UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS

JOSÉ SAMOS JIMÉNEZ

DEFINITION OF EXTERNAL SCHEMAS AND DERIVED CLASSES IN OBJECT ORIENTED DATABASES

TESI DOCTORAL

DIRIGIDA PEL DR. FÈLIX SALTOR I SOLER

BARCELONA

1997

Memòria presentada per José Samos Jiménez

per tal d’aconseguir el grau de Doctor en Informàtica

per la Universitat Politècnica de Catalunya

A mis padres

Agradecimientos

Agradezco muy especialmente al Dr. Fèlix Saltor por haber aceptado dirigir esta tesis, por su confianza, apoyo, comentarios y consejos: por su ejemplo profesional y humano que siempre tendré presente.

A Antoni Olivé, Elisa Bertino, Oscar Díaz, Jesús García y Toni Urpí por honrarme aceptando formar parte del tribunal.

A los revisores de versiones previas de este trabajo y de las publicaciones realizadas durante su desarrollo, por sus comentarios.

Quiero agradecer a los miembros de LSI, especialmente a Laura Espitia, su ayuda en todo momento.

Agradezco a Winterthur Seguros las posibilidades profesionales que me ha ofrecido, resultando éstas de gran utilidad para el desarrollo de la tesis. A todos los compañeros que han vivido día a día mis comentarios sobre los avances en este trabajo, por el interés y paciencia demostrados.

Finalmente, agradecer la colaboración al resto de personas que hayan participado de alguna manera en la elaboración de esta tesis.

Este trabajo ha recibido parcialmente ayuda material del programa CICYT, proyecto TIC93-0436.

Table of contents

1 Introduction

1.1 Motivation

The ANSI/SPARC *three-level architecture* classified database fuctions in physical, logical, and external levels; information at these levels is represented by the internal, conceptual, and external schemas respectively.

External schemas offer views of the information contained in the conceptual schema. They allow the end-users to concentrate on a logical representation of data adapted to their particular requirements. External schemas provide *logical data independence* (many aspects of the conceptual schema may be changed without having to modify the views of the conceptual schema offered by external schemas).

The three-level architecture has been widely applied in relational databases. In object-oriented databases (OODBs), the conceptual schema and the internal schema have also been studied deeply; this is not the case for external schemas. Nonetheless, logical data independence is also a requirement for OODBs.

Therefore, the main target of this thesis is to broaden the study of external schemas in OODBs, in particular, the external schema definition process. The definition of external schemas in OODBs has been previously studied by other authors, but, from our point of view, still there are some issues without a satisfactory solution. Some of these issues are further studied in this work and new solutions to them are proposed.

In OODBs, external schemas can contain classes from the conceptual schema as well as *derived classes* defined from previously existing classes (derived or non-derived); derived classes offer views of the information contained in the classes from which they are defined. The definition of derived classes is an important issue in the definition of external schemas in OODBs. Therefore, the definition of derived classes is our second focus of research.

One of the main uses of external schemas is to provide a mechanism that support the simulation of schema changes. Information in external schemas has to be derived from the conceptual schema; the kind of schema changes that can be simulated using external schemas is conditioned by this fact. Therefore, our third target is to present a mechanism that supports the simulation of a wider spectrum of schema changes. This mechanism is based on the definition of external schemas but incorporates some additional extensions.

1.2 Structure of the thesis

State of the art

In chapter 2 a review and a classification of existing external schema definition methodologies is presented. Most of the methodologies studied do not consider explicitly the ANSI/SPARC framework, but propose similar architectures. An effort is made in this sense, and these methodologies are presented using the ANSI/SPARC terminology. Even the few methodologies that reference the ANSI/SPARC architecture, propose systems that do not totally coincide with the definitions of this architecture.

Different issues dealing with the definition of derived classes, as they are treated by other authors, are presented as well. Derived classes can be included in external schemas. In some outstanding points about the definition of derived classes, different alternatives have been proposed. Specifically, the integration of derived classes with other classes in an object schema, the possibility of defining derived classes with object preserving and object generating semantics, the problems in the generation of identifiers for new objects and the transmission of modifications between the objects in base classes and derived classes.

We consider that the problems presented in this chapter are not satisfactorily solved, and alternative proposals are put forward in chapters 5, 6 and 7.

Object-oriented concepts

The concepts used in this work do not refer to any particular object oriented model; they are general concepts applicable to most existing object models. In chapter 3, the formal definition of the basic object oriented concepts used in our proposal is presented, as well as some additional outstanding concepts used by other authors.

Definition of DCMs of OODBs

In chapter 4 the definition of deductive conceptual models (DCMs) using Prolog in order to specify different aspects of OODBs is proposed. The result of the specification process using this technique is an executable prototype of the system. Having a prototype directly available, along with the system specifications, is particularly useful in order to define additional elements in the context of OODBs (e.g. schema evolution, definition of derived classes, definition of external schemas). The use of this technique is proposed mainly due to the difficulty of building prototypes of the mentioned elements over commercial OODBs. This technique is used in chapter 5 in order to define some of the algorithms proposed there.

A new external schema definition methodology

In chapter 5, a new external schema definition methodology that considerably simplifies the process of definition and the results obtained, is presented. The ANSI/SPARC framework is taken as a reference. The systems of conceptual and external schema definition are based on a *data dictionary*. The universe of discourse of the data dictionary is all information in the management and use of the database system -including the management and use of schemas.

In this approach the process of integration of derived classes has two phases: first, derived classes are integrated directly into the data dictionary by means of the derivation relationship and then, a set of classes that will compose the external schema is selected from the data dictionary. From this set of classes an external schema in which classes are integrated by means of the inheritance relationship is generated. The derivation relationship does not appear in either the conceptual schema or the external schemas, only in the data dictionary and it is not necessary to extend the object orientation paradigm in order to include it.

To carry out the process of generation of the external schema, two algorithms are defined: the basic algorithm and the extended algorithm. The classes of the set with which the extended algorithm works must be qualified as either *transformable* or *non-transformable*, indicating whether they can or cannot be modified automatically, in the sense of adding or removing properties. The extended algorithm automatically modifies as needed the transformable classes, hence avoiding the need to define explicitly all the classes that we want to include in the external schema.

Definition of derived classes

The conceptual schema can contain classes initially defined, and also classes defined from previously existing classes, i.e., derived classes. Sometimes, in order to adapt to final users' needs, the information contained in the conceptual schema's classes must be re-organised in the form of new classes: external schemas may contain conceptual schema classes as well as new derived classes. The classes from which a derived class is directly defined are its *base classes*; they can be derived or non-derived classes. A derived class may be defined either by *object preserving semantics*, if it only contains objects of its base classes; or by *object generating semantics*, if it contains new objects generated from the objects of its base classes. Defining derived classes by object generating semantics makes it possible to carry out sophisticated re-organisations of existing information which would otherwise be impossible -i.e., transformation of values into objects or aggregation of objects to form a new concept.

In chapter 6 we shall study two of the main problems of defining derived classes: the generation of identifiers for the objects of the derived classes; and the transmission of modifications between the objects of the derived classes and those of the base classes.

Schema evolution

External schemas are derived from the database conceptual schema; they can be used to simulate changes to the database conceptual schema. Sometimes the final users' information requirements change; they need new information which cannot be derived from the information previously contained in the database. Therefore, external schemas cannot be used to simulate this kind of schema changes.

In order to provide more flexibility in this area, in chapter 7 we propose the definition of derived classes that can contain non-derived information in their intension as well as in their extension: *partially derived classes*. When an external schema with non-derived information is to be defined, the conceptual schema has to be modified in order to include the non-derived information of the new schema. In order to avoid unnecessary modifications of the conceptual schema the use of a test environment for the definition of temporal external schemas is also proposed in chapter 7.

Conclusions

Finally, some conclusions and future work topics are presented in chapter 8.

2 Views: external schemas and derived classes

A *view* is a simplifying abstraction of a complex structure. In OODBs, some authors identify the term “view” with the concept of schema; others consider it just a class. In order to avoid confusion, in this work the term “view” will not be used. Instead the terms “external schema” or “derived class” will be used respectively. The ANSI/SPARC three-level schema architecture is adopted as a guide and consequently our terminology, concepts, and the terminology used in other referenced works, are adapted to it. The aim of this chapter is to provide insight into the existing methodologies of definition of external schemas and derived classes. In section 2.1 a brief review of the ANSI/SPARC framework is presented. Secondly, section 2.2 corresponds to a survey and a classification of existing external schema definition methodologies. Lastly, in section 2.3 different issues are presented concerning the definition of derived classes as they are treated by other authors.

2.1 The ANSI/SPARC framework

2.1.1 Three-level schema architecture

The ANSI/SPARC framework [ANSI/X3/SPARC, 1975] proposed a three-level architecture for DBMSs, presented in fig. 2.1. The *conceptual schema* is a logical representation of the reality modeled by the database; it describes the relevant aspects of the universe of discourse.

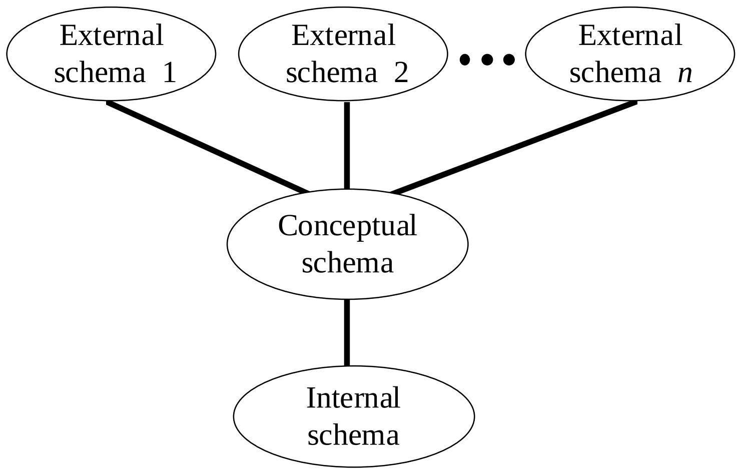


Figure 2.1. Three-level schema architecture.

Each external schema is derived from the conceptual schema and describes the part of the information appropriate for the group of users to whom it is addressed. Databases with external schemas are flexible and adaptable to changes according to how users view the data. External schemas provide logical data independence (many aspects of the conceptual schema may be changed without having to modify the views of the conceptual schema offered by external schemas).

The *internal schema* is a physical representation of the data stored into the database (it specifies what data is actually stored in the database, and how that data is stored). The distinction between the the conceptual schema and the internal schema provides *physical data independence* (many aspects fo the physical implementation may be changed without having to modify the abstract vision of the database).

In this work we focus our attention on the conceptual schema and the external schemas.

2.1.2 Schema definition systems

The systems of conceptual and external schema definition are based on a data dictionary [ANSI/X3/SPARC, 1986] as shown in fig. 2.2. The universe of discourse of the data dictionary is all the information relevant to the management and use of the database system -including the management and use of schemas.

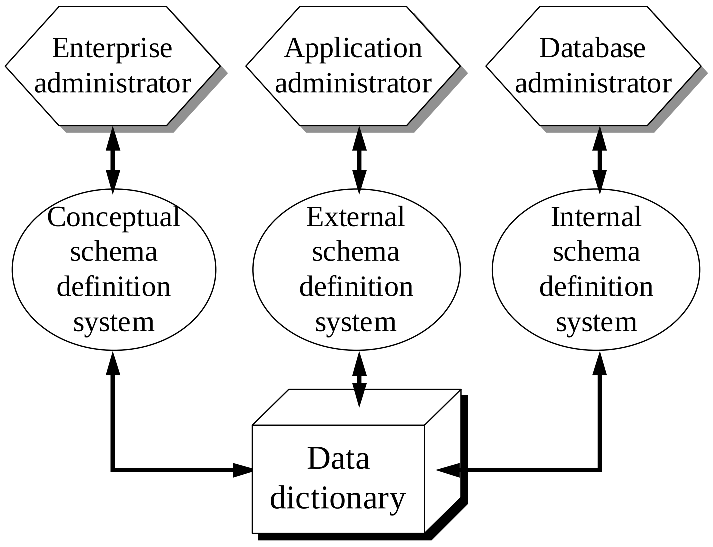

Enterprise administrator
Application administrator
Database administrator
Conceptual schema definition system
External schema definition system
Internal schema definition system
Data dictionary

Figure 2.2. Schema definition framework.

The definition of external schemas is carried out by the *external schema definition system*. The information contained in an external schema must be derivable from the information contained in the conceptual schema.

2.1.3 The ANSI/SPARC framework in OODBs

The ANSI/SPARC three-level schema architecture has been widely applied to relational databases. In OODBs, the conceptual schema and the internal schema have also been studied deeply; this is not the case for external schemas.

External schemas should have the same organisational structure as the conceptual schema from which they are defined; thus, in OODBs the conceptual schema and the external schemas should be object schemas.

In the OODB field only a few works deal explicitly with the definition of external schemas referring the ANSI/SPARC architecture [Barclay & Kennedy, 1993; Santos et *al*., 1994; Kim & Kelley, 1995; Bertino et *al*., 1996]. A common characteristic of most of the works about the definition of external schemas in this field is that they don't use the ANSI/SPARC terminology (including works that refer the ANSI/SPARC architecture).

2.2 External schema definition methodologies

External schemas should have the same organisational structure as the conceptual schema from which they are defined. That is to say, in the object-orientation paradigm they should be object schemas.

A requirement that external schemas have to fulfil is *schema closure* [Dayal, 1989]: every class referenced by some class included in an external schema has to be also included in the same external schema. A class is referenced by another class if it appears as the type of some argument of a method or as the domain of an attribute of the second class. In an external schema definition methodology some mechanism to verify the closure of the schema should be provided.

In order to review what has been done by others authors about the definition of external schemas, a survey of external schema definition methodologies is presented. Finally, in section 2.2.2 a classification of the different methodologies considered is made.

2.2.1 Survey of external schema definition methodologies

In this section, the main characteristics of some of the most relevant external schema definition methodologies are presented. In some of these works, the ANSI/SPARC architecture is referred to. However, each author uses his/her own terminology. This presentation is made using the ANSI/SPARC terminology, adapting its concepts to the particular concepts used in each methodology.

2.2.1.1 [Tanaka et **al.**, *1988]*

In the work of Tanaka, Yoshikawa & Ishihara [Tanaka et *al*., 1988] the definition process of external schemas is called *schema virtualisation*. Their system allows the definition of derived classes by object-preserving semantics; the definition of derived classes by object-generating semantics is proposed as a further research topic.

An external schema has to contain one or more derived classes. The steps in order to construct an external schema are as follows:

- Define all the derived classes in the external schema.

- Define manually all the inheritance relationships between the derived classes in the previous step; these definitions are automatically validated by the system.

After these steps an external schema is obtained: In some cases this can be the final result. In others, the following steps can also be done several times (as many as needed):

- Delete a subschema from a previously existing schema (conceptual or external).

- In order to obtain a new external schema, import the schema derived from the previous step and merge it with the external schema obtained. The merge process is also defined manually but validated automatically. Some of the classes referred to by some class in the external schema may not be included in the external schema (the external schema may not be closed). This is allowed in the proposed system; avoiding this situation is another research topic.

In this work [Tanaka et *al*., 1988], external schemas are called *virtual schemas*; derived classes are *virtual classes*; the conceptual schema is called *base schema*.

2.2.1.2 [Heiler & Zdonik, 1988]

In the system proposed by Heiler & Zdonik [Heiler & Zdonik, 1988], for defining an external schema two sets have to be specified: “a set of types and a set of objects that are instances of those types.” This is equivalent to specifying a set of classes.

An external schema is derived by applying functions to other external schema definitions in order to obtain the new external schema definition. In the data dictionary, external schema definitions are objects which have the functions necessary in order to obtain the new external schema associated. Therefore, the definition of an external schema consists of the definition of these functions.

“Each installation will define a *base view* from which all other views can be derived. One can think of the base view as the conceptual schema of the installation.” External schema derivation functions are defined according to the definition of the conceptual schema. External schemas can contain new derived classes not included in the conceptual schema.

Therefore, external schemas are defined from the conceptual schema and/or other external schemas by derivation functions. The definition of the external schema consists exclusively of the definition of the derivation functions; by means of these functions the classes that will compose the external schema are defined. External schemas have to be closed. However, it is not explained how this property is achieved.

2.2.1.3 [Dayal, 1989]

In the work of Dayal [Dayal, 1989] external schemas can contain classes imported from other schemas and also new derived classes. Derived classes can be defined by object-preserving and also by object-generating semantics.

Therefore, the steps in order to define an external schema are:

- Import of classes previously defined in other schemas.

- Definition of derived classes from previously existing classes (not necessarily included in this schema).

- Definition of inheritance relationships between the classes in the external schema.

It is mentioned that one problem in the external schema definition process is to guarantee closure of the external schema, but it is not indicated how this is achieved.

2.2.1.4 [Abiteboul & Bonner, 1991]

Abiteboul & Bonner presented in [Abiteboul & Bonner, 1991] a system for defining derived classes and external schemas. Derived classes can be defined with object-preserving and with object-generating semantics; external schemas can also include classes defined in other schemas and new defined classes.

The ANSI/SPARC architecture is not referred to in [Abiteboul & Bonner, 1991], but the architecture proposed there can be adapted to it: “In general, there can be *many* databases in a system. In such systems, one database can use data from other databases via *import* statements. A *view* can thus be thought of as a database that imports all its data from other databases. That is, a view has a schema, like all databases, but no proper data of its own.” If in the system there is only one database that has its own data, this database will be the conceptual schema.

The steps in order to define an external schema are as follows:

- Selection of classes from other schemas (external or conceptual) to be included in the actual external schema -they can be classes from more than one schema-; the selection is made using the *import* mechanism. When classes are imported, they become visible together with their subclasses.

- Hide the undesired classes or properties in classes of the schema (hiding a property in a class hides it in all its subclasses). If a class’s property is hidden in a schema, the rest of schemas where this class is included are not affected.

At this point an initial external schema is obtained. It can be further modified by the following operations:

- Definition of classes derived from classes already included in the external schema; the two basic mechanisms for defining derived classes are generalisation and specialisation. Derived classes are automatically integrated with their direct base classes using inheritance relationships.

- Hide additional properties or classes in the resulting external schema.

