
O_3	$<{}_{DB1}O_3, {}_{DB2}O_3>$	${}_{DB1}O_3$	${}_{DB2}O_3$
O_4	$<{}_{DB1}O_4$, NULL>	${}_{DB1}O_3$	NULL

Table 4.1: Relationship between mapped objects, keys and defining objects.

In example 53 we specified two *converters*. We have to specify as many converters as the width of the result tuples of the select statement defining a mapped type. For example, John is represented by ${}_{DB1}O_3$ in `DB1` and ${}_{DB2}O_3$ in `DB2`. In the integrated database he is represented by O_3. The two converters allow us to map back and forth between instances of a mapped type and the defining objects. So, `p2db1p(`O_3`)` returns ${}_{DB1}O_3$. Converters can also be used in the backward direction, e.g. `p2db1p(x)=`${}_{DB1}O_2$ would bind `x` to O_2.

The full syntax of the `create mapped type` statement is;

```
create mapped type <type name>
  [ subtype of <type list> ]
  [ properties <property list> ]
  ( initializor <function definition> )*
  ( constructor <constructor decl.> )*
  converter <fnname list>
  as <select statement>;
```

Example 54: Syntax of the `create mapped type` statement.

We now define the properties for type Person.

```
create function name(Person p)->String as
  select name@:db1(p2db1p(p));
create function age(Persons p)->Integer as
  select age@:db1(p2db1p(p));
create function title(Person p)->String as
  select title@:db1(p2db1(p));
```

Example 55: Person properties stored in DB1.

In example 55 we see how converters are used. For example, name for Person is stored in database DB1 and can be accessed by name@:db1. However, name@:db1 expects to get OIDs originating from DB1 and not OIDs from the integrated database. Thus, we need to convert the mapped object into the corresponding defining object so we can call name@:db1.

In the example above we defined property title. We can now use this property to define the type hierarchy of figure 3.3 on page 37, using derived types.

```
create derived type Employee as
  [Person p where some(title(p))];
create derived type Secretary as
  [Employee e where title(e)="Secretary"];
create derived type Boss as
  [Employee e where title(e)="Boss"];
```

Example 56: Completing the type hierarchy.

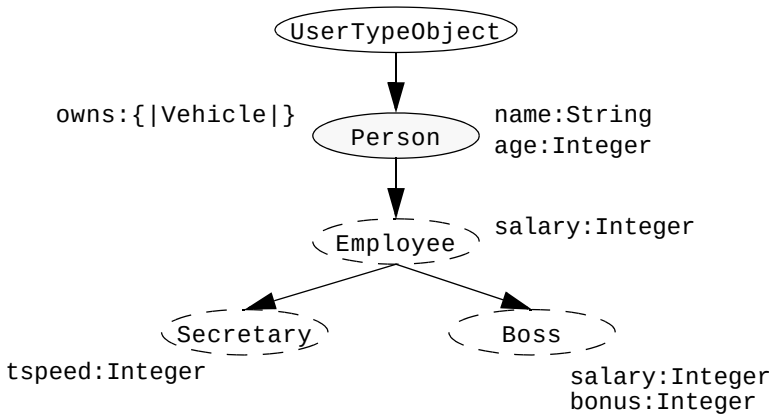


Figure 4.8: Type hierarchy of virtual types.

We have defined three derived types Employee, Secretary and Boss. Employees are all persons that have a title. John, Alice and George are all employees whereas Joe is not. Secretaries are all employees with title secretary and bosses are those employees that have boss as title.

The type hierarchy as it looks after having defined the above types is depicted in figure 4.8[4]. Let us now define the remaining properties.

All employees should have a salary. The salary is stored in DB2; we thus define salary as

```
create function salary(Employee e)->Integer as
  select salary@:db2(p2db2p(e));
```

Example 57: Defining salary for type Person.

For secretaries we want to know the typing speed. This information was stored in the misc property in DB2. We define tspeed as

```
create function tspeed(Secretary s)->Integer as
  select misc@:db2(p2dbd2p(s));
```

Example 58: Defining tspeed for secretaries.

For type Boss we then define the bonus and salary properties as

```
create function bonus(Boss b)->Integer as
  select misc@:db2(p2db2p(b));
create function salary(Boss b)->Integer as
  select Employee.salary->Integer(b)+bonus(b);
```

Example 59: Creating bonus and salary for type Boss.

We have performed semantic enrichment of the information in DB1 and DB2. Two different Person types have been integrated into one. We have introduced inheritance in the integrated schema. In the schema of DB1 and DB2 no inheritance was present. The property misc in DB2 was used for different purposes depending on the value of the property title in DB1. We have now made the interpretation of misc explicit through the definition of properties tspeed and bonus.

To conclude we define type Vehicle and related properties.

```
define mapped type Vehicle converter v2db3v
  as select v for each Vehicle@:db3;
create function name(Vehicle v)->String
  as select name@:db3(v2db3v(v));
create function weight(Vehicle v)->Integer
  as select Vehicle@:db3(v2db3v(v));
create function owns(Person)->bag of Vehicle;
```

Example 60: Creating mapped type Vehicle and associated properties.

We have described how mapped types can help us integrate AMOSs. However, the very same technique is, of course, applicable if we want to integrate some other data source such as a relational database [Fahl, 1994], a file, or perhaps a WWW-client.

It should also be noted that the use of mapped types when integrating AMOSs is only required while full level 3 support is lacking, See “The Global Schema Approach” on page 13. Once we support full level 3 integration, mapped types are only needed when accessing information in non-AMOS systems or when we

4. Shaded ovals denote mapped types.

need to address the object equivalence problem.

4.4 Creating Instances

Our effort to make derived types behave as stored types includes being able to create instances of derived types, i.e. invoke the `create instances` statement for a derived type.

Since the extent of a derived type is defined in terms of other type extents, trying to create an instance of a derived type involves creating instances of the defining types.

Let us return to the `Senior` type defined in example 45 on page 56. A senior is a person whose age is greater than 65. An instance of `Senior` is created by the statement in example 61.

```
create Senior(age) instances :s(67);
```

Example 61: Creating a senior.

Since `Senior` is a derived type, it has no materialized extent[5]. What really happened when we created the senior above was that a person was created and that the age of that person was set to 67. The newly created person then became an instance of `Senior`.

```
create constructor Senior() as
  begin
      declare Object o;
      set o=create_object();
      add type Person to o;
      result o;
  end;
```

Example 62: Default constructor for `Senior`.

If we had given the new person an age of 55 instead, the person would not have qualified as a senior and should thus not have been created. To accomplish this we revise the semantics of the `create instances` statement slightly.

When the `create instances` statement is invoked it:
- invokes the appropriate constructor
- assigns values to specified properties
- checks to see if the new object qualifies as an instance of the derived type by calling the type predicate function; see section 5.1.2
- if the object did not qualify, reset the state of the database to the state it had before we invoked the create instances statement.

5. As a consequence of this, make_instance may not be called with the name of a derived type.

The default constructor in example 62 forces us to specify the age whenever we want to create a senior. If we find this cumbersome, then the default constructor could be redefined to give the person created a default age of 66.

AMOS provides default constructors for all derived types. Default constructors for intersection types are defined similarly to default constructors for specialization types, i.e. an object is created and it will become an instance of each of the defining types. In example 63 the default constructor provided by AMOS for type `Amphibian` is shown.

```
create constructor Amphibian() as
  begin
      declare Object o;
      o=create_object();
      add type Boat to o;
      add type Car to o;
      result o;
  end;
```

Example 63: Default constructor for `Amphibian`.

The behaviour of default constructors provided by AMOS for union types, is to signal an error. This is because the system cannot know what subtype a new instance should belong to. For example if we try to create an instance of `Vehicle`, should the instance be a boat or a car. The user has to specify this by supplying a new default constructor.

```
create constructor Vehicle () as
  begin
      print("Unspecified default constructor for
             union type Vehicle");
      abort();
  end;
```

Example 64: System-generated default constructor for union type `Vehicle`.

Typically mapped types are used to integrate external data sources. The default constructors generated for mapped types signal an error since AMOS cannot know how to create instances in the external data sources. If users supply a default constructor, then it will be possible to create instances of a mapped type, too.