In this system, external schemas are called *views* and derived classes are *virtual classes*. If a derived class is defined with object-generating semantics it is called *imaginary class*, and its new objects are *imaginary objects*. No specific name is used in order to denote the conceptual schema or the data dictionary.

2.2.1.5 [Rundensteiner, 1992c]

Rundensteiner proposed an external schema definition methodology called *MultiView* [Rundensteiner, 1992a; Rundensteiner, 1992c]. It is complemented with algorithms presented in [Rundensteiner, 1992b] and [Rundensteiner & Bic, 1992]. In this methodology, the specification of an external schema is divided into the following independent tasks:

- Definition of derived classes needed from previously existing classes, derived or non-derived.

- Automatic integration of defined derived classes with all previously existing classes in an object schema; this integration process is carried out using the inheritance relationship [Rundensteiner, 1992b]. The objective is to explicitly maintain all the existing inheritance relationships between derived and non-derived classes. In this process, in order to have all the inheritance relationships explicitly defined, some additional derived classes may be automatically generated.

- Selection of a set of classes (derived and/or non-derived) from which will be composed the external schema.

- Automatic external schema generation. This task has to parts: Primarily, all classes that are used by originally selected classes have to be included in the external schema. These classes are automatically added to the set of selected classes [Rundensteiner, 1992a; Rundensteiner, 1992c]. Followed by the generation of a class hierarchy from the set of classes obtained [Rundensteiner & Bic, 1992]. The generation process consists of defining the inheritance relationships that exist between the classes in the set. The definition of the inheritance relationships is made taking into account only the inheritance relationships actually defined in the object schema in which all the classes have been integrated.

In this methodology, derived classes are called *virtual classes*, non-derived classes are *base classes*, external schemas are called *views*, *view schemas* or *virtual schemas*, and classes (derived or non-derived) included in an external schema are *view classes*. Two different schemas are defined: the *base schema*, which is the initial object schema where all classes correspond to non-derived classes, and the *global schema*, which is an extension of the base schema augmented by the collection of all derived classes defined during the lifetime of the database. Therefore, the conceptual schema would correspond to the global schema rather than to the base schema because the base schema has the limitation of not containing derived classes. The global schema is not the data dictionary either because it only contains the base schema and the new derived classes. However, it does not contain additional information about the definition of external schemas as the data dictionary should. The global schema partially plays the role of data dictionary.

2.2.1.6 [Tresch & Scholl, 1993]

The presentation of the external schema definition system of Tresch & Scholl [Tresch & Scholl, 1993] is made using the ANSI/SPARC terminology.

The steps in order to define an external schema are:

- Extend the conceptual schema by a set of derived classes that simulate the desired schema organisation; derived classes are defined by object-preserving semantics. Position the derived classes in the schema -related by inheritance only to its direct base classes [Scholl & Schek, 1991].

- Define a subschema of the extended schema, by selecting a set of classes (derived and non-derived) that corresponds to the restructured schema. Close the subschema adding the required classes.

The position of the derived classes in the schema is made automatically, but for any one of the other operations the possibility of it being done automatically is mentioned.

2.2.1.7 [Geppert et **al.***, 1993]*

Geppert, Scherrer & Dittrich [Geppert et *al*., 1993] proposed an architecture in which there are two levels of conceptual schemas: the level called *logical schema*, which describes the structure of a part of interest for a set of applications (in a database there can be many logical schemas); the other level is composed by the union of all the logical schemas to form the *global schema* (there is only one global schema). External schemas - or *subschemas*- are defined from logical schemas: an external schema is a subset of classes of a logical schema.

Focusing only on one logical schema, the steps in order to define an external schema are as follows:

- Definition and integration of the necessary derived classes into the conceptual schema (the logical schema considered). Only classes defined with object-preserving semantics can be defined.

- Selection of the set of classes that will compose the external schema. The external schema is required to be closed, but it is not explained how this is achieved.

2.2.1.8 [Barclay & Kennedy, 1993]

In the system proposed by Barclay & Kennedy [Barclay & Kennedy, 1993] each external schema is implemented defining a new class with just an object that simulates the behaviour of all classes of the new external schema; the operations of this class provide a site for the various queries to define how the external schema is derived from its base classes.

Consequently, the necessary steps in order to define an external schema are as follows:

- Define a new class to represent the external schema itself; this class can be defined by inheritance from other classes that also define external schemas. The extension and intension of the classes in the external schema are defined as operations of the new class.

- Define operations to represent the extent of the classes in the external schema.

- Define operations to represent all attributes and methods of classes in the external schema.

It is the responsibility of the application administrator to ensure that the resulting external schema is closed.

In [Barclay & Kennedy, 1993] the schema (conceptual schema or data dictionary) where the new defined classes are integrated is not explicitly defined, but it should be in the data dictionary. The ANSI/SPARC architecture is referred to and its terminology is used, but the data dictionary is not mentioned anywhere.

A significant disadvantage of this approach is that using a class that represents a schema means a change in the schema nature and in the way of dealing with it.

2.2.1.9 [Santos et **al.**, *1994]*

The external schema definition system proposed by Santos, Abiteboul & Delobel [Santos et *al*., 1994; Santos, 1995], continuation of the work of Abiteboul & Bonner [Abiteboul & Bonner, 1991], is based on the ANSI/SPARC three level architecture. However, they use their own terminology.

The main change of this proposal with regard to the system of Abiteboul & Bonner [Abiteboul & Bonner, 1991] is the way it integrates derived classes with the rest of classes in external schemas. In this case, a new kind of relationship is used: *may_be* relationship, using the concept of non-strict inheritance. Since external schemas are offered to end users, this means a modification of the object-orientation paradigm.

- Therefore, the first step of this methodology is to build the initial external schema that imports definitions from other schema or schemas, and hides unneeded properties or classes.

- Then, additional derived classes can be defined in the external schema; these classes are defined from the set of classes initially included in the external schema, as well as from previously defined derived classes (also included in the external schema). Derived classes with object-preserving semantics are defined through the generalisation or specialisation mechanisms; these classes are integrated in the external schema using the *may_be* relationship (instead of using inheritance with its direct base classes as was done by Abiteboul & Bonner).

- Finally, as in the system proposed by Abiteboul & Bonner, additional properties or classes in the resulting external schema can be hidden.

Here, external schemas are called *virtual schemas* or *views* and the term *real schema* is used in contrast to virtual schema to allude to the conceptual schema. As mentioned in [Santos, 1995], “Once defined, a virtual schema definition is compiled and stored into a view repository.” Therefore, this repository would be the data dictionary.

2.2.1.10 [Kim & Kelley, 1995]

Kim & Kelley [Kim & Kelley, 1995] refer to the ANSI/X3/SPARC three-level schema architecture, but the ANSI terminology is not totally used along their work. The term *view* is used with the meaning of derived class and also signifying external schema.

According to the concepts presented in [Kim & Kelley, 1995], the process of defining an external schema is as follows:

- Definition of the derived classes needed to form the external schema; derived classes are defined from previously existing classes (non-derived or derived). Derived classes are related with the classes from which they have been defined using a new relationship called *derived-from* relationship. The set of classes related by this relationship also form a *view-derivation hierarchy*. Derived and non-derived classes have separate inheritance hierarchies. A derived class is related by inheritance with the classes indicated at the moment of its definition, as is done for non-derived classes: “Information about superclasses and subclasses of a class (view) is defined within the class (view).” Therefore, “the burden of ensuring the correctness of the view hierarchy falls on the users.”

- Selection of the classes that will compose the external schema (this step is not explicitly defined). External schemas can be composed by derived and non-derived classes. Derived and non-derived classes have separate inheritance hierarchies, but they can be related by aggregation. The domain of a property of a derived class can be a derived or a non-derived class, but the domain of a property of a non-derived class can not be a derived class.

Thus, in [Kim & Kelley, 1995] the conceptual schema, called the *database schema*, “consists of two separate structures; one for the classes (i.e., the class hierarchy), and one for the views (derived classes).” Therefore, derived classes are integrated in the conceptual schema using a new relationship not defined in the object-orientation paradigm.

2.2.1.11 [Naja & Mouaddib, 1995]

In the external schema definition system defined by Naja & Mouaddib [Naja & Mouaddib, 1995], derived classes can only be defined by object-preserving semantics, and they have to contain the same set of objects as their base classes; also, derived classes can contain non-derived attributes.

In external schemas, derived classes are integrated using the inheritance relationship. Derived classes are related with its base classes using a derivation relationship, called the *is_derived_from* relationship; this relationship does not appear in external schemas, nor in the conceptual schema (it is not mentioned, but it should only appear in the data dictionary).

Therefore, external schemas are defined manually from other external schemas and/or the conceptual schema. External schemas can contain new derived classes which have the same extension as their respective base classes and, possibly, a new intension.

2.2.1.12 [Bertino et **al.,** *1996]*

Bertino, Catania, García-Molina & Gerrini [Bertino et *al.*, 1996] presented a system that allow "the definition of external schemas with the meaning proposed by the ANSI three-level architecture" and that can be used "to simulate schema evolution, allowing the users to experiment with schema changes without affecting other users."

This system is defined as an extension of Bertino's system [Bertino, 1992], where only the definition of derived classes is studied.

Using the terminology of ANSI/SPARC, the definition process of an external schema consists of the following steps:

- Take the conceptual schema or a previously defined external schema and define the necessary derived classes. All the classes to be included in a new external schema have to be new derived classes defined from only one schema. Derived classes are connected with the classes from which they have been defined using a new relationship called a *view derivation* relationship. The inheritance relationships between derived classes are defined at the moment of definition of the classes; the correctness of the inheritance relationships defined will be ensured by the system. This verification can be made because some limitations have been imposed upon the derived class definition language.

- The external schema has to be closed; the classes from the original schema referred to and not included in the external schema, are automatically redefined as new derived classes and included in the external schema.

In [Bertino et *al.*, 1996], the conceptual schema is called *base schema*; derived classes are called *views* and instead of external schema, the concept of *schema view* is used. It is an extension of the concept of external schema that allows the incorporation of derived classes with non-derived properties in order to support schema evolution.

As defined in [Bertino et *al.*, 1996]: "A *global database schema* consists of a base schema together with a set of schema views" and "the schema derivation and view derivation relationships are part of the global database schema too." Consequently, the concept of global database schema shall be the data dictionary.

2.2.2 A classification of the external schema definition methodologies

Three kinds of external schema definition methodologies are distinguished, just by considering the relation between the conceptual schema and the different external schemas defined.

2.2.2.1 External schemas are subschemas of the conceptual schema

In this group of methodologies [Rundensteiner, 1992c; Geppert et *al.*, 1993; Tresch & Scholl, 1993; Kim & Kelley, 1995] the conceptual schema has to contain all the classes of the defined external schemas. If an external schema containing a class that is not previously included in the conceptual schema is to be defined, this class has to be defined and integrated into the conceptual schema. Class integration assures the consistency of all external schemas with the conceptual schema and with one another. The main problem here is that the conceptual schema becomes more complex each time a new derived class is defined. The conceptual schema is used in part with the function of the data dictionary, in the sense that all the classes included in external schemas have to be previously integrated into it.

In some cases [Rundensteiner, 1992c; Geppert et *al.*, 1993; Tresch & Scholl, 1993], new derived classes are integrated by inheritance into the conceptual schema; however, in the system proposed by Kim & Kelley [Kim & Kelley, 1995] derived classes are integrated using a new derivation relationship with its base classes and by inheritance with other derived classes.

The most representative methodology of this group is Rundensteiner's one. Some examples developed according to this methodology are presented in chapter 5 in order to show the problems that it has. Finally, a solution to them is proposed.

2.2.2.2 External schemas are not necessarily subschemas of the conceptual schema

External schemas can contain classes not included in the conceptual schema [Tanaka et *al.*, 1988; Heiler & Zdonik, 1988; Dayal, 1989; Abiteboul & Bonner, 1991; Santos et *al.*, 1994; Naja & Mouaddib, 1995; Bertino et *al.*, 1996]. These new classes are derived directly or indirectly from conceptual schema classes, but they are not necessarily included into the conceptual schema. Therefore, the conceptual schema is not affected by external schema definitions.

The main problem of most existing methodologies of this second group is that external schemas are defined independently. Except for the system proposed by Bertino, Catania, García-Molina & Gerrini [Bertino et *al.*, 1996], an external schema definition data dictionary that allows all the defined classes integrated, does not exist, so it is difficult to re-use previous definitions: i.e., in the systems presented in [Abiteboul & Bonner, 1991], and in [Santos et *al.*, 1994; Santos, 1995], derived classes included in an external schema can only be defined from classes already included in this external schema; even in the system presented in [Bertino et *al.*, 1996], an external schema can be defined from only one schema (external schema or conceptual schema), isolated from the rest of definitions already made.

2.2.2.3 External schema as a class of the conceptual schema

This is a unique case [Barclay & Kennedy, 1993], where each external schema is implemented defining a new class which only contains one object that simulates the behaviour of all the classes of the new external schema. Compared with the conceptual schema, the way of defining external schemas means a change in the nature of these schemas and also in the way of working with them.

The classes that conceptually compose the external schema are not classes previously existing in the conceptual schema, therefore, from this point of view this methodology might have been classified in the second group. The main reason for making a new group for this methodology is to make evident its peculiarity.

2.3 Issues in the definition of derived classes

Derived classes are classes which are defined from previously existing classes (derived or non-derived) using object-oriented queries. *Non-derived classes* are defined during the initial definition of the conceptual schema (it can contain non-derived and also derived classes). Derived classes are defined during the lifetime of the database in order to be included in some external schema (or in the conceptual schema). Normally, a derived class can be used like a (non-derived) class.

2.3.1 Integration of derived classes in a schema

Integration of derived classes refers to two different scopes: integration of derived classes and previously existing classes in the data dictionary (or in the conceptual schema playing the role of data dictionary) and integration of a set of classes (derived and/or non-derived) to form an external schema.

In external schemas defined according the object-orientation paradigm, integration has to be done using the inheritance relationship; in some cases this is not respected.

On the other hand, the main objective of integrating new derived classes with the rest of the existing classes in an object schema is twofold: primarily, it is to maintain explicit class relationships between derived and non-derived classes in order to have external schemas consistently defined; secondly, as pointed out in [Rundensteiner, 1992b], class integration serves data modelling purposes, therefore, “classes should be organised in a systematic manner such that they are more easily comprehensible by the users of the system.” With this aim in mind, data dictionaries may allow other forms of organisation. Consequently, besides the inheritance relationship, other kind of relationships -maybe more suitable than inheritance- may be used in the integration of derived classes.

In order to show the different alternatives of integration of a derived class in a class hierarchy (external schema or data dictionary), the example object schema in fig. 2.3 is going to be used. A new class EMPLOYEES', defined from class EMPLOYEES hiding the Salary property and selecting objects that are not manager employees, is defined. If this new class is to be integrated into the class hierarchy, it can be seen that it is not a subclass of the original class EMPLOYEES because it has less properties (Salary); nor it is a superclass of EMPLOYEES because its set of objects is a subset of the set of objects of EMPLOYEES (only not manager employees). In the following parts of this section, some of the solutions to this problem offered in different approaches are presented.

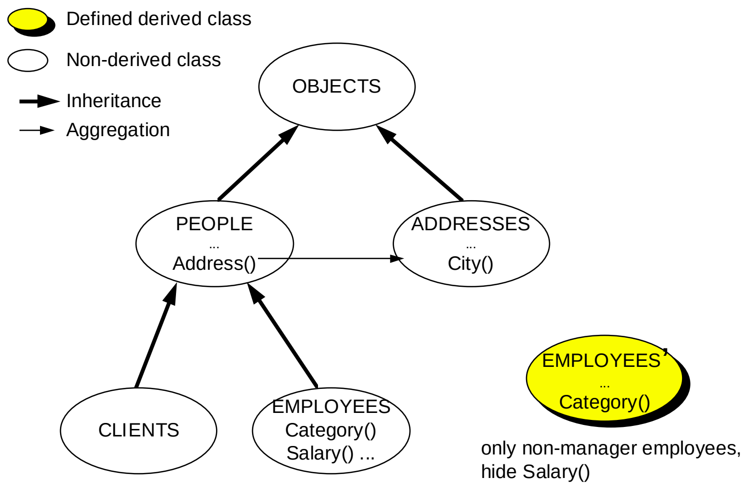


Figure 2.3. OODB example schema and definition of a derived class.

2.3.1.1 Integration using the inheritance relationship

As defined in [Rundensteiner, 1992c]: “Class integration is concerned with finding the most ‘appropriate’ location in the schema graph for a virtual class in terms of property inheritance and subset relationships between classes.”

2.3.1.1.1 Direct subclasses of class *objects*

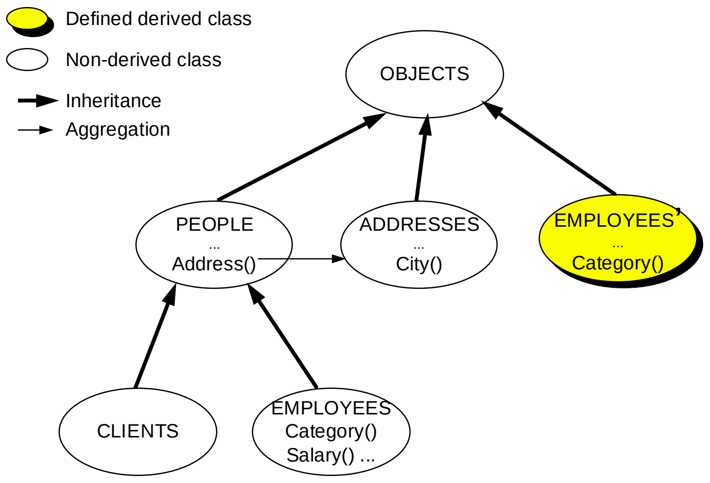


Figure 2.4. Derived class direct subclass of **objects**.

One possibility is to define the new derived classes as direct subclasses of class **objects**, as shown in fig. 2.4. This approach completely ignores the issue of classification, thus resulting in a flat class structure that does not take advantage of the mentioned possibilities offered by inheritance, neither of the modelling possibilities of the object-orientation paradigm.

This solution is adopted in Kim's system [Kim, 1989]. With regard to our example, the solution proposed in [Kim, 1989] would have defined class EMPLOYEES' with object-generating semantics -since it does not allow another definition of derived classes.