4.5 Concluding Remarks

In this section we have seen how objects views can be used to perform integration. Two different kinds of objects views were introduced, derived types and mapped types.

Derived types allowed us to define type membership in terms of property values. This is useful to solve the problem of structural differences as described in

section 1.4.3. An example of this was shown in section 4.3.

Mapped types allowed us to construct objects for foreign data allowing us to treat external information in an object-oriented manner. Mapped types were also used to solve the object equivalence problem described in section 1.5.2. An example of this was shown in section 4.3.

In section 4.4 we saw how constructors and initializors allow us to create instances of derived and mapped types.

Let us compare our approach with the ones described in chapter 2. Compared to MultiView we can offer the same functionality with the exception of the view mechanism. Specialization types in AMOS correspond to the `select` operator of MultiView and COOL*. The `union` and `intersection` operators correspond to our intersection and union types. The difference operator has to be expressed using negation combined with an intersection or union type.

We may define new properties for a type as `refine` does in MultiView. However, we do not get a new type. If we want to achieve the same semantics, we could do it by defining a specialization type without any condition, for which we define the new property.

`Hide`, can be simulated by defining a supertype, defining the properties that we want to keep, for the supertype. However, there is not much point doing this as AMOS currently does not provide a view mechanism similar to view schemas in MultiView. This is one of the weaknesses of AMOS and future work should include some authorization mechanism combined with the possibility of defining view schemas as in MultiView.

In addition to derived types we provide mapped types. This enables us to create object views of non object-oriented information. It also provides us with the ability to merge objects as shown in example 53 on page 62. Thus, we can provide at least level 2 integration (section 1.5.3). COOL* provides level 3 integration. We feel that this will be possible in AMOS, with a fair amount of work. The main reason for not providing it now is that it requires the query compiler to be rewritten.

Pegasus provides both derived types, the ability to define object views of non-object-oriented information by means of *producer types*, and merge objects using *unifying types*. In our approach we use the same mechanism, mapped types, to provide the functionality of both producer and unifying types. We believe that it is easier for users not having to learn many different concepts, but be able to use the same concept for several things.

5 Implementation

In this section we provide a brief description of how mapped and derived types are implemented in AMOS. We also describe some of the problems encountered during our work. We conclude with a discussion comparing the behaviour of our implementation with the systems described in chapter 2.

5.1 Derived Type Implementation

If we want to implement derived types as described in section 4.2, there are two main approaches that can be chosen. The *materialized approach* and the *derivation approach*. We just outline the materialized approach as we have not implemented it yet. The derived approach will be discussed more thoroughly and we will also describe the problematic issues involved.

5.1.1 The Materialized Approach

The materialized approach means that we materialize the extent of derived types and that we keep the extent updated as the state of the database changes.

Suppose that we define a derived type `Senior` as:

```
create derived type Senior
  [Person p where age(p)>65];
```

Example 65: Example of a derived specialization type.

We would then perform an initial materialization by selecting all persons older than 65 and add type `Senior` to them.

```
for each Person p where age(p)>65
  add type Senior to p;
```

Example 66: Adding type senior to persons older than 65.

Having done this we now need to maintain the extent of `Senior`, i.e. whenever an instance of `Person` changes the value of the `age` property, we have to check whether that instance should become a member of `Senior` or, if the instance already is a senior, if it should lose the `Senior` type and just become an ordinary person. This can be accomplished by specifying the following two rules.

```
create rule newSenior() as
  when for each Person p
  where age(p)>65
  do add type Senior to p;
```

```
create rule removeSenior() as
  when for each Person p
  where age(p)<=65
  do remove type Senior from p;
```

Example 67: Rules maintaining the extent of `Senior`.

The `newSenior` rule makes sure that any person older than 65 becomes a senior, whereas the `removeSenior` rule removes from the extent of `Senior` any person whose `age` is reduced to 65 or less.

Using this approach requires some changes to the rule machinery of AMOS. Today rules are checked at commit time or when the system function `check()` is explicitly called. We would have to make sure that rules associated with derived types would be checked whenever an event occurred since a user expects a person to become a senior as soon as the value of the `age` property gets a value greater than 65. This immediate checking of rules would make the system slow if we had a high update ratio compared to the number of accesses. However, if we had very few updates, this approach would be preferable since we never need to calculate what types an object is an instance of, extents of derived types would always be materialized. When we use the derivation approach below, we must calculate the extent of a derived type whenever it is needed.

5.1.2 The Derivation Approach

The derivation approach means that we never materialize the extent of a derived type. Instead we derive it when it is needed. This is where type extent functions and type predicate functions come in.

Whenever a type is created, a type extent and a type predicate function are also created. The type extent function returns the extent of the type and the type predicate function returns *true* if an object is an instance of the specific type.

For stored types these functions are trivial. If we assume the existence of a stored type `Car` then the type extent function for `Car` will be defined as in the example below.

```
create function car()->Car as
  select o for each Car o;
```

Example 68: Type extent function for the stored type Car.

The type predicate function is equally simple.

```
create function carp(Object o)->Boolean as
  select true for each Car c where c=o;
```

Example 69: Type predicate function for the stored type Car.

Let us now look at how type extent functions for derived types are constructed. The syntax for the `create derived type` statement was shown in example 44 on page 56. The type extent function and the type predicate function are

constructed from the <type expression>.

When we define a derived type we write:

```
create derived type <typename> as <type expression>
```

Example 70: Template for defining a derived type.

A set of simple transformation rules can be specified that construct the type extent and the type predicate function from this expression. Let us begin with the type extent function.

Let us define a function DEF (*Derived type Extent Function*) that, given a type expression and two names, will construct the AMOSQL statement creating the type extent function for a derived type. Given the statement in example 70, we call DEF as DEF⟦ <type expression> ⟧ <typename> <typename>.

```
DEF⟦ [TPNM] ⟧ FNM TNM =
  create function FNM()->TNM as
      select o for each TPNM o;

DEF⟦ [TPNM VAR] ⟧ FNM TNM =
  create function FNM()->TNM as
      select VAR for each TPNM VAR;

DEF⟦ [TPNM VAR where COND] ⟧ FNM TNM =
  create function FNM()->TNM as
      select VAR for each TPNM VAR where COND;

DEF⟦ [TPNM VAR for each FE-PAIRS where COND] ⟧ FNM TNM =
  create function FNM()->TNM as
      select VAR for each TPNM VAR, FE-PAIRS where COND;
```

Example 71: Definition of DEF.

Given the definition of the derived type RPV (RichPersonsVehicles) in example 72, DEF will construct the type extent function as shown in example 73.

```
create derived type RPV as
  [Vehicle v for each Person p
      where  owns(p)=v and
             salary(p)>25000];
```

Example 72: A specialization type.

```
create function rpv()->RPV[1] as
  select v for each Vehicle v, Person p
  where owns(p)=v and salary(p)>25000;
```

Example 73: Type extent function for derived type RPV.

We should note that the type extent function performs a type cast on objects. The objects selected by the function above are of type Vehicle but the return

1. The upper and lower case letters are just for notational convenience and of no semantic importance.

type of the function is of type RPV, i.e. whatever is returned by this function is an RPV.

Not only is a type extent function constructed but also a *type predicate function*. Thus, we define a function DPF (*Derived type Predicate Function*) that given a type expression and a name will construct the AMOSQL statement creating the derived type predicate function.

```
DPF⟦ [TPNM] ⟧ FNM =
  create function FNMp[2](Object o)->Boolean as
      select o=v for each TPNM v;

DPF⟦ [TPNM VAR] ⟧ FNM =
  create function FNMp(Object o)->Boolean as
      select o=VAR for each TPNM VAR;

DPF⟦ [TPNM VAR where COND] ⟧ FNM =
  create function FNMp(Object o)->Boolean as
      select o=VAR for each TPNM VAR where COND;

DPF⟦ [TPNM VAR for each FE-PAIRS where COND] ⟧ FNM =
  create function FNMp(Object o)->Boolean as
      select o=VAR for each TPNM VAR, FE-PAIRS where COND;
```

Example 74: Definition of DPF.