2.3.1.1.2 Relation only with its base classes

In this solution, each derived class is only related by inheritance with its direct source classes. The main problem, as the case of the example shows, is when a derived class is not directly related by inheritance to its immediate source classes. This problem is avoided allowing only definition operations of derived classes in which the resulting class can be directly related with its source classes. Therefore, it would have best results in a partial, hence less informative, class hierarchy; as pointed out in [Rundensteiner, 1992b]: "There may be additional *subsumption relationships* between the derived class and other classes in the schema that are not directly derivable from the class derivation. It is the task of class integration to find these class relationships and to explicitly represent then in the schema graph."

In the external schema definition methodology presented in [Abiteboul & Bonner, 1991], in order to define an external schema, derived classes can be defined from classes previously included in it; the inheritance relationships between a new classes and its base classes are automatically obtained according to the operations used in the definition of the derived class (generalisation or specialisation).

This solution is also adopted in [Scholl & Schek, 1991; Tresch & Scholl, 1993]: for each derived class definition operation, the relationship between the classes that participate is defined.

2.3.1.1.3 Explicitly defined relations

In this approach the user (the application administrator) is required to specify explicitly the inheritance relationships between the defined derived classes and existing classes. This approach is vulnerable to potential consistency problems, since the users might introduce inconsistencies in the schema graph by inserting "is-a" arcs between two classes not related by an inheritance relationship; also, an incomplete schema graph that does not capture all existing class relationships may be defined. A solution to verifying the correctness, in essence would have to be able to provide automatic verification of the class hierarchy defined.

In our example, see below fig. 2.5, derived class EMPLOYEES' is defined only as a direct subclass of class PEOPLE; as has been previously shown, it can not be directly related with class EMPLOYEES using the inheritance relationship.

In [Bertino et *al*., 1996] the definition of the inheritance relationship between the classes selected to compose an external schema is carried out in this way. The correctness of the inheritance relationships defined are ensured by the system.

Inheritance relationships between derived classes in [Kim & Kelley, 1995] are also defined explicitly, but no verification mechanism is mentioned there, so “the burden of ensuring the correctness of the view hierarchy falls on the users.” This is also the case in [Tanaka et *al*. 1988; Dayal, 1989; Naja & Mouaddib, 1995].

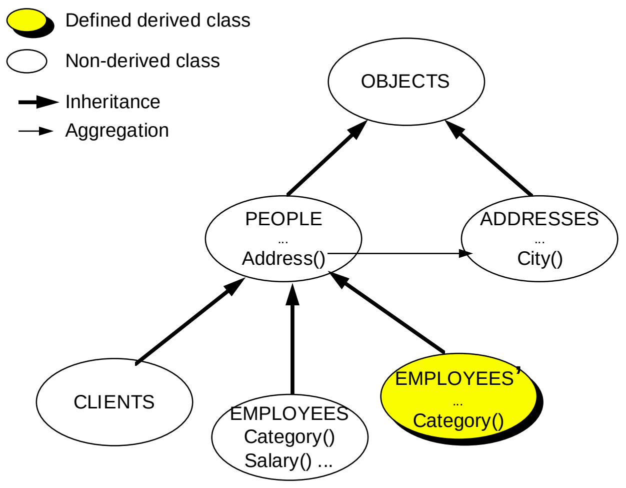


Figure 2.5. Integration of a derived class with existing classes using inheritance.

2.3.1.1.4 Define all the possible relations

Instead of integrating manually the derived classes and verifying automatically that integration has been properly done, an alternative is to carry out the integration process automatically; thus, all explicit inheritance relationships existing between the derived class and the rest of existing classes are obtained.

In the example considered, the result obtained would be fig. 2.5. The resulting object schema may still seem incomplete: classes EMPLOYEES and EMPLOYEES' have objects and properties in common, that are more specific than the ones of class PEOPLE, and this fact is not explicitly represented in the class hierarchy.

In Rundensteiner’s methodology [Rundensteiner, 1992c] a derived class integration process is proposed that solves this question. In order to integrate derived classes by inheritance into an object schema, the object schema requires that for each pair of classes of it that have some property in common, a superclass of them which only has all the properties common to both classes has to be also included in the object schema (this property is called *inheritance closure* of the object schema). The resulting schema of the example is presented in fig. 2.6. In order to integrate the class EMPLOYEES', all the classes required to maintain the object schema according to the enunciated requirement are automatically generated (class EMPLOYEES").

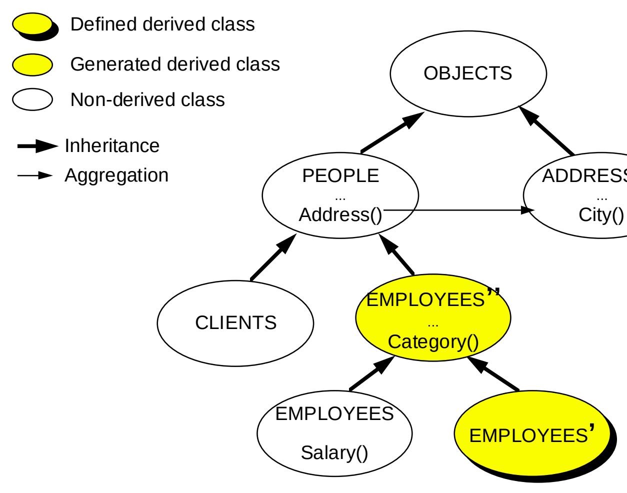


Figure 2.6. Rundensteiner's automatic class integration by inheritance (I).

In order to show this process with more detail, in fig. 2.7 another example of integration is presented. In this case, class EMPLOYEES' is defined hiding the property Salary and adding a new property City which returns the city where the employee lives, already defined in class ADDRESSES. In this case, the inheritance problem between EMPLOYEES and EMPLOYEES' is just in their respective types. A new class EMPLOYEES" is generated which contains a set of objects different of the same class in the previous example. Another additional class has to be generated, the class WITH_CITY. This new class contains the properties common to ADDRESSES and EMPLOYEES'; thus, these properties can be inherited by both classes.

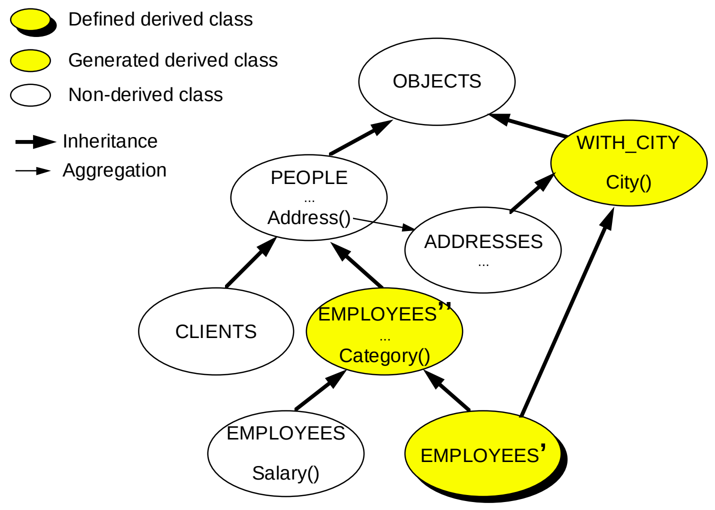


Figure 2.7. Rundensteiner's automatic class integration by inheritance (II).

Therefore, the essence of this solution is the creation of additional intermediate classes that restructure the schema graph.

2.3.1.2 Integration using other relationships

This approach ignores the issue of determining inheritance relationships between derived classes and other classes by using other kind of relationships.

Contrary to the integration of derived classes in a class hierarchy using the inheritance relationship, in [Bertino et *al*., 1996] is argued that: “This approach has the major problem that gives rise to inheritance hierarchies quite complex, often containing classes that are not semantically meaningful for the users.” Then, an alternative solution to the problem of integration of derived classes in an object schema is to use another kind of relationships.

2.3.1.2.1 Derivation relationship

Each derived class is related with the classes from which it has been defined using a derivation relationship, it defines the way in which derived classes are obtained independently if they have been defined using object-preserving or object-generating semantics. In our example, fig. 2.8, class EMPLOYEES' is directly integrated using the derivation relationship.

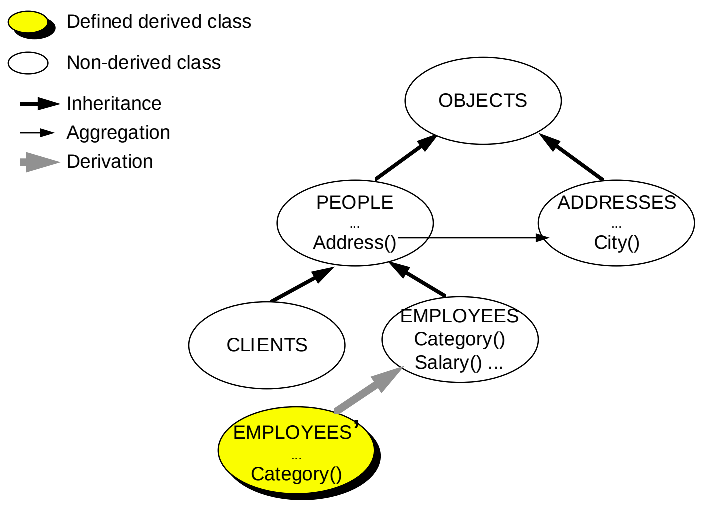


Figure 2.8. Integration of derived classes using the derivation relationship.

This relationship is called *view derivation* in [Bertino, 1992; Bertino et *al*., 1996]. In [Bertino et *al*., 1996] the definition of external schemas is studied, and the derivation relationship only appears in the data dictionary. In [Bertino, 1992] only the definition of derived classes is studied and the object-orientation paradigm is extended by introducing the derivation relationship along side the aggregation and inheritance relationships.

In [Kim & Kelley, 1995] derived classes are related with the classes from which they have been defined by the *derived-from* relationship; the set of classes related using this relationship form a *view-derivation hierarchy*.

In the case of the derivation relationship proposed in [Naja & Mouaddib, 1995], the *is_derived_from* relation, restrictions are stronger than in similar relationships. The extensions of two classes related using this relation have to be the same, the only difference between both classes can be its respective set of attributes. However, this relationship is only used in the data dictionary.

In [Monk, 1994] derived classes are related with its base classes by the *view-of* relationship; derived classes can be defined only by object-preserving semantics.

2.3.1.2.2 *May_be* relationship

The *May_be* relationship is proposed in [Santos et *al.*, 1994] in order to integrate derived classes in external schemas (this means an extension to the object-orientation paradigm). Derived classes defined with object-preserving semantics are related with its base classes using the *may_be* relationship. This relationship is called *may_be* to distinguish it from the conventional inheritance relationship usually called *is_a*. In that sense, an instance of a base class *may_be* an instance of a corresponding derived class (it is defined only with object-preserving semantics).

In relation to the group derivation relationships, a different group has been distinguished here because the meaning of this relationship is different. In the system proposed in [Santos et *al.*, 1994] derived classes defined by object-generating semantics can also be defined and they are not related with any relationship to its base classes. The *may_be* relationship is only defined if the derived class is defined by object-preserving semantics; therefore, it is clearly different to the derivation relationship defined in other works.

In our example, derived class EMPLOYEES' is defined by object-preserving semantics, therefore the result obtained will be the same as the one presented in fig. 2.7 for the derivation relationship.

2.3.1.2.3 *Cluster* of classes

In order to solve the problem of integration of derived classes with other classes, the solution proposed in [Heuer & Sander, 1991] consists of the definition of *clusters* of classes: “A cluster consist of at most one base class and several derived classes having the same set of ‘possible objects’ (the same abstract domain) as the corresponding base class.” If a derived classes is defined by object-preserving semantics, it has its own cluster. Inside each cluster, two different hierarchies are considered: one of types, and another of instances. Each derived class is classified in the hierarchies that belong to its respective cluster.

In the example considered, the derived class EMPLOYEES' is included into the cluster of its base class EMPLOYEES from which it is defined.

2.3.2 Subsumption between classes

As pointed out in [Rundensteiner, 1992b], taken from [Schmolze & Lipkis, 1983]: “*Classification* is the process of taking a new (class) description and putting it where it belongs in the (class) hierarchy.” In the process of automatically integrating a derived class into an object schema, or in the automatic verification of the result obtained from a manual process of integration, a process of classification is carried out. As it is expressed in [Rundensteiner, 1992b]: “A class is in the ‘right place’ (in a class hierarchy) if it is below all classes that subsume it and if it is above all classes that it subsumes.” Therefore, a method for determining the subsumption relationships between classes is needed in both cases, as mentioned in [Rundensteiner, 1992b]: “We thus need to define a boolean function *subsumes()* that given two classes, c_1 and c_2, determine whether the first subsumes the second.”

Class c_1 is said to subsume class c_2, denoted *subsumes*(c_1, c_2), if and only if c_1 can be defined as a superclass of c_2 in a class hierarchy correctly defined. This means that the type associated to c_1 is a supertype of the type of c_2; and, the set of objects of c_1 always contains the set of objects of c_2.

Derived and non-derived classes can have membership constraints associated to them. As defined in [Rundensteiner, 1992b]: “Membership constraints are predicates that restrict the set content of a class, i.e., this could be a subset-predicate for base (non-derived) classes or a derivation query for virtual (derived) classes.” Therefore, “the classification problem for object-oriented models is not decidable since it may involve the comparison of arbitrary functions and predicates.” In order to avoid this situation, “(...) one would either have to limit the expressiveness of the derivation specification such as to be computable, or, we could require a canonical predicate expression that can be broken into decidable expressions. In the later case, we would base the classification on the comparison of this partial information.”

In the development of the classification algorithm presented in [Rundensteiner, 1992b], a *subsumes()* function has been defined -not presented there- with some of the limitations previously indicated, and the results obtained are as follows: “Our classification algorithm is *sound* but *not complete*. The *subsumes()* function being *sound* means that if the function returns true for a pair of classes then the two classes are necessarily is-a related. (...) Second, the subsumes function is *total*, i.e., it always terminates and returns either true, or fail. However, the subsumes function is *not complete*, i.e., the function is not guaranteed to discover a relationship between two classes even if one exists. (...) In the worst case, if some is-a relationship is not discovered, then the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would be a correct but not the most informative class arrangement.”

In [Bertino et *al*., 1996] a subsumption function is also needed in order to verify the correction of external schemas defined manually. The solution proposed there consists of limiting the possibilities in the definition of derived classes, and also limiting the inheritance relationships defined between derived classes in external schemas.

Subsumption is also studied in [Buchheit et *al*., 1994], being the query optimisation problem in a class hierarchy the main target in this case.

The conclusion to this situation is also enunciated in [Rundensteiner, 1992b]: “Hence, the development of a realistic *subsumes()* function for some of the emerging object models needs to be investigated. The goal of such a project would be not to restrict the expressive power of the model nor the constructs used for deriving new classes, while guaranteeing that the *subsumes()* function stays computable.” This topic is not further studied here. We suppose that a *subsumes()* function exists. The study and definition of such a function can be the topic of another thesis.

2.3.3 Object-preserving and object-generating semantics

A derived class is defined with object-preserving semantics if it only contains objects extracted from previously existing classes. A derived class is defined with object-generating semantics if it contains new objects generated in the definition process, the new objects must be identified by newly generated object identifiers.

If the derived class represents a concept previously defined in object form, it will have to be defined by object-preserving semantics: the derived class defines a new interface for its objects. If the derived class represents a concept not previously defined in object form, it will have to be defined by object-generating semantics.

In the papers about the definition of derived classes three tendencies can be distinguished: those that only allow derived classes to be defined by object-preserving semantics, those that only allow definitions to be carried out by object-generating semantics, and those that afford both definition semantics.

2.3.3.1 Only object-preserving semantics

In the case of the systems presented in [Tanaka et *al.*, 1988; Heiler & Zdonik, 1988; Scholl & Schek, 1991; Rundensteiner, 1992c; Monk, 1994; Geppert et *al.*, 1993; Kim & Kelley, 1995; Naja & Mouaddib, 1995], only derived classes defined by object-preserving semantics can be defined.

One of the main reasons for defining this limitation is that, with object-preserving semantics updates can be handled better between objects in derived classes and objects in their corresponding base classes [Scholl & Schek, 1991; Rundensteiner, 1992c; Barclay & Kennedy, 1993; Geppert et *al.*, 1993].

In [Tanaka et *al.*, 1988] the definition of derived classes by object-generating semantics is proposed as a further research topic.

2.3.3.2 Only object-generating semantics

In [Kim, 1989] and [Kifer et *al.*, 1992] only object-generating semantics can be used in the definition of derived classes. Even if the derived class represents a concept previously defined in object form, a new object is generated. Hence, information will be replicated in these systems.

In [Kifer et *al.*, 1992] the main topic studied is the generation of identifiers for objects in derived classes; identifiers are generated using functions specific to each derived class. The only requirement that these functions have to fulfil is that they return a unique value for each different set of input parameters, and that this value does not occur elsewhere in the database. In order to be able to manage the transmission of modifications between objects in derived classes and the respective objects in base classes, the correspondence between its identifiers is stored.

2.3.3.3 Both object-preserving and object-generating semantics

The possibility of defining derived classes by object-preserving and object-generating semantics allows one to carry out sophisticated reorganisations of the existing information which would not be possible if derived classes could only be defined by object-preserving semantics, i.e., “combining small objects into larger aggregate objects; decomposing large objects into several smaller objects; sophisticated restructuring that turns objects into values and values into objects. [Abiteboul & Bonner, 1991]”

Other systems that allow the definition of derived classes with both semantics are the following: [Dayal, 1989; Shaw & Zdonik, 1990; Abiteboul & Bonner, 1991; Heuer & Sander, 1991; Heuer & Scholl, 1991; Hull et *al.*, 1991; Bertino, 1992; Alhajj & Arkun, 1993; Santos et *al.*, 1994; Santos, 1995; Bertino et *al.*, 1996].

Usually, the correspondence among the new object identifier and the base object identifier is stored in order to manage the transmission of modifications between them.

2.3.4 Identifiers of the objects in derived classes

Each object (non-derived or derived) is represented by its identifier. In [Hull et *al.*, 1991] it is stated that: “An object identifier has no intrinsic meaning -and derives its meaning *only* from its relationship to values or other object identifiers in a given database instance. In particular, then, if an object identifier is considered independently from its associated database instance, then it conveys essentially no information other than its identity as being distinct from all other object identifiers.”