The type predicate function for RPV as constructed by DPF will be;

```
create function rpvp(Object o)->Boolean as
  select o=v for each Vehicle v, Person p
  where owns(p)=v and salary(p)>25000;
```

Example 75: Type predicate function for derived type RPV.

The two functions DEF and DPF are not complete yet. We must also define how to handle intersection types and union types. Let us first look at the case for intersection types.

```
DEF⟦ [TPEXP1 and TPEXP2] ⟧ FNM1 TNM =
LET   FNM2 = generate-name()[3]
      FNM3 = generate-name()
IN
  DEF⟦ [TPEXP1] ⟧ FNM2 Object
  DEF⟦ [TPEXP2] ⟧ FNM3 Object
  create function FNM1()->TNM as
      select o for each Object o where
          NAME2()=o and
          NAME3()=o;

DPF⟦ [TPEXP1 and TPEXP2] ⟧ FNM1 =
```

2. *FNM*p means that the value of variable *FNM* should be concatenated with a 'p'. Thus if the value of *FNM* is *rpv* then the result is *rpvp*.
3. Yes, we are cheating here. generate-name has an internal state and generates a new unique name each time it is called. Also, the LET construct is not part of AMOSQL.

```
LET   FNM2 = generate-name()
      FNM3 = generate-name()
IN
  DPF⟦ [TPEXP1] ⟧ FNM2
  DPF⟦ [TPEXP2] ⟧ FNM3
  create function FNM1p(Object o)->Boolean as
      select o where
          FNM2p(o) and
          FNM3p(o);
```

Example 76: DEF and DPF for intersection types.

If we return to our Amphibian example in figure 4.4 on page 58 we would create the type as:

```
create derived type Amphibian as
  [Boat] and [Car];
```

Example 77: Creating an intersection type.

This would result in a call to DEF as DEF⟦ [Boat] **and** [Car] ⟧ Amphibian Amphibian. Three AMOSQL functions will be produced by this call.

```
create function anon1()->Object as
  select o for each Boat o;
create function anon2()->Object as
  select o for each Car o;
create function amphibian()->Amphibian as
  select o for each Object o where
      anon1()=o and
      anon2()=o;
```

Example 78: Type extent function for intersection type Amphibian.

The type predicate function will be constructed similarly.

The case for union types is more interesting. We extend DEF and DPF as follows.

```
DEF⟦ [TPEXP1 or TPEXP2] ⟧ FNM1 TNM =
LET   FNM2 = generate-name()
      FNM3 = generate-name()
IN
  DEF⟦ [TPEXP1] ⟧ FNM2 Object
  DEF⟦ [TPEXP2] ⟧ FNM3 Object
  create function FNM1()->TNM as
      select unique(select o for each Object o where
                        NAME2()=o or
                        NAME3()=o);
```

```
DPF⟦ [TPEXP1 and TPEXP2] ⟧ FNM1 =
LET   FNM2 = generate-name()
      FNM3 = generate-name()
IN
  DPF⟦ [TPEXP1] ⟧ FNM2
  DPF⟦ [TPEXP2] ⟧ FNM3
  create function FNM1p(Object o)->Boolean as
      select some(select o where
          FNM2p(o) or
          FNM3p(o));
```

Example 79: DEF and DPF for union types.

Note the use of unique in the type extent function and the use of some in the type predicate function. Since an object can only be a member of a type extent once, we have to make sure that an object is not returned twice by the extent function of a union type. This would be the case if some amphibian existed and we created a derived union type Vehicle as in figure 4.6 on page 60. Since an amphibian is both a boat and a car, it would be returned twice if we did not use unique.

The use of some in the type predicate function is a *speed optimization*. AMOS uses *pipelined query execution*, i.e. no intermediate results are produced. The semantics of some is such that as soon as the subquestion succeeds, we are done. Suppose that we wanted to know whether an object *o* was an instance of a union type t_U and that the type predicate function for t_U did not contain some. We would first see if *FNM2p(o)* returned true; if it did, the type predicate function would return *true* as a result and would then retrieve the value of *FNM3p(o)* although it is not needed. Using some avoids calling *FNM2p*, thereby saving some time.

We have now shown the principle for how type extent functions and type predicate functions are constructed. Of course the definitions of the type extent function and the type predicate function have to be altered as the type hierarchy evolves and types gain and lose sub- and supertypes. However, we can still use the DEF and DPF functions for this purpose; we only give them as arguments the new defining type expression instead.

5.2 Mapped Type Implementation

Let us now turn our attention to mapped types. When creating a mapped type, a type extent function and a type predicate function are generated as usual. However, two more functions are generated, a *key generating function* (*key function* for short) and an *oidtranslate function*. These two functions are responsible for materializing the extent of a mapped type. The extent, however, is not materialized all at once but bitwise as it is needed [Fahl, 1994]. Instances of a mapped type are called *mapped objects*.

When we create a mapped type we write;

```
create mapped type MTPNM converter CNM1, ..., CNMn as
  select V1, ..., Vn for each TPNM1 V1, ..., TPNMn Vn
  where COND;
```

Example 80: Template for defining a mapped type.

where *MTPNM* is the name of the mapped type we wish to create, CNM_i is a name of a converter, V_i a variable name, $TPNM_i$ a type name, and *COND* the condition that selects the tuples we want to generate mapped objects for.

First of all the key function is generated. It returns all the tuples that we wish to create mapped objects for, i.e. the *keys*, See "Mapped Types" on page 62.

```
create function MTPNM_key()-><TPNM1, ..., TPNMn> as
  select V1, ..., Vn for each TPNM1 V1, ..., TPNMn Vn
  where COND;
```

Example 81: Template for defining the key function.

Let us look at an example[4]. Assume that we have two types defined `APerson` and `BPerson` both with the property `ssn` (social security number) defined. We now want to create a mapped type `Person` with one mapped object for each person that is represented as an instance of both `APerson` and `BPerson`. We would then define the mapped type `Person` as:

```
create mapped type Person converter p2ap, p2bp as
  select p1,p2 for each APerson p1, BPerson p2
  where ssn(p1)=ssn(p2);
```

Example 82: A mapped type definition.

The key function for type `Person` would then become:

```
create function person_key()-><APerson,BPerson> as
  select p1,p2 for each APerson p1, BPerson p2
  where ssn(p1)=ssn(p2);
```

Example 83: The key function for mapped type Person.

The function `person_key` will return a tuple where the two elements will be instances of `APerson` and `BPerson` respectively, that have the same `ssn`, i.e. it is the same person. We now need to create a mapped object for each tuple generated by `person_key`. However, we must be able to map from the new objects back to instances of `APerson` and `BPerson` respectively, by means of the converters. This means that we have to store the relationship between *mapped objects* and *defining objects*. To do this an `oidmaptable` is created as:

```
create function oidmaptable(MTPNM)-><TPNM1, ..., TPNMn>
  as stored;
```

Example 84: Template for defining the `oidmaptable`.

4. The example from chapter 4.3, slightly modified for readability.

Thus, the `oidmaptable` for type `Person` is defined as:

```
create function oidmaptable(Person)-><APerson,BPerson>
  as stored;
```

Example 85: `oidmaptable` for type `Person`.

Having defined the key function and the `oidmaptable` we can now turn our attention to the function that performs the actual translation between *keys* and *mapped object*. This function is named `oidtranslate`. The template for it appears thus:

```
create function oidtranslate(TPNM1 V1,...,TPNMn Vn)->MTPNM
  as begin
      declare Object o;
      select v into o for each MTPNM v
      where oidmaptable(v)=<V1, ..., Vn>;
      if some((select o)) then
          /* Mapped object exists, return it */
          result o
      else
          /* Have not seen this key before, new object */
          begin
              declare MTPNM v;
              set o=create_object();
              make_instace('MTPNM',o);
              set v=o;
              set oidmaptable(v)=<V1, ..., Vn>;
              result v
          end
  end;
```

Example 86: Template for the `oidtranslate` function.

The `oidtranslate` function first tries to look up an already existing mapped object in the `oidmaptable` given a key. If such an object exists, i.e. `some((select o))` is true, the object is returned. If no object exists, we have to create one. We do this using the `create_object` and `make_instance` functions described in section 3.2.1. We then update the `oidmaptable` with the new object and the key it corresponds to. Finally, we return the object.