In some proposals the identifier of a derived object is defined exclusively in function of other object identifiers; in other cases the identifier is defined in function of other object identifiers as well as function of values of some of the properties of the objects; these two options are presented next.

2.3.4.1 Function of identifiers of the base objects

The definition of derived classes by object-preserving semantics can be considered a particular case in which the object identifiers of objects in derived classes only depend on object identifiers of objects in other classes. As a matter of fact, the object identifiers of objects in derived classes are the object identifiers of the objects from which they have been defined.

In other cases, new object identifiers are generated for the objects in derived classes defined with object-generating semantics. In [Bertino, 1992; Kifer et *al.*, 1992; Bertino et *al.*, 1996; Gardarin & Yoon, 1996] the new object identifiers can be defined as a function of the identifiers of the objects from which they are defined, i.e. aggregation of objects in order to form a new concept. A derived class can be defined from many classes; the classes on which the identity of the objects in the derived class depend are explicitly defined using some specific clause.

In [Bertino, 1992] by means of the clause ‘identityfrom’, in [Bertino et *al*., 1996] by means of ‘UPDATE-ON’, and in [Kifer et *al*., 1992] by means of ‘OID FUNCTION OF’ the subset of base classes that the identity of a derived class instance depends upon is specified; the object identifiers of the objects belonging to the classes that participate in the definition of an object, together with the generated object identifier, usually are stored together in order to maintain this correspondence; such a correspondence also can be used to propagate the update operations to the base objects.

In the case of [Dayal, 1989], the identifiers of derived objects are considered as tuples whose components are the identifiers of their corresponding base objects. If a derived class only has a base class, its objects will be defined by object-preserving semantics, base and derived objects will have the same object identifier.

2.3.4.2 Function of values or identifiers of the base objects

If the identifier of a derived object is defined exclusively in function of other object identifiers, then, the transformation of values into objects cannot be defined. In order to offer this kind of transformations, some systems [Abiteboul & Bonner, 1991; Heuer & Sander, 1991; Hull et *al*., 1991; Santos et *al*., 1994] allow the definition of new objects as functions of values of properties, or a combination of values and object identifiers of other objects.

It can be considered that the object identifier of a derived object is generated from a set of its attributes; in [Abiteboul & Bonner, 1991] these attributes are called *core attributes*. As mentioned in [Santos et *al*., 1994]: “The first time an imaginary object (derived object defined with object-generating semantics) is accessed, the attributes from which the identity of the object depends must be given so that the object can be properly constructed. Further accesses return the same object identifier.”

An additional problem in this case is the management of changes in the values of the core attributes. As pointed out in [Santos et *al*., 1994]: “The values of the core attributes may change but their object identifiers must remain the same.”

2.3.5 Transmission of modifications

In order to transmit the modifications from the objects in the derived classes to the objects in the classes upon which those derived classes are defined, a connection must exist between them. If the definition of the derived classes is carried out exclusively with object-preserving semantics, this connection is immediate as it is given directly by the object identifier according to [Scholl & Schek, 1991; Rundensteiner, 1992c; Kim & Kelley, 1995]. If a derived class is defined by object-generating semantics, a connection has to be maintained between the identifier of the derived object and the objects from which it is defined [Kifer et *al*., 1992; Bertino et *al*., 1996].

As defined in [Heiler & Zdonik, 1988], in the transmission of modifications from objects of derived classes to the corresponding objects in base classes, the *equivalence preservation property* has to be fulfilled: correct changes in the base objects have to be produced in order to provide the desired updates in derived objects.

Regarding the problem of modification transmission, the solutions which are put forward by other authors are presented: automatic transmission limiting the definition of the derived classes and transmission of modifications through methods of the derived classes.

2.3.5.1 Automatic transmission of modifications

According to Gottlob, Paolini & Zicari [Gottlob et *al*., 1988]: “Most of the authors who have been studying the view-update problem concentrate their attention on finding ways for deriving translations automatically or semi-automatically by restricting the set of allowed (static) view definitions and the set of allowed update policies. Their derivation rules usually are based upon notions of ‘natural translation’ (typically minimalty of side-effects) and upon constraints on the data model and on the database instances (functional dependencies and other data dependencies for relational databases).” Such an affirmation -made several years ago when most of the papers on this subject focused on relational databases- still applies to a large number of subsequent papers on the transmission of modifications in OODBs.

The main restriction imposed upon the definition of the derived classes in order for their instances to be modifiable, is that it be carried out by object-preserving semantics [Scholl & Schek, 1991; Rundensteiner, 1992c; Kim & Kelley, 1995]. Thus, the number of possible cases in the modification transmission is reduced. The implementation is simplified considerably in view of the fact that, having the same identifier, no additional structure is needed to relate the base objects and the derived objects -the derivation of identity alone being taken into consideration. The modification operators which are applied to the objects of the derived class bring about the same effect as if they had been applied directly to the objects of the corresponding base class [Ra & Rundensteiner, 1995; Kim & Kelley, 1995].

Regarding the restrictions in the set of possible modification policies, those in which the transmission is not direct are not permitted, i.e., the direct assignment to derived properties [Scholl & Schek, 1991; Kim & Kelley, 1995] and the creation of new objects in derived classes [Abiteboul & Bonner, 1991; Santos et *al*., 1994]. Among all the possible forms of transmission of a modification to a derived object one is selected, as mentioned in [Gottlob et *al*., 1988], normally the one which brings about the least side-effects or the one which may be considered the ‘most natural’, i.e., create, delete, add, remove and set update operators applied to an instance of a virtual difference class work on the first argument class [Ra & Rundensteiner, 1995]; deletion of an object related by aggregation with other objects, in [Kim & Kelley, 1995] only the root object is deleted, not the component objects. Nevertheless, this form of transmission is not necessarily always the most appropriate.

The automatic definition of modification policies keeps the effort of defining derived classes to a minimum -the definition of a derived class consisting merely in defining the manner in which objects are obtained. This requires major restrictions, both in derived class definition as well in permitted modification operations. If derived classes are defined by object-generating semantics, the automatic definition of modifications would become more complicated because the number of ways of possible transmissions increases, this being the reason why it is not permitted.

2.3.5.2 Transmission of modifications through methods of the derived class

An alternative to the automatic transmission of modifications is to use the additional mechanisms that are offered by the object-orientation paradigm -in particular, that of encapsulation: defining the manner of propagation of all the operations by means of methods [Dayal, 1989; Kimura & Tsuruoka, 1991; Rundensteiner, 1992c; Kifer et *al.*, 1992; Andersen & Reenskaug, 1993; Barclay & Kennedy, 1993]. In [Bertino et *al.*, 1996] the use of the automatic transmission mechanism is proposed when the derived class is defined by object-preserving semantics and there are no problems of ambiguity. If ambiguity in the manner of transmission exists (i.e., deletions on derived classes defined as joins), it is suggested that methods which implement it are used. Normally, the methods of the derived class are defined using methods of the base classes.

The cost of definition involved in this solution is greater than that of automatic modification transmission since the definition of the derived class consists in the definition of the manner in which its objects are obtained and also in the implementation of the methods of modification transmission.

2.3.6 Definition of non-derived attributes

In some systems [Bertino, 1992; Naja & Mouaddib, 1995; Bertino et *al.*, 1996], the definition of derived classes with non-derived attributes is allowed. The definition of those classes permits one to simulate some additional transformations in a schema evolution environment.

As pointed out in [Kim & Kelley, 1995]: “A non-derived attribute defined in a view (derived class) obviously has no corresponding attribute in a stored class. Therefore, values inserted into the attribute cannot be stored in any ‘corresponding’ stored class, and also the attribute cannot be materialised;” and no alternative solution to this situation is given there.

According to the ANSI/SPARC architecture, external schemas can only contain information derived from the conceptual schema. Therefore, if an external schema is to be defined including a class with some non-derived attribute, the conceptual schema should be modified in order to include the additional information [Ra & Rundensteiner, 1995]; therefore, the information in external schemas can always be derived from the conceptual schema.

2.4 Conclusions

The main target of this chapter has been studying the elements that take part in the definition of external schemas and derived classes in OODBs.

The ANSI/SPARC framework defined a general architecture for DBMSs. This architecture has been widely applied for relational databases but this has not been the case for OODBs. The external schema is a part of this architecture concept, and its definition is studied in the proposed framework; consequently, the main characteristics of this framework have been presented in section 2.1.

In section 2.2 a review and a classification of existing external schema definition methodologies has been made. Most of the methodologies studied do not consider explicitly the ANSI/SPARC framework, but propose architectures that approximately can be translated to this one. An effort has been made in this sense, and these methodologies have been presented using the ANSI/SPARC terminology. Even the few methodologies considered that reference the ANSI/SPARC architecture propose systems that do not totally meet the definitions of this architecture.

Derived classes can be included in external schemas. In some outstanding points about the definition of derived classes, different alternatives have been proposed, some of the most important ones have been presented in section 2.3, specifically, the integration of derived classes with other classes in an object schema, the possibility of defining derived classes with object-preserving and object-generating semantics, the problems in the generation of identifiers for new objects, and the transmission of modifications between the objects in base classes and derived classes.

We consider that the problems presented are not solved satisfactorily, and alternative proposals are presented in chapters 5, 6 and 7 of this work.

3 Object-oriented concepts

The concepts used in this thesis don't refer to any particular object oriented model, they are general concepts applicable to most existing object models. In this chapter, the formal definition of the basic object oriented concepts used in our proposal is presented, as well as some additional outstanding concepts used by other authors.

3.1 Formal definition of our reference basic OODB model

The concepts presented in this section have been defined starting from the object models presented in [Rundensteiner, 1992c] and [Abiteboul et *al*., 1995]. Many of its features are shared by most of the existing OODB models. The terminology used in [Abiteboul et *al*., 1995] with some changes and extensions has been used in the formal definition.

3.1.1 Constants, values and objects

The atomic types **integer**, **string**, **bool**, **float**, and their corresponding domains are considered. The set **dom** of atomic values is the union of these domains; the elements of **dom** are called *constants*. A special constant *nil* represents the undefined (i.e., null) value.

The set **obj** = $\{o_1, o_2, ...\}$ is the infinite set of *object identifiers* (OIDs); and **att** is the set of *attribute names*. Given a set O of OIDs, i.e., $O \subset$ **obj**, then the family of *values* over O, denoted **val**(O), is defined so that

a) the constant $nil \in$ **val**(O);
b) if $v \in$ **dom** then $v \in$ **val**(O);
c) if $o \in O$ then $o \in$ **val**(O);
d) the set $\{v_i \mid v_i \in$ **val**(O), $(i = 1, ..., n)\} \in$ **val**(O); and
e) given a set of values $\{v_i \mid v_i \in$ **val**(O), $(i = 1, ..., n)\}$, and a set of distinct attribute names $\{A_i \mid A_i, A_j \in$ **att**, $(i \neq j \Rightarrow A_i \neq A_j), (i, j = 1, ..., n)\}$ then, the tuple $[A_1 : v_1, ..., A_n : v_n] \in$ **val**(O).

An *object* is a pair (o, v), where o is an OID and v a tuple value.

3.1.2 Types. The aggregation relationship

Objects are grouped in classes; **class** is the set of class names. All objects in a class have values of the same type, these values are tuples. Each class c is associated with a type $\sigma(c)$, which dictates the type of the objects in this class. In particular, for each object (o, v) in class c, v must have the structure described by $\sigma(c)$.

So, types are defined with respect to a given set C of class names, $C \subset$ **class**. The family of *types* over C, denoted **types**(C), is defined so that

1. the atomic types **integer**, **string**, **bool**, and **float** $\in$ **types**(C);
2. if $c \in C$ then $c \in$ **types**(C);
3. if $\tau \in$ **types**(C), then the set $\{\tau\} \in$ **types**(C);
4. given a set of types $\{\tau_i \mid \tau_i \in$ **types**(C), $(i = 1, ..., n)\}$, and a set of distinct attribute names $\{A_i \mid A_i, A_j \in$ **att**, $(i \neq j \Rightarrow A_i \neq A_j)$, $(i, j = 1, ..., n)\}$, then the tuple $[A_1 : \tau_1, ..., A_n : \tau_n] \in$ **types**(C); and
5. given the special class name **objects**, σ(**objects**) = **any**, then **any** $\in$ **types**(C), but this type may not occur inside another type.

Each class c has a type with tuple form associated, i.e., $\sigma(c) = [A_1 : \tau_1, ..., A_n : \tau_n]$. If a class c_1 has an attribute A_i of type τ_i, and $\tau_i = c_2$, being c_2 a class, then between the classes c_1 and c_2 there is an *aggregation relationship* -the objects of class c_1 are composed by objects of class c_2.

3.1.3 Class hierarchy. The inheritance relationship

A *class hierarchy* is a triple $(C, \sigma, \prec)$, where C is a finite set of classes, σ is a mapping from C to **types**(C), and $\prec$ is a strict partial order relation (transitive and irreflexive) on C, called *subclass relation*, corresponding to a specification of the *inheritance relationships* between the classes in the class hierarchy. The class **objects** is included in every class hierarchy, being σ(**objects**) = **any**; and, for each class c in any class hierarchy, $c \prec$ **objects**.

Given a finite set C of class names, the *subtyping relationship* on **types**(C) is the smallest partial order $\leq$ over **types**(C) satisfying the following conditions:

a) if $c_1 \prec c_2$, then $c_1 \leq c_2$;
b) if $\tau_i \leq \tau'_i$, $(i = 1, ..., n)$, and $n \leq m$, then $[A_1 : \tau_1, ..., A_n : \tau_n, ..., A_m : \tau_m] \leq [A_1 : \tau'_1, ..., A_n : \tau'_n]$;
c) if $\tau \leq \tau'$, then $\{\tau\} \leq \{\tau'\}$; and
d) for each τ, $\tau \leq$ **any**.

For two types $\tau_1, \tau_2 \in$ **types**(C), τ_2 is called a *subtype* of τ_1, if and only if $\tau_2 \leq \tau_1$; in this case, τ_1 is called a *supertype* of τ_2. The type τ_2 is a *direct subtype* of τ_1 and τ_1 is a *direct supertype* of τ_2, if $\tau_2 \leq \tau_1$, and $((\nexists\ \tau_i \in$ **types**(C)$)\ (\tau_i \leq \tau_1, \tau_2 \leq \tau_i, \tau_i \neq \tau_1, \tau_i \neq \tau_2))$.

In a class hierarchy the type associated with a subclass should be a refinement (the same type or a subtype) of the type associated with its superclasses. A class hierarchy $(C, \sigma, \prec)$ is *well formed with regard to* **types**(C) if for each pair $c_1, c_2 \in C$, $c_1 \prec c_2$ implies $\sigma(c_1) \leq \sigma(c_2)$.

Given a well-formed class hierarchy $(C, \sigma, \prec)$, for two classes $c_1, c_2 \in C$, c_2 is called a *subclass* of c_1, if and only if $c_2 \prec c_1$; in this case, c_1 is called a *superclass* of c_2. The class c_2 is a *direct subclass* of c_1 and c_1 is a *direct superclass* of c_2, if $c_2 \prec c_1$, and $((\nexists\ c_i \in C)\ (c_i \prec c_1, c_2 \prec c_i))$.

3.1.4 Methods

A *method* has three components: a *name*, a *signature*, and an *implementation* (or *body*). The set **meth** is an infinite set of method names. Let $(C, \sigma, \prec)$ be a class hierarchy. For method name *m*, a *signature* of *m* is an expression of the form $m : c \times \tau_1 \times \ldots \tau_{n-1} \rightarrow \tau_n$, where *c* is a class name in *C* and each τ_i is a type over *C*. This signature is associated with the class *c*; a method *m applies* to objects of class *c*.

The same method name can have different signatures in connection with different classes; but, if $m : c_1 \times \tau_1 \times \ldots \tau_{n-1} \rightarrow \tau_n$ and $m : c_2 \times \tau'_1 \times \ldots \tau'_{p-1} \rightarrow \tau'_p$ are two definitions of *m* and $c_1 \prec c_2$, then, $n = p$, and $\tau_i \leq \tau'_i$ ($i = 1, \ldots, n$). This rule is called *covariance* of the definition of *m* in c_1 and c_2.

If a method *m* is defined for a class c_1 but not for a direct subclass c_2 of c_1, then, the definition of *m* for c_2 is *inherited* from c_1. The signature of *m* in c_2 has the form $m : c_2 \times \tau_1 \times \ldots \tau_{n-1} \rightarrow \tau_n$; the signature of *m* on c_2 is identical to the one of *m* on c_1, except that the first c_1 is replaced by c_2. In general, the method *m* is inherited by a class c_2 from a class c_1 where *m* has been defined, if $c_2 \prec c_1$, *m* is not defined in c_2, and *m* is not defined in any class c_i, being $c_2 \prec c_i$, $c_i \prec c_1$. If *m* has been inherited by c_2, the implementation of *m* for c_2 is identical to that for c_1.

A set *M* of method signatures is associated to a class hierarchy $(C, \sigma, \prec)$. Each class *c* of *C* has a set of method signatures associated, denoted $\mu(c)$: the set of methods defined in *c* or inherited by *c* from its superclasses; according to this, $M = \cup\{\mu(c) \mid c \in C\}$. Objects in a class *c* can be accessed only using the set of methods $\mu(c)$. The methods of a class *c* are defined using the attributes of $\sigma(c)$; methods are dependent on the set of attributes used in their definition. Methods allow us to consult or modify the value of the attributes of the objects in class *c*. The set of methods applicable to an object is called the *interface* of the object. Objects are accessed only via their interface; this principle is called *encapsulation*.

In order to avoid the production of ambiguities in the inheritance of methods, the *unambiguity* rule is defined: if c_1 is a subclass of c_2 and c_3, and there is a definition of *m* for c_2 and c_3, then there is a definition of *m* for a subclass of c_2 and c_3 that is either c_1 itself, or a superclass of c_1.

The set *M* of method signatures associated to the class hierarchy $(C, \sigma, \prec)$ is *well formed* if it obeys the unambiguity and the covariance rules.

Extending the definition previously given, a class hierarchy $(C, \sigma, \prec)$ is *well formed* if it is well formed with regard to **types**(*C*), and the set *M* of method signatures associated to the class hierarchy is also well formed. Only well-formed class hierarchies are considered in this work.

3.1.5 Attributes and methods: properties

Each class c in a class hierarchy $(C, \sigma, \prec)$ has a set of attribute names associated corresponding to the ones in $\sigma(c)$, and a set of methods $\mu(c)$. The attributes of c constitute its internal structure; the methods of $\mu(c)$ constitute the interface of c.

An attribute of name a and type τ defined in $\sigma(c)$ can be expressed in the same form as a method: $a : c \rightarrow \tau$. According to the definition of well-formed class hierarchy, if $c_1, c_2 \in C$, $c_2 \prec c_1$, and a is an attribute of type τ_1 defined in $\sigma(c_1)$, i.e., $a : c_1 \rightarrow \tau_1$, then the attribute a is also defined in $\sigma(c_2)$, i.e., $a : c_2 \rightarrow \tau_2$, and $\tau_2 \leq \tau_1$. (The covariance rule for methods was defined in a similar way.)

The set of *properties* or the *intension* of a class c, denoted $\rho(c)$, is defined as the union of the set of attributes defined in $\sigma(c)$ and the set of methods $\mu(c)$. In a class hierarchy $(C, \sigma, \prec)$, if $c_1, c_2 \in C$, and $c_2 \prec c_1$, then $\rho(c_1) \subseteq \rho(c_2)$, and $((\forall\, p \in \rho(c_1))$ (if $p : c_1 \times \tau_1 \times \ldots \tau_{n-1} \rightarrow \tau_n$ and $p : c_2 \times \tau'_1 \times \ldots \tau'_{m-1} \rightarrow \tau'_m$ then $n = m$, and $\tau'_i \leq \tau_i$ $(i = 1, \ldots, n)$)), covariance rule for properties.

As in [Rundensteiner, 1992c], for simplicity, we assume for the following that all properties in a class hierarchy have unique property names. To ensure uniqueness of properties, an unique property identifier can be associated to each newly defined property; therefore, two properties that have the same property name could thus be distinguished internally based on their identifier.

Each property p in a class c may be defined using other properties of the class c or from other classes in the class hierarchy. The set of properties that property p needs for its definition in class c is denoted by $def(p,\mathrm{c}) = \{(p_i,c_j) \mid$ property $p \in \rho(c)$ uses the property $p_i \in \rho(c_j)$ in its definition$\}$.

3.1.6 The structural semantics of a class hierarchy

Let $(C, \sigma, \prec)$ be a class hierarchy. An *OID assignment* is a function π mapping each name in C to a finite set of OIDs. Given OID assignment π, the *instances* or the *proper extension* of $c \in C$ is $\pi(c)$. An object o instance of a class $c_1 \in C$, denoted $o \in \pi(c_1)$, can not be instance of any of the subclasses of c_1 in the class hierarchy: $((\nexists\; c_i \in C)\; (o \in \pi(c_i), c_i \prec c_1))$. A major point of difference of the model presented here with respect to the model of [Abiteboul et *al.*, 1995] is that an object can be instance of two classes not related by inheritance in a class hierarchy, like in [Rundensteiner, 1992c] or [Bertino et *al.*, 1995].

The *members* or the *extension* of c, denoted $\pi^*(c)$, is $\cup\{\pi(c_i) \mid c_i \in C, c_i \prec c\}$; if an object o is an instance of a class c, then o is also a member of all the superclasses of c. If π is an OID assignment, then $\pi^*(c_2) \subseteq \pi^*(c_1)$ whenever $c_2 \prec c_1$.

The semantics for types is now defined relative to a class hierarchy $(C, \sigma, \prec)$ and an OID assignment π. Let $O = \cup\{\pi(c) \mid c \in C\}$, and define $\pi^*(\textbf{objects}) = O$. The *interpretation* of a type τ, denoted $dom(\tau)$, is given by

a) for each atomic type τ, $dom(\tau)$ is the usual interpretation of that type;
b) $dom(\textbf{any})$ is $\textbf{val}(O)$ (being $\sigma(\textbf{objects}) = \textbf{any}$);
c) for each $c \in C$, $dom(c) = \pi^*(c) \cup \{nil\}$;
d) $dom(\{\tau\}) = \{\{v_1, ..., v_n\} \mid n \geq 0$, and $v_i \in dom(\tau)$, $(i = 1, ..., n)\}$; and
e) $dom([A_1 : \tau_1, ..., A_k : \tau_k]) = \{[A_1 : v_1, ..., A_k : v_k , A_{k+1} : v_{k+1}, ..., A_l : v_l] \mid v_i \in dom(\tau_i), (i = 1, ..., k), v_j \in \textbf{val}(O), (j = k + 1, ..., l)\}$.

3.2 Class hierarchy closure

The definitions presented in this section are based on [Rundensteiner, 1992c]. Closure of a class hierarchy referes to the closure of the inheritance relationship, and also to the closure of the references made between classes through properties.

3.2.1 Inheritance closure

Let $(C, \sigma, \prec)$ be a class hierarchy. Given $c_1, c_2 \in C$, a class c_3 is a *common superclass* of c_1 and c_2 if the properties of c_3 are $\rho(c_3) \subseteq (\rho(c_1) \cap \rho(c_2))$, and the set of objects of c_3 are $\pi^*(c_3) \supseteq (\pi^*(c_1) \cup \pi^*(c_2))$; if the definition of a property p for c_3 is $p : c_3 \times \tau''_1 \times ... \tau''_{n-1} \rightarrow \tau''_n$, then, the definition of p for c_1 and c_2 is, respectively, $p : c_1 \times \tau_1 \times ... \tau_{n-1} \rightarrow \tau_n$, and $p : c_2 \times \tau'_1 \times ... \tau'_{n-1} \rightarrow \tau'_n$, where each $\tau''_i = \tau_i$ if $\tau'_i \leq \tau_i$, otherwise $\tau''_i = \tau'_i$ (for $i = 1, ..., n$). If c_1 and c_2 do not have any property in common, their only common superclass is the class **objects**.

The class c_3 is the *lowest common superclass* of c_1 and c_2, denoted $LCS(c_1, c_2) = c_3$, if and only if c_3 is a common superclass of c_1 and c_2, but $\rho(c_3) = (\rho(c_1) \cap \rho(c_2))$, and $\pi^*(c_3) = (\pi^*(c_1) \cup \pi^*(c_2))$.

Given $c_1, c_2 \in C$, then $\sqcap$ is a function from $C \times C \rightarrow C$ that defines a new class c_3 by $c_3 = c_1 \sqcap c_2$, being c_3 a common superclass of c_1 and c_2, and $\rho(c_3) = (\rho(c_1) \cap \rho(c_2))$.

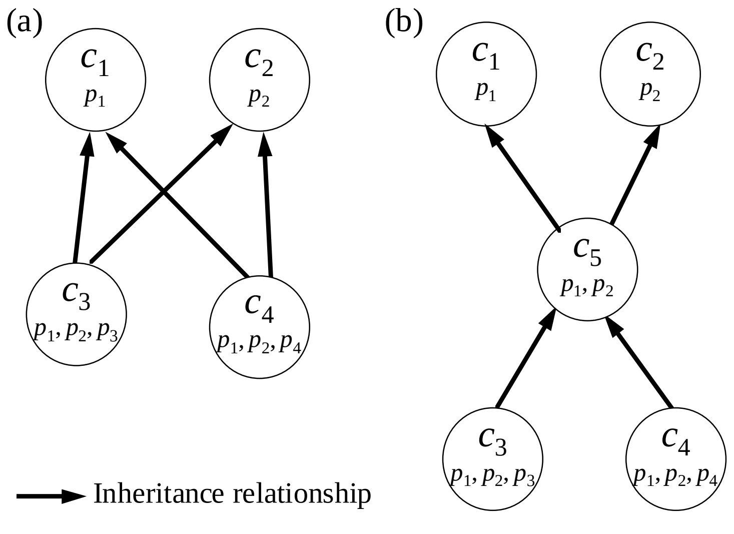


Figure 3.1. Class hierarchy closure under function $\sqcap$.

A class hierarchy (C, σ, $\prec$) is *closed under* $\sqcap$ if and only if for any two classes c_1, $c_2 \in C$, a class c_3 is included in C, being $c_3 = c_1 \sqcap c_2$ (c_3 is a superclass of c_1 and c_2 which only has all the properties common to both classes).

In fig. 3.1.a, part of a class hierarchy that is not closed under $\sqcap$ can be seen; classes c_3 and c_4 inherit the properties defined in c_1 and c_2. In order to obtain a class hierarchy closed under $\sqcap$ class c_5 has to be defined (fig. 3.1.b), class c_5 contains all the properties common to both c_3 and c_4.

In a class hierarchy closed under $\sqcap$ each property is defined only in one class, if used elsewhere, it is inherited from this original definition class; properties can be redefined in accordance with the rules that define a well formed class hierarchy.

A way of integrating derived classes in the class hierarchy is to make them direct subclasses of class **objects** [Kim, 1989]; the main reasoning against this method of derived class integration is that some information that may be interesting to the end-user is lost, because in the class hierarchy all the inheritance relationships existing between the classes are not explicitly expressed. The main aim of having a class hierarchy closed under function $\sqcap$ is to have explicitly represented all the inheritance relationships existing between all the classes in the class hierarchy: all the properties common to any pair of classes are explicitly defined in (or inherited by) another class which only has these properties, and it is superclass of both classes in the class hierarchy.

In the model proposed by Rundensteiner [Rundensteiner, 1992c] class hierarchies have to be closed under $\sqcap$ in order to integrate automatically new defined classes. We do not have this requirement in our model, it is only an additional possibility as it will be shown in chapter 5.

3.2.2 Property decomposition closure

Let (C, σ, $\prec$) be a class hierarchy; a class $c_1 \in C$ is directly related with another class $c_2 \in C$ via a *property relationship*, denoted $pr(c_1,c_2)$, if and only if (($\exists\, p \in \rho(c_1)$) ($p : c_1 \times \tau_1 \times \ldots \tau_{n-1} \rightarrow \tau_n$ and $c_2 = \tau_i$, for some $n \geq i \geq 1$. In general, a class c_1, is related with another class c_2 via a property relationship, denoted $pr^*(c_1,c_2)$, if and only if, $pr(c_1,c_2)$ or, $pr(c_1,c_i)$ and $pr^*(c_i,c_2)$ for some $c_i \in C$; that is to say, c_1 is related with c_2 via a property relationship if they are directly or indirectly related via a property relationship.

The aggregation relationship is only a particular case of the property relationship; the aggregation relationship referes to the relationship expressed in attributes, the property relationship referes to the aggregation relationship and the relationships expressed in methods, so the aggregation relationship is included in the definion of the property relationship.

The set of all property relationships among the classes in a class hierarchy is refered as its *property decomposition hierarchy*.

A class hierarchy (C, σ, $\prec$) is *closed in relation to the property relationship* or its *property decomposition hierarchy is closed*, if and only if, for all $c_i \in C$, if c_i is related with some other class c_j via a property relationship, then $c_j \in C$.

According to the definition of class hierarchy (secction 3.1.3), all the class hierarchies have to be closed in relation to the property relationship; the reason is that in a class hierarchy (C, σ, $\prec$), σ is defined as a mapping from C to **types**(C) -the family of types over C- thus, all the classes included in **types**(C) are also included in C.

3.3 Object schemas

3.3.1 Valid object schema

A class hierarchy is frequently represented as a graph. An *object schema* is a representation of a class hierarchy (C, σ, $\prec$) in the form of a rooted directed acyclic graph $S = (C, E)$, where C is the set of classes and E is a finite set of directed edges. Each edge $e = <c_1, c_2>$, being $c_1, c_2 \in C$, represents the fact that the class c_1 is a direct subclass of class c_2.

As in [Rundensteiner, 1992c], given an object schema $S = (C, E)$, an edge $e = <c_1, c_2>$ is defined to be

- *required* in S, if c_1 is a direct subclass of c_2;
- *redundant* in S, if c_1 is an indirect subclass of c_2;
- or *inconsistent* in S, if c_1 is not a subclass of c_2.

An object schema $S = (C, E)$ is *valid* if the set of edges E contains all required and no redundant, neither inconsistent edges in S; that is, the set of edges E represents all the existing pairs of direct subclasses in C, and only contains these edges.

3.3.2 Closed object schema

By definition, an object schema contains the class **objects**, and also has to be closed in relation to the property decomposition relationship, because it is a representation of a class hierarchy.

Given a set of classes C, and the set E of all required, non-redundant and non-inconsistent edges that can be defined between the classes of C, if $S = (C, E)$ is not an object schema (because it does not represent a class hierarchy closed in relation to the property decomposition relationship -and inheritance relationship if required), then, $S' = (C', E')$ is a minimal object schema defined from C, if and only if, S' is a valid object schema, $C \subseteq C'$, and $((\nexists\ c \in C')\ (c \notin C,\ C'' = C' - \{c\}$, and $S'' = (C'', E'')$ is an object schema for some set of edges E'')) -that is to say that the set C' contains the classes originally included in C, and some additional classes which are strictly necessary in order to obtain a closed object schema.

Given a set of classes *C*, many minimal object schemas can be obtained from it. With regard to the property decomposition hierarchy closure, there is no doubt about the classes which have to be included in *C'* in order to obtain a closed schema: all the classes referenced by some property of any class of *C* or of any referenced class. For obtaining a class hierarchy closed under function $\sqcap$ (see section 3.2.1), different sets of classes can be considered: the function $\sqcap$ is defined such that, if $c_3 = c_1 \sqcap c_2$ then, there may be different possibilites for defining the extension of class c_3 in order to have c_3 as a common superclass of c_1 and c_2 -the only condition about the extension is $\pi^*(c_3) \supseteq (\pi^*(c_1) \cup \pi^*(c_2))$.

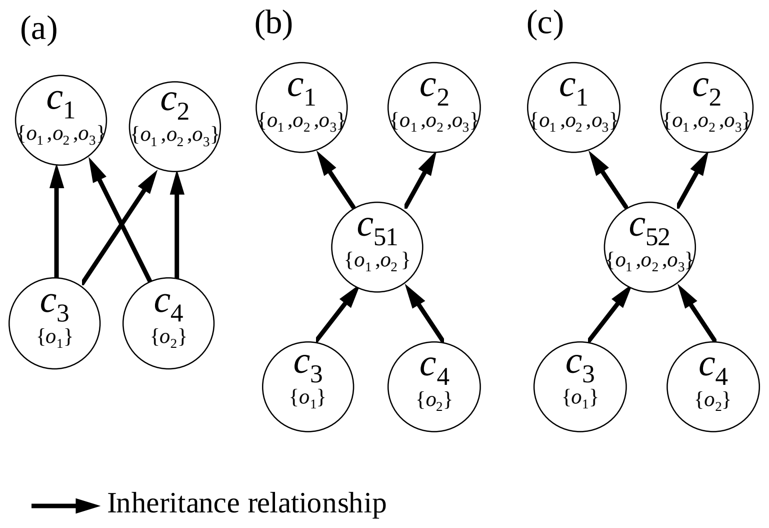


Figure 3.2. Different possibilities obtaning a class hierarchy closed under function $\sqcap$.

In the example of fig. 3.1, class c_5 was defined in order to have a class hierarchy closed under function $\sqcap$. Concerning the extension of the new class c_5, in fig. 3.2 two different possibilities are shown, adding class c_{51} or class c_{52}, and in both cases a minimal object schema is obtained.

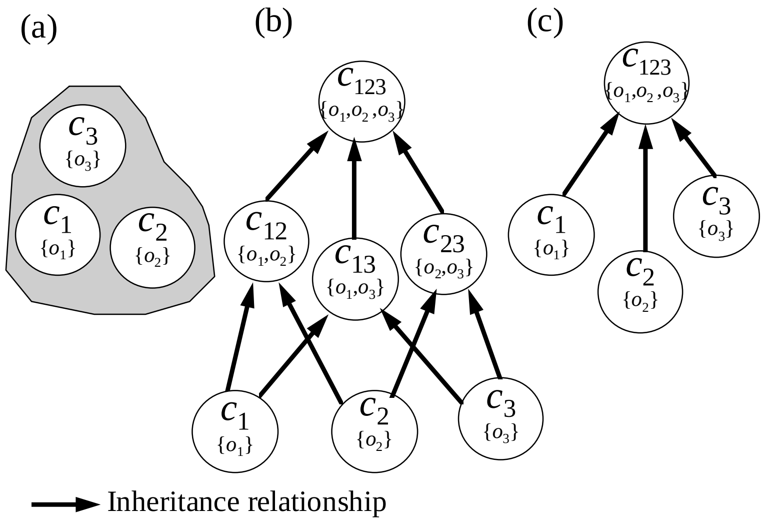


Figure 3.3. Minimal object schema.

In fig. 3.3 a new example is presented; in fig. 3.3.a, classes c_1, c_2 and c_3, all of them with the same intension ($\rho(c_1) = \rho(c_2) = \rho(c_3)$), are selected to compose an object schema. In figs. 3.3.b and 3.3.c two different class hierarchies closed under function $\sqcap$ obtained from the set of classes of fig. 3.3.a are represented. All the classes generated in both figures also have the same intension. In fig 3.3.b, classes c_{12}, c_{13}, c_{23} can be removed and the class hierarchy remains closed under function $\sqcap$. The object schema represented in fig. 3.3.c is minimal, because the class added can not be removed without afecting the closure of the schema.

Given any two minimal object schemas $S' = (C', E')$ and $S'' = (C'', E'')$ defined from a set of classes C, the cardinality of C' and C'' is the same and $((\forall\ c \in (C' - C), \exists\ c' \in (C'' - C))\ (\rho(c) = \rho(c')))$ -both schemas have the same number of classes, and the classes added to the initial set of classes C in order to obtain a minimal object schema have the same set of properties for any minimal object schema:

- With regard to the property decomposition hierarchy closure, in order to obtain a closed schema, the set of classes that have to be added are all the classes referenced by some property of any class in the initial set of classes C, or of any one of the referenced classes; so, there is no alternative in obtaining this set of classes.