For our example the `oidtranslate` function is as follows:

```
create function oidtranslate(APerson p1, BPerson p2)->Person
  as begin
      declare Object o;
  select v into o for each Person v
  where oidmaptable(v)=<p1,p2>;
  if some((select o)) then
      result o
  else
      begin
          declare Person v;
          set o=create_object();
```

```
            make_instance('Person', o);
            set v=o;
            set oidmaptable(v)=<p1,p2>;
            result v;
        end
    end
```

Example 87: The `oidtranslate` function for mapped type `Person`.

The reader may wonder about the `set v=o` statement in the `else`-clause. Since `o` is declared to be of type `Object` AMOS would not be able to resolve at compile-time the function `oidmaptable`. The signature of the `oidmaptable` function that we want to use is `oidmpatable:`*`MTPNM->TPNM`*$_1$`....`*`TPNM`*$_n$. By declaring `v` to be of type *`MTPNM`*, AMOS is able to resolve the `oidmaptable` function using the type of `v` and V_1 to V_n respectively.

We are now ready to define the type extent function and the type predicate function.

```
create function MTPNM()->MTPNM as
  select oidtranslate(V1, ..., Vn)
  for each TPNM1 V1,...,TPNMn Vn where
     MPTPN_key()=<V1, ..., Vn>;
```

Example 88: Type extent function template for mapped types.

```
create function MTPNMp(Object o)->Boolean as
  select true for each MTPNM v, TPNM1 V1,...,TPNMn Vn
  where  o=v and oidmaptable(V1, ..., Vn) and
         MTPNM_key()=<V1, ..., Vn>;
```

Example 89: Type predicate function template for mapped types.

For the mapped type `Person` the type extent and type predicate functions appear thus:

```
create function person()->Person as
  select oidtranslate(p1,p2) for each APerson p1, BPerson p2
  where person_key()=<p1,p2>;
create function personp()->Boolean as
  select true for each Person v, APerson p1, BPerson p2
  where  o=v and oidmaptable(v)=<p1,p2> and
         person_key()=<p1,p2>;
```

Example 90: Type extent function and type predicate function for mapped type `Person`.

Finally we look at the converters. The purpose of a converter is to map back and forth between a mapped object and a defining object.

```
create function CNMi(MTPNM v)->TPNMi as
  select Vi for each TPNM1 V1,...,TPNMn Vn
  where oidmaptable(v)=<V1, ..., Vn>;
```

Example 91: Template for converter definition.

For our example two converters will be defined.

```
create function p2ap(Person v)->APerson as
  select p1 for each APerson p1, BPerson p2
  where oidmaptable(v)=<p1,p2>;
create function p2bp(Person v)->BPerson as
  select p2 for each APerson p1, BPerson p2
  where oidmaptable(v)=<p1,p2>;
```

Example 92: Converters for mapped type `Person`.

Given a `Person` object `p2ap` will return the corresponding `APerson` object and `p2bp` will return the `BPerson` object. The converters can also be used in the backward direction, returning a `Person` object *if* it has been created.

In [Fahl, 1994] a good description of mapped type for providing object-views of relational data can be found. Reasons for having the `oidtranslate` and the key function as separate functions are given and interesting optimizations are described. It should be noted that the `oidtranslate` function need not use an oidmaptable but could be computed. However, the function must be invertible since we want to map both between the OID of mapped objects and the elements of the key.

5.3 Late Binding and Derived Types

Late binding combined with derived types can be very useful. However, if allowed some problems are encountered that have to be solved. We will first show how late binding and derived types when combined allow for easy schema evolution. Next we will examine query processing in AMOS in order to understand the problems that we have to address.

5.3.1 Derived Types and Late Binding

Late binding can be very useful combined with derived types. It allows for very easy schema evolution in certain cases. Let us return to our `HeavyVehicle` example. Assume that we have the situation depicted in figure 5.1. `HeavyVe-`

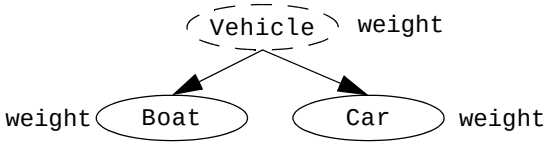


Figure 5.1: Example type hierarchy.

`hicle` can then be defined in two different ways either as in example 93 or as in example 94.

```
create derived type HeavyVehicle as
  [Boat b where weight(b)>10] or
  [Car c where weight(c)>10];
```

Example 93: Possible definition of derived type `HeavyVehicle`.

```
create derived type HeavyVehicle as
      [Vehicle v where weight(v)>10];
```

Example 94: Alternative definition of derived type `HeavyVehicle`.

If `HeavyVehicle` is defined as in example 93 then the extent will always be defined in terms of the extents of `Boat` and `Car` respectively. However, if `HeavyVehicle` is defined as in example 94 then the extent of `HeavyVehicle` will be defined in terms of subtypes of `Vehicle` that has the property `weight` defined[5].

For example if we define a new type `Bike`, with the property `weight`, as a subtype of `Plane`, then planes weighing more than 10 tons will be included in the extent of `HeavyVehicle`. Note, no modification of `HeavyVehicle` was needed in order to incorporate `Plane` as well. If we had defined `HeavyVehicle` according to example 93, then we would have needed to redefine it if we wanted to incorporate `Plane`.

The ability to choose whether to define a union type as in example 93 or as in example 94 provides the user with greater flexibility in the modelling process. He can decide whether a certain union type should only consider some types or whether it should evolve as new subtypes are added.

Another example where late binding is necessary is depicted in figure 5.2. Here

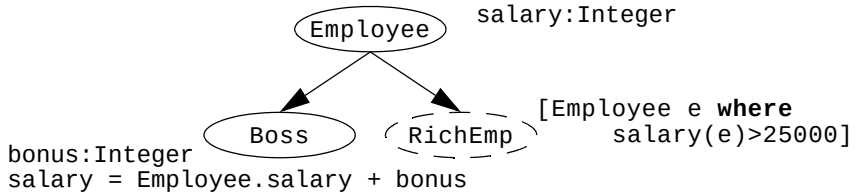


Figure 5.2: Late binding combined with specialization type.

we have defined a specialization type `RichEmp`, that is all employees that earn more than 25000. Ordinary employees just have a monthly salary but the salary for bosses is the monthly salary plus a bonus. When selecting which employees should be members of the `RichEmp` extent, we have to choose different `salary` functions depending on whether the employee is just an employee or if (s)he is a boss.

It should be evident by now that late binding combined with derived types is a useful feature. However, it also causes some problems which we will review in

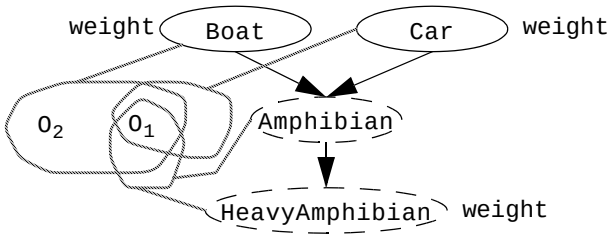
5. With proper result type.

the next section.

5.3.2 Problems with Derived Types and Late Binding

As we have seen, late binding can be very useful combined with derived types, however, it also causes some problems if we are not careful.

Let us again examine the `HeavyAmphibian` example. In figure 5.3 on page 80, the extents of the different types are shown. There exist two instances of type `Boat`, O_1 and O_2. O_1 also happens to be a member of the extent for types `Car`, `Amphibian` and `HeavyAmphibian`.




```
create derived type HeavyAmphibian as
  [Boat b where weight(b)>10] and
  [Car c where weight(c)>10];
```

Figure 5.3: Late binding problem.

Suppose that a user submits the following query.

```
select weight(b) for each Boat b;
```

Example 95: Retrieving the weight for boats.

Function `weight` has to be late bound in this case. For O_1 `weight:Boat->Integer` should be chosen, whereas `weight:HeavyAmphibian->Integer` should be chosen for O_2.