- Once a set of classes closed with regard to the property relationship has been obtained, in order to obtain from it an object schema closed under function $\sqcap$, for each pair of classes c_1, c_2 of this new set, a class $c_3 = c_1 \sqcap c_2$ has to be included in the resulting set of classes. By definition of function $\sqcap$, each one of the classes c_3 will only have properties already defined in c_1 and c_2. So, the intension of the classes to be added is clearly defined; on the other hand, as has been shown in fig. 3.3, if there are many classes with the same intension in an object schema closed under function $\sqcap$, in order to obtain a minimal object schema, all of them except the highest class in the class hierarchy with the same intension can be removed and the object schema still remains closed. The only degree of freedom that remains in the definition of those classes is in the definition of their extension, as has been shown in fig. 3.2.

4 Definition of DCMs of OODBs

In this chapter the definition of deductive conceptual models (DCMs) using Prolog in order to specify different aspects of OODBs is proposed. The result of the specification process using this technique is an executable prototype of the system. Having a prototype directly available, along with the system specifications, is particularly useful in order to define additional elements in the context of OODBs (e.g. schema evolution, definition of derived classes, definition of external schemas). The use of this technique is proposed mainly due to the difficulty of building prototypes of the mentioned elements over commercial OODBs. The specification of a conceptual schema definition system and its associated data model is presented.

4.1 Introduction

In the field of OODBs a variety of proposals over a broad range of aspects are found: from the actual data model to be used, to proposals about the evolution of the schema, the definition of derived classes, the definition of external schemas, etc. Their implementation, or even prototype implementation, is carried out in very few cases.

The carrying out of prototypes serves as a great help in the specification of any kind of complex system, and the mentioned elements of OODBs are no exception. The problem lies in the fact that it is not always easy to carry out a prototype of the system to be specified -not on account of the system itself, but because there is not an adequate platform or tool available for building it. In the actual case of the OODBs, one possibility is to use a commercial OODB as the platform for the construction of the prototype (depending on the aspect to be prototyped and the possibilities that the database offers, this could be a good solution). The main problem in using a commercial OODB lies in the fact that the database itself could be a limitation -e.g. its data model or the possibility of its' being extended.

The definition of DCMs is a technique which allows the specification of information systems (ISs) by expressing only their logical component [Olivé, 1989]. Moreover, if Prolog is used for the construction of the DCM [Costal et *al.*, 1989], together with the formal specification, a prototype of the system is also obtained. On account of this, in this chapter the development of DCMs in Prolog is proposed as a means of specifying the different aspects of interest of OODBs. As far as we know, this is the first proposal in this sense.

In section 4.2 the features and basic elements of DCMs are briefly outlined. In section 4.3 a general architecture for developing DCMs of the different aspects of OODBs is proposed. In order to show the practical application of DCMs, part of the specification of a conceptual schema definition system is carried out along section 4.4. Finally, in sections 4.5 and 4.6, some comparisons with other works and conclusions are presented.

4.2 Deductive conceptual models

The basic feature of conceptual models is that they make it possible to specify the logic of ISs. There are two types of conceptual models: operational and deductive. Operational conceptual models, as well as specifying the logic of ISs, define part of their control component. However, DCMs specify ISs expressing only their logical component [Olivé, 1989].

A detailed description of DCM construction can be found in [Olivé, 1989], and the use of Prolog language as a means to this end in [Costal et *al.*, 1989]. In the remainder of this section we summarise the most important points of this process.

One of the main elements to be considered in order to construct a DCM is time; all the external events relevant to the system, together with information about the moment in which they occurred, are recorded in the form of *base predicates*. Based on the base predicates, *derived predicates* are defined; they represent the information about the system in any given moment. The *output requirements* and the desired behaviour of the system are also expressed using derivation rules.

The system's internal status must be always consistent; such consistency is defined through a set of *integrity constraints* based on the base and derived predicates. If, as a result of some external event or in the course of time, the system is found to be no longer fulfilling any of the integrity constraints, consequent action must be taken so that these rules are fulfilled overall. If it is due to a recorded event, the immediate solution is to cancel the record of the said event. If an inconsistency comes about due to the passing of time there is no general solution; the most suitable course of action will vary according to each case.

Thus, the components of a DCM are:
- Base predicates.
- Derived predicates.
- Integrity constraints.
- Output requirements.

In section 4.4 these components will be seen in greater detail for the model we are concerned with.

4.3 Architecture of DCMs of OODBs

We propose to model each aspect of the OODBs separately, using the output of one model as the input for the rest of the models as may be necessary.

The first element to be modelled corresponds to the definition of the conceptual schema - marked in grey in fig. 4.1. The input for the conceptual schema definition system is a set of definitions of types, classes, etc.; as its output, a representation of the OODB's conceptual schema is obtained, which in turn can be the input for another system -as showed in fig. 4.1.

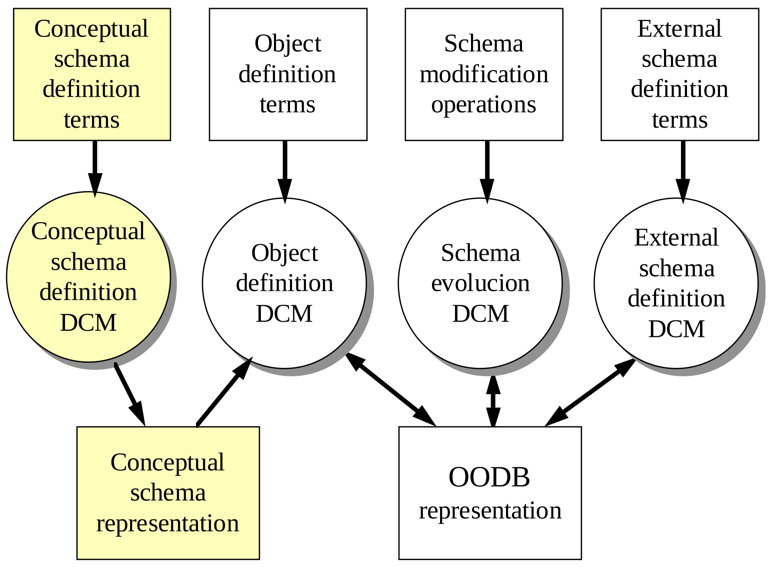

Conceptual schema definition terms
Object definition terms
Schema modification operations
External schema definition terms
Conceptual schema definition DCM
Object definition DCM
Schema evolucion DCM
External schema definition DCM
Conceptual schema representation
OODB representation

Figure 4.1. Proposed DCM definition architecture.

Our initial purpose is to specify an external schema definition system. With this aim, we have build a conceptual schema definition DCM, an object definition DCM and the external schema definition DCM -some derived predicates of the last one can be found in chapter 5.

4.4 Conceptual schema definition DCM

Let's take as our example of an OODB conceptual schema the one represented in fig. 4.2; a possible syntax to express it can be seen in fig. 4.3.

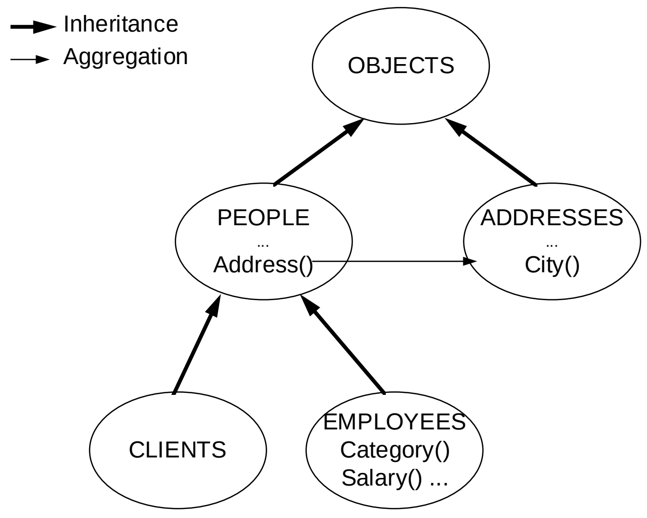


Figure 4.2. Representation of the OODB example schema.

```
type person is_a any
      name (string)
      address (addresses)
      ...
type employee is_a person
      category (string)
      ...
type address is_a any
      city (string)
      ...
class people (person) is_a objects
class clients (person) is_a people
class employees (employee) is_a people
class addresses (address) is_a objects
```

Figure 4.3. Definition of the OODB example schema.

The conceptual schema definition terms showed in fig. 4.3 correspond to the system input. In order to deal only with the semantic aspects of the system, let's suppose that we start out with definition terms which are syntactically correct.

Below we describe in detail the different components of the DCM that has been developed. The features of the used object-oriented model will be defined through the development of the corresponding DCM. For the development the PDC Prolog (before Turbo Prolog) language has been used.

4.4.1 Base predicates

In fig. 4.4 the base predicates are represented. They are recorded in response to the definition events.

```
domains
      id          = integer
      idList      = id*
      time        = integer
      domain      = t(id); c(id); t_(id); c_(id)
      signature   = domain*
      propertyName= symbol
      typeName    = symbol
      className   = symbol

database
      csDefineProperty(id,propertyName,time)
      csDefineType(id,typeName,idList,time)
      csDefineTypeProperty(id,id,signature,time)
      csDefineClass(id,className,id,idList,time)
      csDefineEnd
```

Figure 4.4. Base predicates.

The events and their corresponding base predicates are the following:

- Definition of property (`csDefineProperty`): the name of a property is defined and an internal identifier is assigned to it -by which reference can be made from the rest of the predicates. Additionally, the moment of definition is recorded (in all of the following predicates also, this being the case we shall not mention this aspect any further).

- Definition of type (`csDefineType`): the internal type identifier, the name, and a list of type identifiers corresponding to the supertypes of the type in question are recorded.

- Properties of a type (`csDefineTypeProperty`): for each property of a type, the type internal identifier, the identifier of the property and the property signature are recorded. The property signature is composed by a list of terms which may be types or classes -represented by `t` or `c` respectively, depending on whether the element is a value or an object when working with instances. Also they may be made up of sets of values or objects all of the same type or class -represented by `t_` or `c_` respectively. Thus, associated with the symbols `t`, `c`, `t_` and `c_`, the identifier of the corresponding type or class is given.

- Definition of a class (`csDefineClass`): the internal identifier of the class, the class name, the identifier of the associated type, and the list of identifiers corresponding to the superclasses of the class are recorded.

- End of the definition (`csDefineEnd`): the definition of the conceptual schema has finished.

The system disposes of some predefined types and classes (the types `any`, `real`, `integer`, `char` and `string`; and the class `objects`). For these predefined types and classes, the definition system automatically generates the corresponding definition events (recorded in the form of base predicates as can be seen in fig. 4.5, together with de base predicates corresponding to the schema in fig. 4.2).

```
csDefineType(Tany,any,[],t0)
csDefineType(Tstring,string,[Tany],t0)
...
csDefineClass(Cobjects,objects,Tany,[],t0)
...
csDefineProperty(P1,name,t1)
csDefineProperty(P2,address,t2)
csDefineProperty(P3,categoy,t3)
csDefineProperty(P4,city,t4)
...
csDefineType(T1,person,[Tany],t5)
csDefineType(T2,employee,[T1],t6)
csDefineType(T3,address,[Tany],t7)
...
csDefineClass(C1,people,T1,[Cobjects],t8)
csDefineClass(C2,clients,T1,[C1],t9)
csDefineClass(C3,employees,T2,[C1],t10)
csDefineClass(C4,addresses,T3,[Cobjects],t11)
...
csDefineTypeProperty(T1,P1,[t(Tstring)],t12)
csDefineTypeProperty(T1,P2,[c(C4)],t13)
csDefineTypeProperty(T2,P3,[t(Tstring)],t14)
csDefineTypeProperty(T3,P4,[t(Tstring)],t15)
```

Figure 4.5. Base predicates instances.

4.4.2 Derived predicates

From the base predicates, a set of derived predicates is defined. They represent information about the status of the modelled system at any given moment. Derived predicates are defined in the form of rules of deduction -there may be one or more rules of deduction for each derived predicate. Each rule of deduction is defined using base predicates or other derived predicates (including the predicate that is being defined by the rules: recursivity).

(a)

```
superclass(IdC,IdC2,T) :-
      directSuperclass(IdC,IdC2,T),
      !.
superclass(IdC,IdC2,T) :-
      indirectSuperclass(IdC,IdC2,T),
      !.
```

(b)

```
directSuperclass(IdC1,IdC2,T) :-
      classSuperclasses(IdC2,Superclasses,T),
      memberId(IdC1,Superclasses),
      !.

indirectSuperclass(IdC1,IdC2,T) :-
      classSuperclasses(IdC2,Superclasses,T),
      superclassOfSomeClass(IdC1,Superclasses,T),
      !.

superclassOfSomeClass(IdC1,[IdC2|_],T) :-
      superclass(IdC1,IdC2,T),
      !.
superclassOfSomeClass(IdC,[_|L],T) :-
      superclassOfSomeClass(IdC,L,T).
```

(c)

```
classSuperclasses(IdC,IdCs,T) :-
      csDefineClass(IdC,_,_,IdCs,T1),
      T1 <= T.
```

Figure 4.6. Examples of derived predicates.

In fig. 4.6 some examples of derived predicates are presented. In fig. 4.6.a the definition of the `superclass` derived predicate, which defines the fact that a class may be direct or indirect superclass of another, is shown. This predicate is defined using the predicates of fig. 4.6.b. In the case of a direct superclass, the identifier of the first class will be in the list of superclasses of the second class; the first class will be an indirect superclass if it is a superclass of one of the superclasses of the second class.

In the derived predicates presented so far, the only direct reference to a base predicate has been made through the `classSuperclasses` derived predicate, whose definition can be found in fig. 4.6.c. The base predicates have been encapsulated through derived predicates; thus the derived predicate definitions are isolated from possible changes in the base predicates.

4.4.3 Integrity constraints

The semantics of the DCM is defined by the integrity constraints; they are expressed using derived predicates.

When a new definition event is produced it is recorded by the `assertz` standard language predicate; also, the rules of integrity are verified by the defined `inconsistency` predicate. If some inconsistency is produced, the record of the event must be cancelled; this operation is done by the `retract` standard language predicate. This procedure, for the case of the class definition predicate, can be seen in fig. 4.7. Similar predicates are defined for each of the specified events.

```
classDefinition(IdC,Class,IdT,IdCs) :-
      now(T),
      assertz(csDefineClass(IdC,Class,IdT,IdCs,T)),
      inconsistency(T),
      retract(csDefineClass(IdC,Class,IdT,IdCs,T)),
      assertz(error),
      !.
classDefinition(_,_,_,_) :-
      !.
```

Figure 4.7. Events and inconsistency validation.

Below some of the rules of deduction corresponding to the `inconsistency` derived predicate are presented.

- A property may only be defined once in a type. Fig. 4.8.a corresponds to the rule of deduction that defines this inconsistency. By means of the `typePropertyAtT` predicate (becomes true if a property has been defined in a type at the indicated moment) the fact that a property has not been defined twice in the same type is validated.

- A type cannot be a supertype of itself. In order to verify that loops have not come about in the definitions of inheritance relationship between types the `supertype` derived predicate has been used, similar to the `superclass` defined beforehand. In this way, if, when defining a new type, it turns out that a type is a supertype of itself, the definition would not be valid, as can be seen in the rule of deduction in fig. 4.8.b.

- There cannot be redundant supertypes in a type definition. In the supertype list of a type there must be no redundant types -repeated types or supertypes of types already included in the list, fig. 4.8.c. This fact can be verified by using the `redundantSupertypes` derived predicate, presented in fig. 4.8.d.

- Single definition of properties between types. A property can only be defined once. If two types share properties, these have to have been inherited from the same type. The rule of deduction that defines this inconsistency, fig. 4.8.e, consists in searching for two types that share some property and that are not related by inheritance (`typeInheritance`), and also have no existing common supertype that has the property in question defined (`commonTypeProperty`).

- Domains in the redefinition of properties. If a property is redefined in a type, the property domain has to be a subdomain of the corresponding property domains as previously defined in the type supertypes. This fact is verified in the rule of deduction of fig. 4.8.f.

- A class cannot be a superclass of itself. The definition of this inconsistency for classes is equivalent to the already defined in fig. 4.8.b for types.

- There cannot be redundant superclasses in a class definition. The definition of this inconsistency for classes is equivalent to the already defined in figs. 4.8.c and 4.8.d for types.

(a)

```
inconsistency(T) :-
      typePropertyAtT(IdT,IdP,T1),
      typePropertyAtT(IdT,IdP,T2),
      T1 <> T2,
      T1 <= T,
      T2 <= T,
      !.
```

(b)

```
inconsistency(T) :-
      typeName(IdT,Type,T),
      supertype(IdT,IdT,T),
      !.
```

(c)

```
inconsistency(T) :-
      typeSupertypes(IdT,IdTs,T),
      redundantSupertypes(IdTs,T),
      !.
```

(d)

```
redundantSupertypes([Id|L],_) :-
      memberId(Id,L),
      !.
redundantSupertypes([Id|L],T) :-
      typeInherWithSomeType(Id,L,T),
      !.
redundantSupertypes([_|L],T) :-
      redundantSupertypes(L,T).

typeInherWithSomeType(IdT1,[IdT2|_],T) :-
      typeInheritance(IdT1,IdT2,T),
      !.
typeInherWithSomeType(IdT,[_|L],T) :-
      typeInherWithSomeType(IdT,L,T).
```

(e)

```
inconsistency(T) :-
      typeProperty(IdT1,IdP,T),
      typeProperty(IdT2,IdP,T),
      IdT1 <> IdT2,
      not(typeInheritance(IdT1,IdT2,T)),
      not(commonTypeProperty(IdT1,IdT2,IdP,T)),
      !.
```

(f)

```
inconsistency(T) :-
      typePropertySignature(IdT1,IdP,S1,T),
      typePropertySignature(IdT2,IdP,S2,T),
      IdT1 <> IdT2,
      supertype(IdT2,IdT1,T),
      not(signatureStrictSubdomains(S1,S2,T)),
      !.
```

Figure 4.8. Inconsistency deduction rules (I).

- Types of classes related by inheritance. The types of the superclasses of a defined class have to be either supertypes or the same type as that which corresponds to the class. Fig. 4.9.a, given the types of the superclasses of a class, it is verified that this condition is fulfilled by way of the `sameTypeOrSubtype` derived predicate presented in fig. 4.9.b.

- Single definition of properties between classes. Similar to the inconsistency defined in fig. 4.8.e for types; corresponds to fig. 4.9.c: a property can only be defined once, and inherited from the type of the class where it was originally defined.

(a)

```
inconsistency(T) :-
      className(IdCobjects,objects,T),
      classType(IdC,IdT,T),
      IdC <> IdCobjects,
      classSuperclasses(IdC,IdCs,T),
      classesTypes(IdCs,IdTs,T),
      not(sameTypeOrSubtype(IdT,IdTs,T)),
      !.
```

(b)

```
sameTypeOrSubtype(IdT,[IdT],_) :-
      !.
sameTypeOrSubtype(IdT,[IdT2],T) :-
      supertype(IdT2,IdT,T),
      !.
sameTypeOrSubtype(IdT,[IdT|L],T) :-
      !,
      sameTypeOrSubtype(IdT,L,T).
sameTypeOrSubtype(IdT,[IdT2|L],T) :-
      supertype(IdT2,IdT,T),
      !,
      sameTypeOrSubtype(IdT,L,T).
```

(c)

```
inconsistency(T) :-
      classType(IdC1,IdT1,T),
      classType(IdC2,IdT2,T),
      IdC1 <> IdC2,
      typeProperty(IdT1,IdP,T),
      typeProperty(IdT2,IdP,T),
      not(classInheritance(IdC1,IdC2,T)),
      not(commonClassProperty(IdC1,IdC2,IdP,T)),
      !.
```

Figure 4.9. Inconsistency deduction rules (II).

4.4.4 Output Requirements

In response to the event corresponding to the base predicate `csDefineEnd`, the system has just one output requirement: to produce a set of correct terms of definition of the conceptual schema. These terms have to be in a format adequate for the definition of DCMs that model other aspects of OODBs. In particular, the conceptual schema definition terms will be base predicates of the object definition DCM represented in fig. 4.1. So, the output terms represented in fig. 4.10 have been adapted to the format required by the derived predicates defined in the object definition DCM.

```
csType(Tany,any,[],[],ta)
...
csClass(Cobjects,objects,Tany,[],[C1,C4],tb)
...
csProperty(P1,name,t1)
csProperty(P2,address,t2)
csProperty(P3,category,t3)
csProperty(P4,city,t4)
...
csType(T1,person,[Tany],[P1,P2],t5)
csType(T2,employee,[T1],[P1,P2,P3],t6)
csType(T3,address,[Tany],[P4],t7)
...
csClass(C1,people,T1,[Cobjects],[C2,C3],t8)
csClass(C2,clients,T1,[C1],[],t9)
csClass(C3,employees,T2,[C1],[],t10)
csClass(C4,addresses,T3,[Cobjects],[],t11)
...
csTypeProperty(T1,P1,[t(Tstring)],t12)
csTypeProperty(T1,P2,[c(C4)],t13)
csTypeProperty(T2,P3,[t(Tstring)],t14)
csTypeProperty(T3,P4,[t(Tstring)],t15)
```

Figure 4.10. Conceptual schema DCM output.

4.5 Related work

In [Gray et *al*., 1992; Díaz et *al*., 1991] some proposals are made about implementing an OODB in Prolog. Unlike these, in this paper the development of DCM of different elements of OODBs in Prolog is proposed. Although the subject (OODBs) and the tool (Prolog language) are the same in both cases, the aspects taken into consideration are completely different.

In [Gray et *al*., 1992; Scholl et *al*., 1992; Lemke, 1995] among others, different metamodels of object-oriented models are presented. Metamodels and DCMs are different concepts and they are used with different objectives.

Following the focus presented in [Quer & Olivé, 1994], object-oriented DCM to specify any IS can be constructed -CIAM language [Gustafsson et *al*., 1982] would correspond to a materialisation of that focus, as pointed out there. Using some version of Prolog extended with concepts of object-orientation, executable object-oriented DCM in Prolog could also be defined.

4.6 Conclusions

The definition of DCMs in Prolog allows the specification of ISs by expressing only their logical component and, together with the formal specification, a prototype of the system is also obtained. The availability of prototypes of the elements which are specified is very useful. In the actual field of OODBs this possibility is very valuable, since we consider that no alternative means of creating prototypes in a simpler manner exists. We are so far unaware of similar experiences along these lines and consider that its application in this field may turn out to be of great use.

In order to specify the different elements of OODBs carrying out DCMs in a progressive way, a DCM definition architecture has been presented.

Part of the DCM of a conceptual schema definition system -and its corresponding data model- has been shown. The definition of predicates that has been carried out in this example is similar to the formal definitions found in chapter 3; the main difference is that in developing the DCM in Prolog a prototype of the system has been also obtained. This mechanism is of particular use in order to specify additional elements of OODBs, like, for example: richer semantic models [Castellanos et *al*., 1992], schema evolution [Peters & Özsu, 1995], definition of derived classes [Abiteboul & Bonner, 1991; Santos, 1995], definition of external schemas [Rundensteiner, 1992c; Samos, 1995].

In order to specify an external schema definition DCM according to [Samos, 1995], a conceptual schema definition DCM, an object definition DCM and the external schema definition DCM have been constructed -some derived predicates of the last one can be found in chapter 5.

5 A new external schema definition methodology

In this chapter, a new external schema definition methodology that considerably simplifies the definition process and the results obtained regarding other authors' proposals is presented. The ANSI/SPARC framework is taken as a reference. In section 5.1, the relationship between the conceptual schema and external schema is discussed together with the organisation of the data dictionary and the external schema definition system. In section 5.2 the different processes of integration needed in order to define an external schema, are reviewed (integration in the data dictionary and in the external schema) comparing them with alternative proposals. In order to reduce the number of classes that is required to be explicitly defined, a new proposal is made: different categories of classes can be defined when the classes composing the external schema are selected; classes can be qualified as transformable or non-transformable if they can be modified or not, respectively, before being included into the external schema. In section 5.3 different issues related with the process of generation of external schemas are studied, particularly those related with the transformation of transformable classes. In section 5.4 two external schema generation algorithms are presented. These algorithms are defined in the framework of the new external schema definition methodology for OODBs put forward in the previous sections. Lastly, in section 5.5, some conclusions are presented.

5.1 Definition of external schemas in the ANSI/SPARC framework

As mentioned before, the ANSI/SPARC framework defined a general architecture for DBMSs. This architecture has been widely applied for relational databases, but, as has been shown in chapter 2, this has not been the case for OODBs. Our basic idea is that the ANSI/SPARC architecture should also be applied in OODBs. Therefore, an interpretation of this architecture is given in this section.

Relating to the ANSI/SPARC framework, three main issues are considered: first, the relationship between the database conceptual schema and each one of the external schemas defined over it; secondly, the organisation of the data dictionary and, finally, the services offered by the external schema definition system.

5.1.1 Conceptual schema and external schemas

In OODBs, conceptual schema and external schemas have to be object schemas defined according to the object-orientation paradigm. In this sense, from the end-users point of view, there should be no difference between working over the conceptual schema or over an external schema: the data has the same kind of organisation in both schemas.

The information contained in the conceptual schema is represented by its classes. External schemas are derived from the database conceptual schema. The information contained in each external schema is derived from the information of the conceptual schema -this does not necessarily mean that the classes included in an external schema need to have been previously defined in the conceptual schema. An external schema may include classes defined in the conceptual schema just as it may also contain derived classes -directly or indirectly defined on the basis of conceptual schema classes- that, from our point of view, do not necessarily need to be included in the conceptual schema. These classes are defined and included in the data dictionary. A derived class may represent a relevant concept in an external schema, but this concept does not have to be especially significant to be explicitly represented in the conceptual schema, i.e., a derived class that hides some property of a class already included in the conceptual schema.

According to the classification of external schema definition methodologies made in section 2.2.2, our interpretation of the ANSI/SPARC architecture coincides with the second group of methodologies: external schemas are not necessarily subschemas of the conceptual schema.

Therefore, the two main ideas concerning the relationship between the conceptual schema and external schemas are:

- Conceptual schema and external schemas are object schemas defined according the object-orientation paradigm.

- External schemas can contain conceptual schema classes as well as classes derived from conceptual schema classes and not included in the conceptual schema.

5.1.2 Organisation of the data dictionary

The data dictionary contains all information relating to the management and use of the database system: information related to the definition of the different schemas -internal, conceptual and externals- of the database.

In order to define the conceptual schema or an external schema, the users of the information contained in the data dictionary are the *enterprise administrator* -by means of the *conceptual schema definition system*- or the *application administrator* -through the external schema definition system (see fig. 2.2). Therefore, information in the data dictionary should be organised in order to fulfil the requirements of these users (as well as other users' requirements not studied in this work).

Derived classes represent a customisation of part of the information contained in the conceptual schema in the form of new classes; they are defined from previously existing classes using object-oriented queries. If an external schema is to be defined containing some new class, this derived class has to be defined and integrated into the data dictionary together with the definition of the external schema. In section 2.3.1, two different groups of methods of integration of derived classes in a schema have been presented: integration using the inheritance relationship and integration using other relationships. In our opinion, the most suitable way of having derived classes relating to the classes from which they have been defined in the data dictionary is the derivation relationship, as defined in [Bertino, 1992; Monk, 1994; Kim & Kelley, 1995; Naja & Mouaddib, 1995; Bertino et *al*., 1996].

Therefore, the conceptual schema is contained in the data dictionary together with the definition of derived classes, and also with the definition of the external schemas. In the data dictionary, each derived class is related through the derivation relationship with the classes from which it has been defined; this relationship only appears in the data dictionary.

5.1.3 The external schema definition system

The definition of the external schemas is carried out by means of the external schema definition system over the data dictionary.

External schemas have to be closed with regard to the property relationship: any class referenced by a class included in an external schema has to be included in the same external schema (see section 3.2.2). Also, external schemas have to be valid according to the inheritance relationship -all required inheritance relationships are defined in the schema, and not redundant, neither inconsistent relationships should be included, as defined in section 3.3.1.

In order to have external schemas correctly defined, one possibility is that application administrators define manually all the components of the schema; to avoid possible mistakes a validation system should be provided. Another possibility is to have the external schema definition system automated -application administrators only select the classes to be included in the external schema, and the system generates the resulting external schema. The later is our choice. Therefore, generation algorithms of the external schema from a set of classes have to be provided.

5.2 Integration of derived classes in the schemas of an OODB

In order to make the reasons of some of the decisions expressed in the previous section in this section clear, the different processes of integration needed in order to define an external schema are reviewed.

Given the conceptual schema of fig 5.1 (the same example schema used in section 2.3.1 to illustrate the different possibilities of integration of a derived class in a schema), and the derived class EMPLOYEES' -defined from class EMPLOYEES hiding the Salary property and selecting objects that are not manager employees- an external schema with the same structure of the conceptual schema but with the new class EMPLOYEES' replacing class EMPLOYEES is needed to be defined.

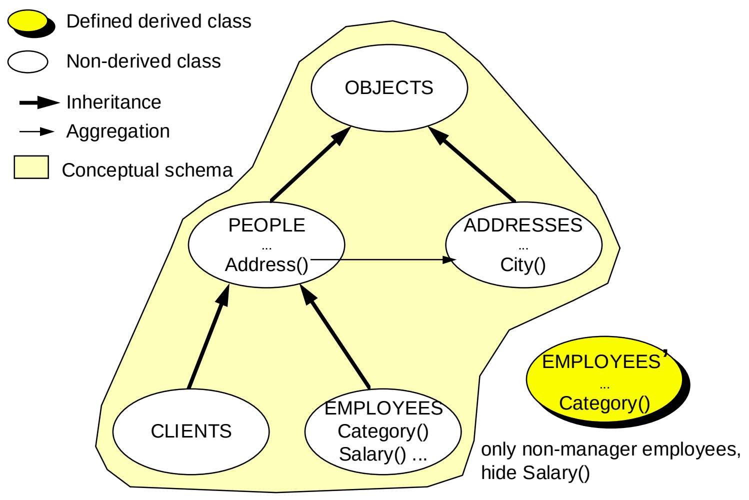


Figure 5.1. Conceptual schema and definition of a derived class.

5.2.1 Integration in the data dictionary

Derived classes do not need to be included in the conceptual schema before being included in an external schema, derived classes are defined and included in the data dictionary (different methods of integration were given in section 2.3.1).

In this case, the purpose of having the data dictionary is to define external schemas. If inheritance is considered as a way of relating new derived classes with the rest of the classes, frequently, as happens in our example with class EMPLOYEES', a new derived class will not be directly related by inheritance with its base classes. Therefore, in order to have explicitly defined all the existing inheritance relationships between the classes in the data dictionary, some intermediate classes have to be generated, as proposed in [Rundensteiner, 1992b; Rundensteiner, 1992c], and was shown in fig. 2.6.

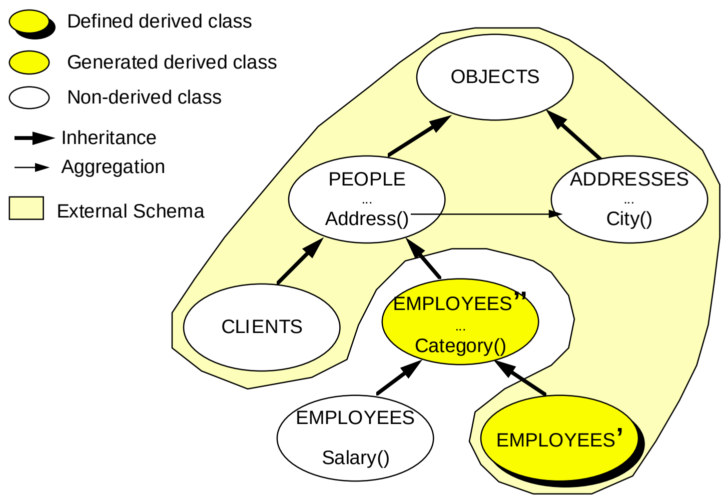


Figure 5.2. Automatic integration by inheritance. External schema class selection (I).

Fig. 5.2 shows the selection of the set of classes that will compose the external schema corresponding to the example considered. It can be seen that, in order to integrate the new defined class, a class (EMPLOYEES") has been generated automatically and added to the data dictionary, but this class is not included in the external schema. This class represents exclusively the inheritance relationship between the original base class (EMPLOYEES), and the defined class (EMPLOYEES'). However, the external schema only includes one of them. In most cases, the only schema to contain both of them will be the data dictionary (or the conceptual schema playing the role of data dictionary in the Rundensteiner's methodology) and it would be useful only for defining more external schemas; even for this operation it will be problematic, because complex schemas not meaningful to the external schema definer may be obtained.

An alternative solution is to use other kind of relationships in order to integrate derived classes into the data dictionary. From the existing relationships (presented in section 2.3.1.2), we selected the derivation relationship [Bertino, 1992; Monk, 1994; Kim & Kelley, 1995; Naja & Mouaddib, 1995; Bertino et *al*., 1996]. However, the definition of clusters of classes [Heuer & Sander, 1991] can also be a suitable solution.

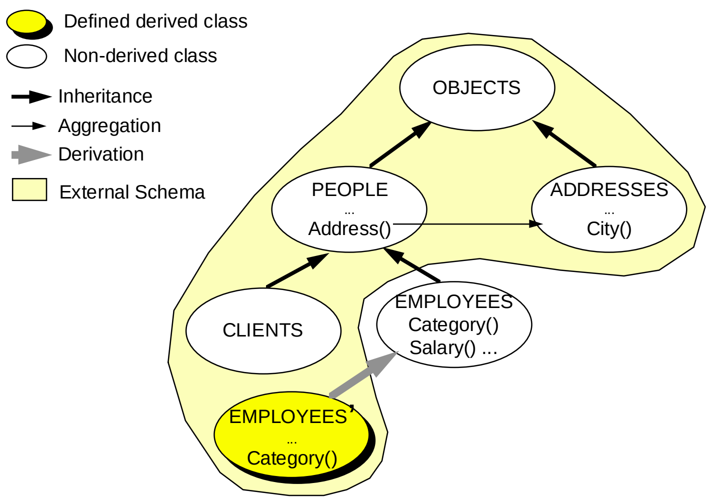


Figure 5.3. Integration by derivation. External schema class selection (I).

As is shown in fig. 5.3, the new derived class is directly integrated into the data dictionary by means of derivation relationship, without generating any additional class.

5.2.2 Integration in an external schema

When the new derived classes are integrated into the data dictionary, as can be seen in figs. 5.2 and 5.3, a set of classes can be selected to form an external schema. Obtaining a schema closed in regard to the property relationship is independent of the method of integration of derived classes in the data dictionary. However, obtaining the inheritance relationships will depend on the way in which derived classes have been integrated.

In the case of integration of derived classes by inheritance in the data dictionary, as can be seen in fig. 5.2, two classes will be related by inheritance in an external schema if and only if, they are related by inheritance (directly or indirectly) in the data dictionary -the integration process in the data dictionary consists on defining explicitly all the inheritance relationships between the classes, therefore, all the inheritance relationships between classes in the external schema have been already defined, and can be directly obtained.

The effort of integration has been already made, integrating the new derived classes into the data dictionary.

If the integration into the data dictionary has been made using the derivation relationship, some of the inheritance relationships between the classes in the external schema have to be further discovered. If two classes included in an external schema are related by inheritance in the data dictionary (in the conceptual schema or in an external schema previously defined), they will also be related by inheritance in the external schema. However, there may be classes not related by inheritance in the data dictionary, and an inheritance relationship between them nevertheless exists. In the example of fig. 5.3, the inheritance relationship between EMPLOYEES' and PEOPLE has to be obtained. In this case, the effort of integration has to be done when derived classes have been selected to become part of an external schema, but in this case, the quantity of classes to take into account into the integration process is less than in the previous case (only the set of classes selected to compose the external schema vs. all the classes in the data dictionary).

5.2.3 Effort of integration in the data dictionary vs. in the external schema

As has been shown in the previous point, the main effort of integration can be done by the process of integration in the data dictionary (if derived classes are integrated by inheritance in the data dictionary), or in the generation of the external schema (if derived classes are integrated by derivation into the data dictionary).

In order to show more clearly the differences in the results obtained between the two different methods of integration, let's study an additional example. Starting from the conceptual schema in fig. 5.1, an external schema with similar structure is needed to be defined, but instead of having the property Address in PEOPLE, CLIENTS and EMPLOYEES, it has a property City, already defined in class ADDRESS; and it also hides the property Salary in class EMPLOYEES.