In example 96 the same query is shown after it has been type checked and optimized by AMOS.

```
select _g1 where
  typesof(b)[6]=#Boat[7] and
  dtr([HeavyAmphibian.weight->Integer,
       Boat.weight->Integer], b)=_g1;
```

Example 96: The type checked and optimized query.

The function call `weight(b)` has been replaced by a system-generated variable `_g1` and the `for each`-clause has been replaced by a call to function `typesof`. The `typesof(b)=#Boat` call will bind variable `b` to objects of type `Boat`.

6. Here `typesof` is executed in the backward direction.
7. `#Boat` denotes the OID for type `Boat`.

Since `weight` was late bound, AMOS has inserted a call to function `dtr` (*Dynamic Type Resolver*) [Flodin, 1996]. The `dtr` function will at run-time choose the correct resolvent to apply depending on the type of the object that variable `b` is bound to. The chosen resolvent will then be applied to `b` and `_g1` will be bound to the result of the application.

For function `dtr` to know what resolvent to choose, it first tests whether the object bound to `b` is of type `HeavyAmphibian`. It does so by executing the type predicate function. However, executing the type predicate function involves an identical call to `dtr` since `weight` in `[Boat b where weight(b)>10]` would be late bound. This would cause a loop.

A possible solution to the problem would be to not allow late binding in the type expressions defining derived types. However, this would render the examples in section 5.3.1 void. The approach taken here is to disregard properties having the derived type as argument type when compiling the type expression defining the derived type.

Having solved one problem that caused a loop, we address the next problem that can cause loops. In figure 5.4 a type hierarchy is shown for a database stor-

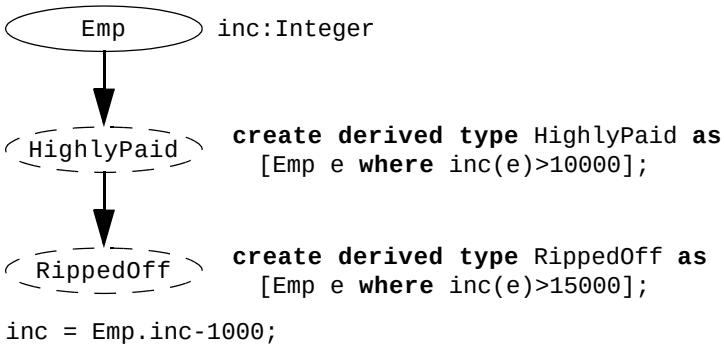


Figure 5.4: Example hierarchy with derived types.

ing information about employees (type `Emp`) and their income (property `inc`). We have defined two derived types, `HighlyPaid` and `RippedOff`. `HighlyPaid` is all those employees who have an income greater than 10000. `RippedOff` is all `HighlyPaid` whose income is greater than 15000. For `RippedOff` we redefine the `inc` property to be derived. The `inc:RippedOff->Integer` deducts a special tax from the income reducing it by 1000.

```
create Emp(inc) instances :e1(12000), :e2(15500);
```

Example 97: Creating two employees.

The decision we made above, to disregard properties having the derived type as argument when compiling the type expression defining the derived type, seems sensible when we look at the type expression defining type `RippedOff`. The function `inc` that should be used is clearly `inc:Emp->Integer` and

`inc:RippedOff->Integer` should not be considered.

Assume that a user wishes to check whether a certain employee is highly paid or not. This could be accomplished by calling the type predicate function for `HighlyPaid` as:

```
highlypaidp(:e2);
```

Example 98: Checking whether `:e2` is highly paid or not.

Let us look at what happens when this function is called. The compiled function `highlypaidp` looks like

```
highlypaidp(o) =
  select true where
      typesof(o)=#Emp and
      dtr([RippedOff.inc.->Integer,
           Emp.inc->Integer], o)= _g7 and
      _g7 > 10000;
```

Example 99: Compiled code for function `highlypaidp`.

When this function is called we first check to see if the object is of type `Emp` by calling the `typesof` function. Since :e2 is of type Emp we proceed by calling dtr. Function `dtr` now has to choose the right resolvent of `inc`. This is accomplished by invoking the type predicate function for the argument types for each of the possible resolvents in order from left to right. As soon as a type predicate function return true, we choose the corresponding resolvent. In our case we would first issue the call `rippedoffp(:e2)`.

```
rippedoffp(o)=
  select true where
      typesof(o)=#HighlyPaid and
      Emp.inc->Integer(o)=_g10 and
      _g10 > 15000;
```

Example 100: Compiled code for function `rippedoffp`.

The first thing that happens when we enter `rippedoffp` is that we check that the argument `o` is of type `HighlyPaid` by calling function `typesof`. When `typesof` has to determine if an object is of a specific type, i.e. we know both the argument and the result of `typesof`, and the type is derived or mapped. Then `typesof` will apply the type predicate function of the type to the object, i.e. we will make the call `highlypaidp(:e2)`. If the result of the application is true, then `typesof` succeeds; otherwise it fails.

As we enter `highlypaidp` once more we will check `typesof(:e2)=#Emp` which is true. We then execute the `dtr` resulting in yet another call to `rippedofp` with the argument `:e2`. By now we are looping and could go on forever if we do not take steps[8]. Observe that if we could only get an answer, either true or false from `rippedofp`, the loop would be broken and we would be able to

8. The top loop call `highlypaidp(:e2)` does not count when we are looking for loops. Thus, we do not loop until we enter `rippedofp` the second time.

proceed. AMOS acknowledges that it is looping in this situation and assumes that `:e2` is of type `RippedOff` returning *true* as the answer to the second invocation `rippedoffp(:e2)`.

If we look at the stack after this assumption has been made we have the following invocations:

```
highlypaidp(:e2) The top-level call
  rippedoffp(:e2) Call made by dtr
      highlypaidp(:e2) Call made by typesof in rippedoffp
```

Example 101: Call stack just after we have assumed `rippedoffp(:e2)` is true.

Since AMOS assumed that `:e2` was of type `RippedOff`, `dtr` chooses to apply `RippedOff.inc->Integer` to `:e2` binding `_g7` to 14500[9]. The value of `_g7` is greater than 10000 which means that true will be returned from `highlypaidp`.

`typesof(:e2)=#HighlyPaid` in `rippedoffp` succeeds and `Emp.inc->Integer` is applied to `:e2` binding `_g10` to 15500 which is greater than 15000. Thus `rippedoffp` succeeds and the `dtr` call in the top level call of `highlypaidp` can apply `RippedOff.inc->Integer` to `:e2`, binding `_g7` to 14500 which is greater than 10000. As a result the top-level call of `highlypaidp` returns true, the answer expected.

Let us take a brief look at the case where we check if `:e1` is an `HighlyPaid` instance.

```
highlypaidp(:e1);
```

Example 102: Check whether `:e1` is highly paid.

Everything will be the same as for `highlypaid(:e2)` to the point where we make the assumption that `:e1` is of type `RippedOff`. The `dtr` call in the second invocation of `highlypaidp` will then invoke `RippedOff.inc->Integer` with `:e1` as argument. `_g7` will be bound to 11000, which is greater than 10000. The second invocation of `highlypaidp` will return true. Thus, `typesof(:e1)=#HighlyPaid` in `rippedoffp` will succeed and `Emp.inc->Integer(:e1)` will bind `_g10` to 12000. However, 12000 is not greater than 15000 and `rippedoffp` will fail. The `dtr` call in the top level invocation of `highlypaidp` will then apply `Emp.inc->Integer` (the correct resolvent) to `:e1`[10], binding `_g7` to 12000 which is greater than 10000. Thus, the top level invocation of `highlypaidp` will return true, which is correct.

As we have seen, the problem is that there may exist a mutually recursive dependency between type predicates, causing a loop. When AMOS has compiled a type predicate it may look something like the template in example 103 on page 84. First comes a number of expressions $\texttt{expr}_1$ to $\texttt{expr}_k$ that may

9. 14500 = 15500 - 1000
10. After having executed `empp(:e1)` which succeeds.

involve the argument o. Should any of these expressions fail, then we never reach the typesof or dtr function call which might cause a loop and the typepredp function simply fails. However, if $expr_1$ to $expr_k$ do not fail, we have to execute the typesof or dtr call before we can execute $expr_m$ to $expr_n$. If $expr_m$ to $expr_n$ all succeed for a given argument o, we know that o is a member of the derived type. Likewise if some expression of $expr_m$ to $expr_n$ fails, we can say for certain that the argument o is not an instance of the derived type.

Since it is the *second* recursive invocation of typepredp we assume returns true, it does not matter if we are wrong, since we will fail on some of the expression $expr_m$ to $expr_n$ in the *first* invocation and thus typepredp will fail as it should.

```
typepredp(o) =
  select true where
      expr1
       ...
      exprk
      typesof or dtr call causing loop
      exprm
       ...
      exprn
```

Example 103: Template for compiled type predicates.

For this reasoning to be valid users must not specify *tautologies* when defining derived types. In figure 5.5 an example of this is shown. Type A has two proper-

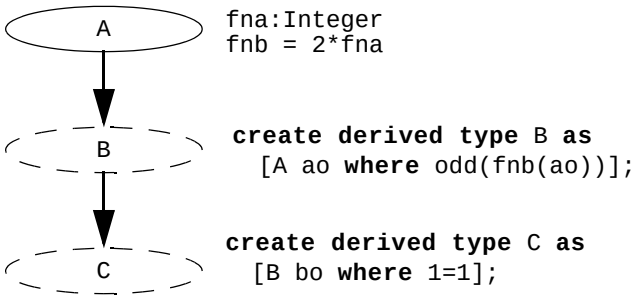


Figure 5.5: Example of a harmful tautology.

ties fna and fnb. fnb is specified in terms of fna in such a way that it is always guaranteed to be *even*. Then we create the derived type B as being all instances of A for which fnb is *odd*. The extent of B should be empty since fnb is never odd. Next we define derived type C as being all instances of B for which 1=1 . Property fnb is also overridden and C.fnb->Integer is guaranteed to always be odd. When C.fnb->Integer is defined, the fnb in odd(fnb(ao)) becomes late bound causing a mutual recursive dependency between the type predicate bp and cp. Now suppose that we have created an

instance of type A as

```
create A(fna) instances :a(1);
```

Example 104: Creating an instance of type A.

If we now asked if :a was an instance of type B, the answer would be *true* which is counter-intuitive. The reason is that the call bp(:a) would result in cp being invoked recursively twice with the argument :a. On the second invocation AMOS would assume that cp(:a) should return true, hoping that if the assumption was wrong, it would be discovered later by the rest of the condition in the where-clause defining derived type C. However, this condition says 1=1 which is always true. Thus, no assumption made can ever fail and we can prove anything.

To conclude, derived types combined with late binding are very useful as we saw in section 5.3.1. However, we must be careful about how we define our derived types or we may end up with paradoxes. A more careful study should be conducted and a formal framework defined allowing us to define what definitions of derived types can be considered safe.

5.4 Multidatabase Queries

In this section we briefly describe how multidatabase queries are treated in AMOS.

Let us return to our integration example in section 4.3. Suppose that a user wants to retrieve the name and the salary for all bosses.

```
select name(b), salary(b) for each Boss b;
```

Example 105: Retrieving the name and salary for all bosses.

When a query or a function definition is submitted to AMOS it is first flattened. Flattening means that nested function calls are removed by introducing new variables and derived functions are substituted for their definitions. The query in example 105 after flattening is seen in example 106.

```
select nm,sal
for each Boss B, String nm, Integer sal, Integer i1,
         Integer i2,Person@:db1 db1o, Person@:db2 db2o
where p2db1p[11](b)=db1o and
      name@:db1(db1o)=nm and
      p2db2(b)[12]=db2o and
      salary@:db2(db2o)=i1 and
      misc@:db2(db2o)=i2 and
      plus(i1,i2)=sal;
```

Example 106: Query after flattening.

11. Left for readability; is replaced by its definition in reality.
12. Left for readability; is replaced by its definition in reality.

We see how the name and salary functions have been replaced by their definitions and how these definitions have been flattened. Next AMOS type checks the query and if no type errors are found, the query is optimized for best possible performance. The optimizer will rearrange the expressions in the `where`-clause to minimize the response time.

Communication can add significantly to the cost of executing a query and therefore it has to be carefully optimized. One way to reduce communication is to build *chunks* of expressions. A chunk is a number of expressions that can be executed together by a data source.

Since AMOS transforms queries into disjunctive normal form before optimization, a chunk in AMOS is a number of conjuncts. Therefore, for each data source defined, we record whether it supports conjunction or not. If a data source supports conjunction we may group expressions together and thereby reduce communication. Whether it is beneficial or not to build a chunk has to be determined by the query optimizer based on the *expected cost* of the execution plan. Due to the large search space a randomized search method [Näs, 1993] or a heuristic search method [Litwin and Risch, 1992] have to be used.

Communication can also be reduced further if *universal functions* are supported. A universal function is a function that has the same functionality at different data sources. For example, the function `plus` performing addition has the same functionality in all AMOSs and even in relational databases. Another example would be comparison operators. For each data source we record what universal functions it supports. This enables the optimizer to move execution of universal functions to data sources that support them if it improves the execution plan.

Suppose that our query, after optimization, appears thus:

```
select nm,sal
for each Boss B, String nm, Integer sal, Integer i1,
         Integer i2,Person@:db1 p1, Person@:db2 p2
where oidmaptable[13](b)=<p1,p2> and
      name@:db1(p1)=nm and
      salary@:db2(p2)=i1 and
      misc@:db2(p2)=i2 and
      plus@:db2(i1,i2)=sal;
```

Example 107: Query after optimization.

We see how some expressions have been rearranged and also how the optimizer has chosen to invoke the function `plus@:db2` instead of `plus`. This reduces the communication since addition now is performed at `DB2` and only the result has to be sent back. If we had executed `plus`, we would have to send back both the operands to `plus`.

The optimized query is fed to the *plan transformer* that replaces chunks with

13. The expanded form of the two converter function calls in example 106.

function calls and ships chunks to appropriate data sources. In our example the last three expressions in the where-clause should all be executed at DB2 and DB2 supports conjunction. Thus, the last three expressions constitute a chunk.

After plan transformation our query becomes

```
select nm,sal
for each Boss B, String nm, Integer sal, Integer i1,
         Integer i2,Person@:db1 db1o, Person@:db2 db2o
where p2db1p(b)=db1o and
      p2db2(b)=db2o and
      name@:db1(db1o)=nm and
      db2_chunk(db2o)=sal;
```

Example 108: Query after plan transformation.

At DB2 a function corresponding to the chunk has been created.

```
create function db2_chunk(Person p)->Integer as
  select sal for each Integer sal, Integer i1, Integer i2
  where  salary(p)=i1 and
         misc(p)=i2 and
         plus(i1,i2)=sal;
```

Example 109: Function created at DB2.

Functions corresponding to chunks in a query plan are called *chunk functions*. A chunk function is optimized and compiled when it is created. In an AMOS a cache of chunk functions exists. If the same or a different query needs the same chunk function and it has not been replaced in the cache, it is just invoked without the overhead of optimization and compilation.

5.5 Concluding Remarks

In this chapter we have described how virtual types are implemented in AMOS. For derived types we described two implementation approaches, the materialized approach in section 5.1.1, and the derivation approach in section 5.1.2.

In AMOS the derivation approach is used and in section 5.1.2 we described how the type extent function and the type predicate function are constructed for derived types. A similar description for derived types was given in section 5.2.

In section 5.3.1 we looked at late binding combined with derived types and showed how this can be helpful. However, late binding combined with derived types causes some problems. A description of these problems was given in section 5.3.2.

The approach taken by COOL* and MultiView, described in chapter 2, to avoid the problems described in section 5.3.2 is to disallow overriding of functions. A function cannot be declared on a class if it already exists. Suppose that we want to redefine a function *f* for a class *C*. We must then first *hide* the function which means that the class, *C'*, resulting from the hide operation will have a type that is a supertype of the type of *C*. When we define the new function *f'* on *C'*, we get a new class *C''* whose type will be a subtype of the type of *C'*. Thus, *C* and *C''* will become siblings in the class hierarchy and will not be comparable.

We feel that this approach is too restrictive and that we should try instead to determine whether the definition of a derived type is safe or not. If it is unsafe, then the system should issue an error message.

6 Summary and future work

In this last section of the thesis, a summary of the main contributions of this work and directions for future work are presented.

6.1 Summary

In this thesis we have discussed the problem of *information integration* in a multidatabase environment. Two different integration strategies were discussed, the *global schema approach* and the *multidatabase language approach*. The various advantages and restrictions of the two approaches were presented. In AMOS we opted for the multidatabase language approach and we discussed how to achieve information integration in an object-oriented multidatabase environment using *multidatabase queries* and *object views*.

Two important problems that have to be addressed when performing object-oriented information integration are resolving *structural differences* and the *objects equivalence problem*.

Since object-oriented data models are semantically rich, the same concept may be represented by different modelling constructs in different data sources. For example, what is a type in one data source might be modelled as a value of an attribute in another data source. To be able to perform a successful integration we must be able to map between different modelling constructs. In chapter 2 we reviewed proposals of how to achieve this using *virtual* or *derived classes*. A virtual class is a class whose extent is defined by a declarative query expression. In AMOS we have adopted this concept and we call it *derived types*. We described the semantics of derived types in chapter 4, and in chapter 5 we outlined the implementation.

Two key features in object-oriented data models are *inheritance* and *function overriding*. These features allow users to model a problem in a natural way. Function overriding requires functions to be *late bound* when the correct resolvent cannot be determined at compile-time. The resolution of late bound functions has to be performed at runtime based on the type of the actual argument supplied to the function. The systems described in chapter 2 providing virtual classes, disallowed, by the definition of their data models, the use of late bound functions in expressions defining virtual classes. In section 5.3.1 we gave some examples of the usefulness of derived types combined with derived types. Thus, in AMOS we decided to allow late binding in expressions defining derived types. However, this causes some problems. These problems were identified and described in section 5.3.2.

We also addressed the problem of object equivalence. Unifying types and *same*-functions was introduced by Pegasus and COOL* described in chapter 2 to address this problem. In AMOS we use *mapped type*. *Same*-functions define equivalence classes and avoid producing new objects when objects are merged. In Pegasus and AMOS a new type is created whose instances are new objects, each representing a number of merged objects. However, the mechanism of AMOS is more general than the one used in Pegasus. Mapped types cannot only be used for merging objects; we also use them to create object-oriented views of non object-oriented information, i.e. we can use mapped types for integrating EDS as well. In Pegasus the concept of *producer types* was used to accomplish this.

As well as discussing various integration strategies and object views we presented the architecture of our prototype platform AMOS. Also, we gave a short description of how multidatabase queries could be compiled and executed in AMOS.

6.2 Future Work

There are many issues remaining, both practical and theoretical, that need to be addressed to provide a complete framework for integration.

6.2.1 Multidatabase Queries

The query compiler of AMOS should be rewritten to support the syntax of multidatabase queries as presented in section 3.3.1. The syntax supported today is much more cumbersome, forcing users to specify complete query expressions to be remotely evaluated.

The optimizer of AMOS also needs to be rewritten to build chunks, support universal functions, and estimate the cost of communication. Today functions are built for the complete query expression that should be remotely evaluated. These functions are treated as black boxes and the optimizer currently has no opportunity to optimize them.

6.2.2 Initializors and constructors

Since a user has to specify explicitly, in constructors and initializors, what type new objects should become instances of, inconsistencies may arise if the wrong type is specified. A foolproof method of declaring initializors and constructors should be provided.

Also, the current implementation of constructors and initializors has not been tuned for performance. Hand-coded examples have shown that rewriting the compiler for intializors and constructors can result in significant efficiency gains when creating new objects.

6.2.3 Formalization

A formal model of the data manipulation language and the query language of AMOS should be defined. This is necessary if we want to be able to determine whether a definition of a derived type is safe or not, i.e. if we will encounter the problems described in section 5.3.2.

A formal model would also allow us to study the problem of subsumption further. Being able to decide whether one expression subsumes another allows us perform better placement of derived types in the type hierarchy. Also, we would be able to improve query optimization. Today, queries may contain several statements that subsume each other, e.g. a variables value should be greater than 15 and also greater than 20. Clearly, if the value is greater than 20 it is also greater than 15. Thus, we would be able to remove a number of expressions from queries, resulting in better response time. In multidatabase queries we could expect even greater savings, as removal of expressions from a query may result in less communication, which reduces response time significantly.

6.2.4 Extended view mechanism

The view mechanism of AMOS should be extended to allow definition of view schemas as in MultiView. This would allow us to hide information. Today the whole schema is visible at all times. The view mechanism could also be used for authorization purposes if user entities were introduced.

The view mechanism should also be extended with support for parameteriezed views, as described in section 2.5, to allow us to resolve all structural differences in a declarative manner. The current implementation should be relatively easy to extend since functions are used to implement type extent functions and type predicate functions. We would have to introduce three new types, `TypeSchema`, `FunctionSchema`, and `ObjectSchema` whose instances would be schemas declared by the user.

Since derived functions are used to implement object views, the schema parameters would simply become extra arguments to the functions. However, compilation and optimizations of queries using parameteriezed views is a challenge.

6.2.5 Level 3 integration support

The query compiler should also be modified to support full level 3 integration, i.e. it should be possible to create stored functions whose argument and result types may stem from different databases.

One difficulty with full level 3 integration is that it creates inter-database dependencies. These decrease the autonomy of the AMOS servers. Objects referred by other AMOS servers cannot be deleted. This means that a server containing much used information may run out of memory since the information it contains are referred by many other servers. In the KIWIS system [Ahlsén, 1995] the concept of *contract* is introduced. When information is exported

by some server to a client, the server and the client agree on a contract specifying the duration of the information exportation. When the contract expires, the server is free to do as it pleases with the exported information. Contracts appear to be a good solution to the problem.

6.2.6 Tools for integration

As we have seen, we can integrate information using virtual types. However, currently users have to do quite a bit of "programming". A higher level abstraction should be provided that allow users to specify what information to integrate instead of how this information should be integrated, i.e. a declarative integration specification is required. An example of such a mechanism is the proposal for automatic importation of relational schemas in Pegasus [Albert et al., 1993].

References

[Ahlsén, 1995] M. Ahlsén. *The Federation as a Model for Information Systems Architecture*. Licentiate Thesis, DSV No. 95-049, Department of Computer & Systems Science, Stockholm University, October 1995.

[Ahmed *et al*., 1991a] R. Ahmed, P. DeSmedt, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan. Pegasus: A System for Seamless Integration of Heterogeneous Information Sources. In *Proceedings of the 36th IEEE Computer Society International Conference - COMPCON Sping '91*, San Francisco, 1991.

[Ahmed *et al*., 1991b] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan. The Pegaus Heterogeneous Multidatabase System. *IEEE Computer*, Vol. 24, No. 12, December, 1991.

[Ahmed *et al*., 1993] R. Ahmed, J. Albert, W. Du, W. Kent, W. Litwin, M.-C. Shan. An Overview of Pegasus. In *Proceedings of the Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, RIDE-IMS '93, Vienna, Austria, April 1993.

[Albert *et al*., 1993] J. Albert, R. Ahmed, M. Ketabchi, W. Kent, M.-C. Shan. Automatic Importation of Relational Schemas in Pegasus. In *Proceedings of the Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, RIDE-IMS '93, Vienna, Austria, April 1993.

[Ameli and Dujardin, 1995] E. Ameli and E. Dujardin. *Supporting Explicit Disambiguation of Multi-Methods*. Research Report n2590, Inria, 1995.

[Bertino, 1991] E. Bertino. Integration of Heterogeneous Data Repositories by Using Object-Oriented Views. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991.

[Bright *et al.*, 1992] M. W. Bright, A. R. Hurson, and S. H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *IEEE Computer*, vol. 25, no. 3, March, 1992.

[Brodie and Stonebraker, 1992] M. L. Brodie and M. Stonebraker. DARWIN: On the Incremental Migration of Legacy Information Systems. TR-0222-10-92-165, GTE Laboratories, March 1993.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In *ACM Computing Surveys*, Vol. 17, No. 4, 1985.

[Catell, 1992] R. G. G. Catell. *Object Data Management*, Addison-Wesley Pub-

lishing Company, 1992.

[Ceri *et al*., 1992] S. Ceri, G. Gottlib, L. Tanca. What You Always Wanted to Know about Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 1, March, 1989.

[Chimenti *et al*., 1989] D.Chimenti, R.Gamboa, R.Krishnamurthy. Towards an Open Architecture for *LDL*. In *Proceedings of the 15th VLDB Conference*, pp. 195-205, 1989.

[Chomicki and Litwin, 1994] J. Chomicki and L. Litwin. Declarative Definitions of Object-Oriented Multidatabase Mappings. M.T. Özsu, U. Dayal, P. Valduriez (eds.): *Distributed Object Management*, Morgan Kaufmann Publisher, Inc., 1994.

[Codd, 1970] E. F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, Vol. 13, No. 6, pages 377-387, 1970.

[Connors and Lyngbaek, 1988] T. Connors and P. Lyngbaek. Providing Uniform Access to Heterogeneous Information Bases. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, Bad Munster am Stein-Ebernburg, FRG, September 1988, Lecture Notes in Computer Science, No. 334, 1988.

[Date, 1986] C. J. Date. *An Introduction to Database Systems*, Volume I. Fourth Edition, Addison-Wesley Publishing Company, 1986.

[Dayal, 1989] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, Glenden Beach, Oregon, USA, June 1989.

[Eliassen and Karlsen, 1991] F. Eliassen and R. Karlsen. Interoperability and Object Identity. *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[Elmasri and Navathe, 1994] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems* (2nd Ed.). ISBN 0-8053-1753-8, The Benjamin/Cummings Publishing Company, Inc., 1994.

[Fang *et al*., 1993] D. Fang, S. Ghandeharizadeh, D. McLeod, A. Si. The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database Systems. In *Proceedings of the 1993 IEEE 9th International Conference on Data Engineering*, Vienna, Austria, 1933.

[Fang *et al*., 1992] D. Fang, J. Hammer, D. McLeod. A Mechanism and Experimental System for Function-Based Sharing in Federated Databases. In *Proceedings of the IFIP WG2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, Victoria, Australia, 1992.

[Fahl, 1994] G. Fahl. *Object Views of Relational Data in Multidatabase Systems*. Licentiate Thesis No 446, Dept. of Computer and Information Science, Linköping University, 1994.

[Fishman *et al*., 1989] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.

A. Neimat, T. Risch, M. C. Shan, W. K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F.H. Lochovsky (eds.): *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1989.

[Flodin, 1994] S. Flodin. *An Incremental Query Compiler with Resolution of Late Binding*. Research Report LiTH-IDA-R-94-46, Department of Computer and Information Science, Linköping University, 1994.

[Flodin, 1996] S. Flodin. *Efficient Management of Object-Oriented Queries with Late Binding*. Licentiate Thesis No 538, Department of Computer and Information Science, Linköping University, 1996.

[Flodin *et al*., 1996] S. Flodin, J. Karlsson, T. Risch, M. Sköld, M. Werner. *AMOS User's Guide*. Available at URL `http://www.ida.liu.se/labs/edslab/amos/amosdoc.html`, 1996.

[Flodin and Risch, 1995] S. Flodin, T. Risch. Processing Object-Oriented Queries with Invertible Late Bound Functions. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15, 1995.

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

[Heiler and Siegel, 1991] S. Heiler and M. Siegel. Heterogeneous Information Systems: Understanding Integration. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991.

[Ioannidis and Kang, 1990] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the ACM SIGMOD Conference*, Atlantic City, 1990.

[Kent, 1988] W. Kent. The Many Forms of a Single Fact. Technical Report HPL-SAL-88-8, Hewlett-Packard, 1988.

[Kent, 1991] W. Kent. The Breakdown of the Information Model in Multi-Database Systems. *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[Krishnamurthy *et al.*, 1991] R. Krishnamurthy, W. Litwin, and W. Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver Colorado, May 1991.

[Kuno and Rundensteiner, 1993] H. A. Kuno and E. A. Rundensteiner. Implementation Issues with Building an Object-Oriented View Management System. Tech. Rep. CSE-TR-191-93, Electrical Engineering and Computer Science Dept., Computer Science and Engineering Division, University of Michigan, Ann Arbor, August 1993.

[Landers and Rosenberg, 1991] T. Landers and R.L. Rosenberg. An Overview of

Multibase. In *Distributed Data Bases*, H.-J. Schneider (ed.), North-Holland Publishing Company, 1992.

[Laasch and Scholl, 1993] C. Laasch and M. H. Scholl. A Functional Object Database Language. In *Proceedings of the 4th international Workshop on Database Programming Languages*, New York City, August 1993.

[Litwin and Abdellatif, 1986] W. Litwin and A. Abdellatif. Multidatabase Interpretability. *IEEE Computer*, Vol. 19, No. 12, 1986.

[Litwin *et al.*, 1990] W. Litwin, L. Mark and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[Litwin and Risch, 1992] W. Litwin and T. Risch. Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. In *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December, 1992.

[Lyngbaek *et al*., 1991] P. Lyngbaek and the OODB Team in CSY. OSQL: A Language for Object Databases. Technical Report HPL-DTD-91-4, Hewlett-Packard Company, January, 1991.

[Milliner *et al*., 1995] S. Milliner, A. Bouguettaya, M. Papazoglou. A Scalable Architecture for Autonomous Heterogeneous Database Interactions. In *Proceedings of the 21st International Conference on Very Large Databases*, Zürich, Switzerland, September, 1995.

[Motro, 1987] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 7, July, 1987.

[Näs, 1993] J. Näs. *Randomized Optimization of Object-Oriented Queries in a Main Memory Database Management System*. Master's Thesis, LiTH-IDA-Ex-9325, Linköping University, 1993.

[Ra and Rundensteiner, 1995] Y.-G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In *Proceedings of the 1995 IEEE 11th International Conference on Data Engineering*, Taipei, Taiwan, 1995.

[Rabitti *et al*., 1991] F. Rabitti, E. Bertino, W. Kim, D. Woelk. A Model of Authorization for Next-Generation Database Systems. *ACM Transactions On Database Systems*, Vol. 16, No.1, March 1991.

[Rafii *et al*., 1991] A. Rafii, R. Ahmed, P. deSmedt, B. Kent, M. Ketabchi, W. Litwin and M.-C. Shan. Multidatabase Management in Pegasus. In *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, April 1991.

[Rundensteiner,1992] E. A. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases*, Vancouver, Canada, August, 1992.

[Saltor *et al*., 1991] F. Saltor, M. Castellanos and M. García-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[Scholl *et al.*, 1992] M. H. Scholl, C. Laasch, C. Rich, H. J. Schek, M. Tresch. The COCOON Object Model. Technical Report No. 192, Department of Computer Science, ETH Zürich, December, 1992.

[Shaw and Zdonik, 1989] G.M. Shaw and S.B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, USA, 1990.

[Sheth and Larson, 1990] A.P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[Shipman, 1981] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981.

[Sköld, 1994] M. Sköld. *Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques*. Licentiate Thesis No 452, Department of Computer and Information Science, Linköping University, 1994.

[Sköld and Risch, 1996] M. Sköld and T. Risch. Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Condition, To be presented at the *The 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, Louisiana, February 1996

[Strostrup, 1991] B. Strostrup. *The C++ Programming Language* (2nd edition). Addison-Wesley Publishing Company, 1991.

[Tresch and Scholl, 1994] M. Tresch and M. H. Scholl. A Classification of Multi-Database Languages. *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[Ullman, 1988] J. D. Ullman. *Database and Knowledge-Base Systems*, Volume I & II, Computer Science Press, 1988, 1989.

[Waldo et al., 1994] J. Waldo, G. Wyant, A. Wollrath, S. Kendall. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems Laboratories, 1994.

[Wang and Madnick, 1990] Y. R. Wang and S. E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proceedings of Very Large Databases*, Brisbane, Australia, August 1990.

[Widom and Ceri, 1996] J. Widom and S. Ceri (editors). *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

[Wiederhold, 1992] G. Wiederhold. Mediators in the Architecture of Future Informations Systems. *IEEE Computer*, March 1992.

Index