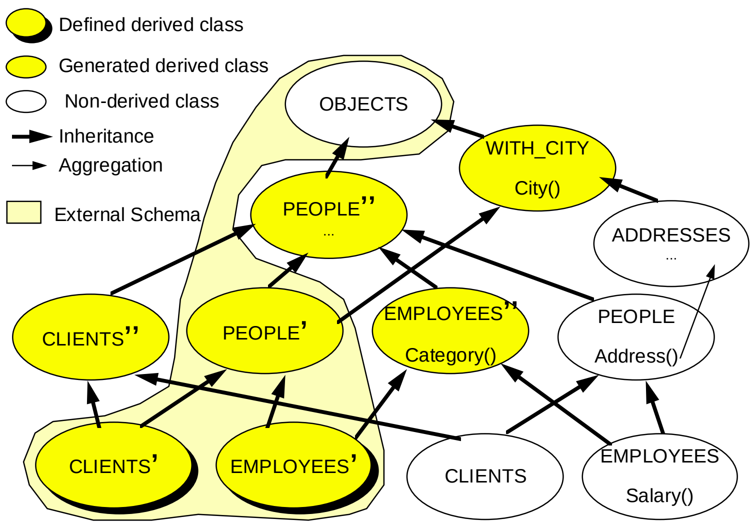


Figure 5.4. Automatic integration by inheritance. External schema class selection (II).

The definition of this external schema, if derived classes are integrated into the data dictionary according to Rundensteiner's algorithm [Rundensteiner, 1992b], can be seen in fig. 5.4. First, CLIENTS' and EMPLOYEES' classes are defined according to requirements. Then, they are integrated into the conceptual schema, generating in this operation some additional classes. It is not necessary to define the new class PEOPLE' because it is generated by the integration algorithm. Finaly, some classes are selected to obtain the defined external schema.

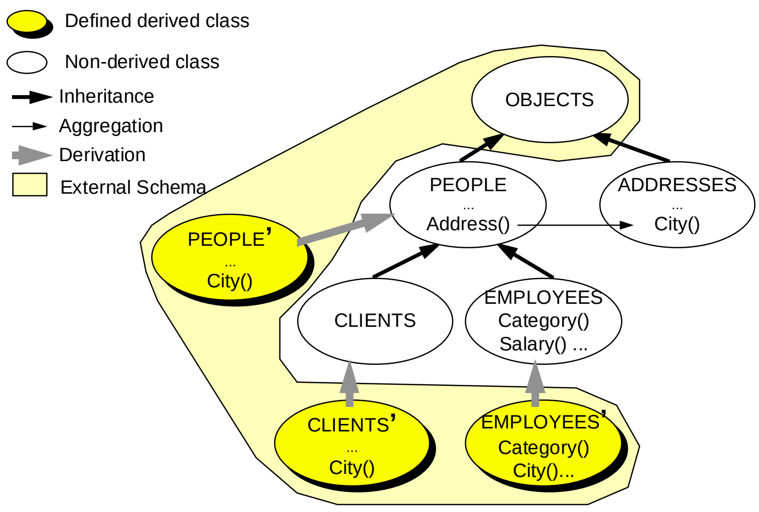


Figure 5.5. Integration by derivation. External schema class selection (II).

If integration of derived classes in the data dictionary is made using the derivation relationship, as can be seen in fig. 5.5, the integration process does not have to generate any additional derived class. However, all the classes needed have to be explicitly defined, and the data dictionary only contains the classes that are strictly necessary in order to define the external schema.

As can be seen in fig. 5.4, if the effort of integration is made in order to integrate by inheritance the new derived classes in the data dictionary, then:

- A great number of classes are generated automatically, and only a few of them are directly used in the external schema defined (all the classes included in the data dictionary have to be taken into account in the integration process).

- Some of the classes required to be included in the external schema do not have to be explicitly defined as they are generated in the process of integration of the rest of the classes explicitly defined.

- The inheritance relationships between the classes selected to compose the external schema can be directly obtained because all the existing inheritance relationships between the classes have already been defined in the data dictionary, therefore, two classes will be related by inheritance into an external schema if and only if they are related by inheritance (directly or indirectly) into the data dictionary.

On the other hand, if derived classes are integrated in the data dictionary using the derivation relationship, as can be seen in fig. 5.5, then:

- No classes are automatically generated in the process of integration of derived classes into the data dictionary.

- All the required classes have to be explicitly defined.

- The effort of integration has to be done in the process of generation of the external schema, inheritance relationships not defined in the data dictionary have to be discovered, but only the set of classes selected to compose the external schema have to be considered.

5.2.4 Qualification of the classes selected to compose the external schema

In order to simplify the external schema definition process, reducing the number of classes that is required to be explicitly defined, an alternative solution is to define different categories of classes when the selection of the classes to compose the external schema is made.

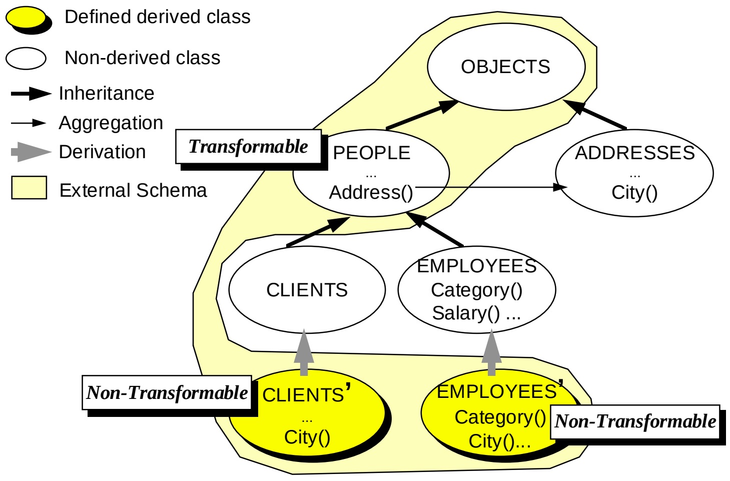


Figure 5.6. Qualification of the classes selected to compose the external schema.

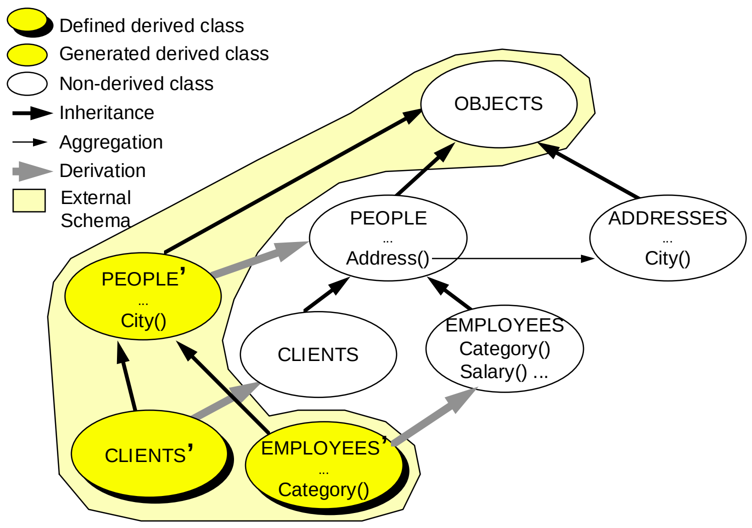


Figure 5.7. Obtained external schema.

Two kinds of classes are distinguished: transformable and non-transformable. Non-transformable classes have to be added to the external schema directly (in the examples considered in previous sections, all classes were non-transformable). Transformable classes can be replaced by another new derived class into the external schema if necessary. This new class is the result of modifying the original transformable class in the sense of adding or removing properties.

Returning to our example, if the class PEOPLE is qualified as transformable and the rest of the classes selected are qualified as non-transformable, as shown in fig. 5.6, the corresponding external schema obtained is shown in fig. 5.7. Therefore it can be seen that the class PEOPLE is replaced in the external schema by the generated class PEOPLE'. This new class is defined according to the structure of the non-transformable classes belonging to the external schema specified. In this case, only strictly necessary classes for the external schema are generated.

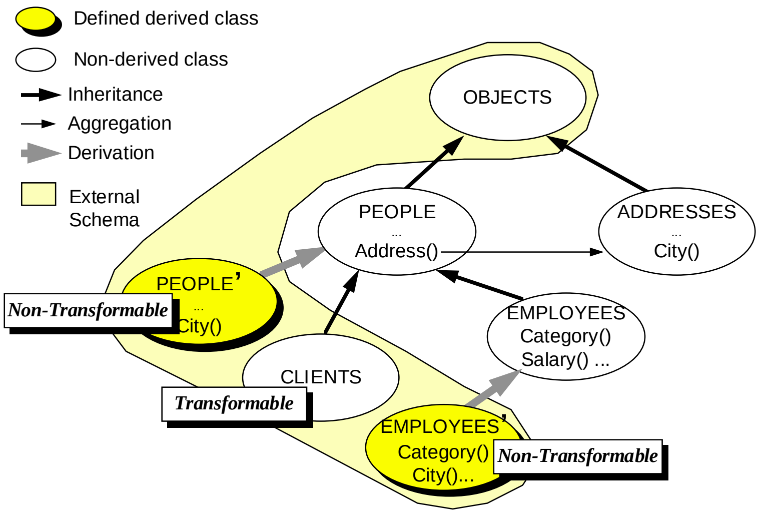


Figure 5.8. Alternative external schema definition.

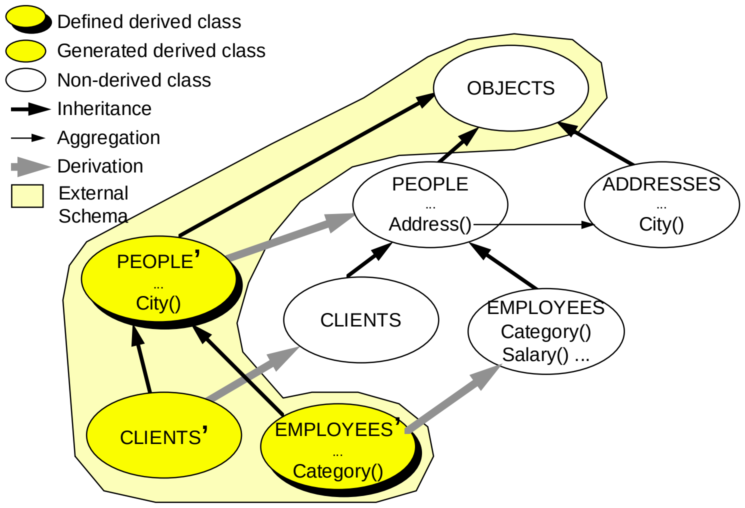


Figure 5.9. Obtained external schema.

There may be other ways of defining the same external schema; as shown in fig. 5.8. In this case, the derived classes PEOPLE' and EMPLOYEES' are explicitly defined, the derived classes PEOPLE' and EMPLOYEES' are qualified as non-transformable, and the class CLIENTS is qualified as transformable. With these conditions, the external schema generated (shown in fig. 5.9) is equal to the one of fig. 5.7. However, the set of generated classes is different since the class CLIENTS' has been generated from the transformable class CLIENTS.

5.3 Generation of external schemas

According to the previous of sections, the steps necessary in order to define an external schema are:

- Definition and integration of the requited derived classes in the data dictionary - derived classes are directly integrated using the derivation relationship. This topic is further studied in chapter 6.

- Selection of the set of classes that will constitute the external schema -classes can be qualified as transformable or non-transformable (by default, a class is considered non-transformable).

- Generation of the external schema.

The different alternatives considered in the generation process of the external schema are presented in this section.

5.3.1 Transformations of transformable classes

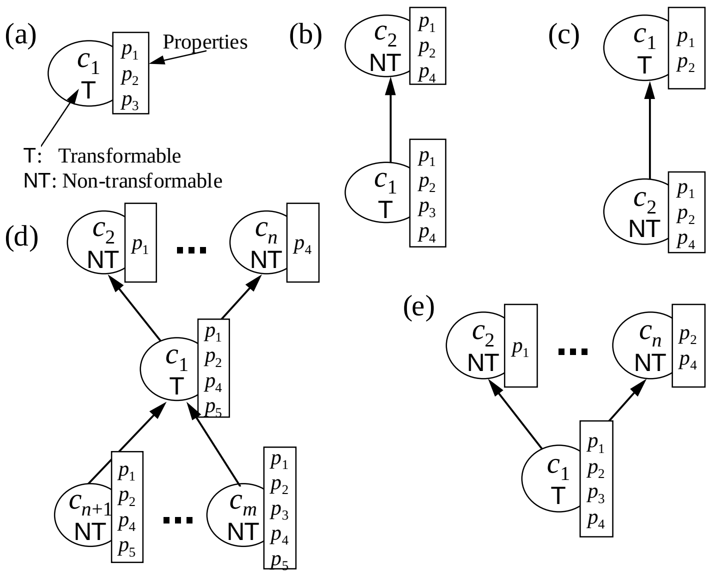


Figure 5.10. Integration of a transformable class.

In order to have a transformable class properly integrated into a class hierarchy, it may have to suffer different transformations, adding or removing properties. The transformations that can be carried out in a transformable class are represented in fig.

5.10. Given a transformable class c_1, to be integrated (fig. 5.10.a), if it has to be subclass of a class c_2, which subsumes it in extension, then the class c_1 inherits the properties of c_2 and, furthermore, maintains its own properties (fig. 5.10.b). However, if the class c_1 has to be a superclass of class c_2, then class c_2 conditions the structure of the transformable class c_1 in such a way that any properties which are not defined for c_2 must be eliminated from c_1 (fig. 5.10.c).

Transformable classes inherit the properties of the classes defined to be their superclasses, and lose the properties not defined in the classes selected to be their subclasses (fig. 5.10.d). If all the classes that are subclasses of a transformable class have some properties in common, and these properties are defined for all the objects of the transformable class, then these properties will also be added to the transformable class (property p_5 in fig. 5.10.d). Finaly, if a transformable class only has superclasses, it doesn't lose any property, it inherits all the properties of its superclasses (fig. 5.10.e).

Therefore, the transformations that a transformable class can suffer when it is integrated into a class hierarchy:

- Addition of the properties of its superclasses not defined within it.

- Addition of the properties common to all of its subclasses, defined for all of its objects and not defined in it.

- Elimination of the properties not defined in all of its subclasses.

5.3.2 Order in the transformation of transformable classes into a class hierarchy

In the transformations presented in the previous point only one transformable class was considered -all the classes with which the transformable class were related were non-transformable classes. Another possibility is to have more than one transformable class integrated in the class hierarchy and then proceed transforming all of them.

Therefore, in defining the order of transformation of the transformable classes, two main situations are considered:

- Gradual integration of the transformable classes into the external schema.

- External schema with all the transformable classes already integrated.

5.3.2.1 Gradual integration of transformable classes

Regarding the class hierarchy formed exclusively by non-transformable classes, transformable classes can be integrated gradually into it. Therefore, each transformable class will be related only with non-transformable classes. A transformable class can be transformed in the integration process but once integrated, it becomes non-transformable.

As has been shown in fig. 5.10.b and 5.10.c, a transformable class can have properties added or eliminated, depending on its relationships with non-transformable classes in the class hierarchy. In general, superclasses make transformable classes gain new properties, but subclasses make them lose properties. Therefore, if one wants a previously transformable class (c_2) already integrated does not cause the effect of losing properties in the integration of another transformable class (c_1) in the way shown in fig. 5.10.c, the transformable classes that are not subsumed in extension by other transformable classes have to be integrated first. Therefore, transformable classes that have already been integrated can only make other transformable classes gain new properties (as shown in fig. 5.10.b).

5.3.2.2 Schema with all the transformable classes integrated

If a class hierarchy is defined containing several transformable classes, the order in which the transformations of the transformable classes are carried out may affect the end result. With the intention of keeping as much properties as possible in the transformable classes, the first transformations to be applied should be the addition of properties of their respective superclasses or subclasses.

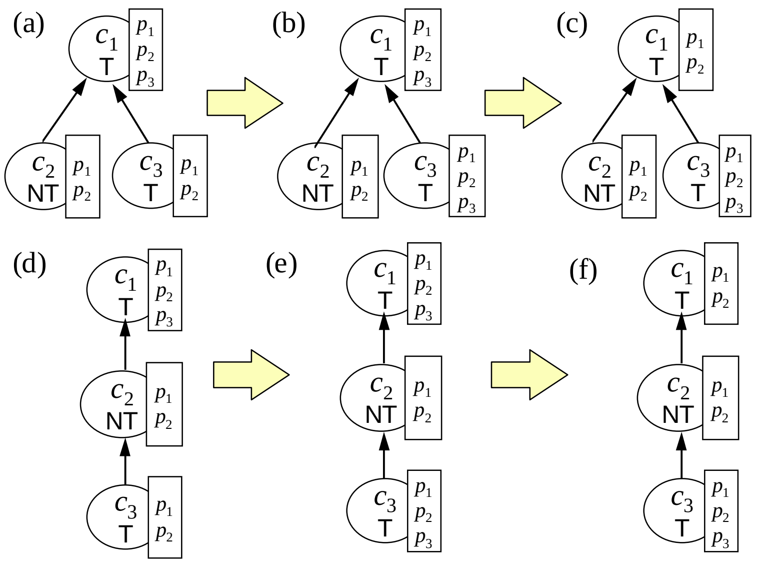


Figure 5.11. Addition of properties of the superclasses.

In fig. 5.11 two different cases of adding properties defined in the superclasses of a transformable class are shown. Figs 5.11.a and 5.11.b correspond to the first case, they show respectively the situation before and after this transformation. Figs. 5.11.d and 5.11.e correspond to the second case. In both situations property p_3 is inherited by class c_3 from class c_1. If another kind of transformation was carried out before this one (eliminate the properties not defined in all the subclasses of a transformable class), property p_3 may not have been defined for class c_3. The final result of the transformation process is shown in figs. 5.11.c and 5.11.f, respectively. As can be seen, in both cases property p_3 is eliminated from class c_1, but it remains defined in class c_3. If integration were made gradually, property p_3 would not have been defined in class c_3 because it would have been previously eliminated from class c_1 when it was integrated.

An example of transformation consisting of the addition of properties taken from the subclasses of the transformable class is shown in fig. 5.12. If all the non-transformable classes that are subclasses of a transformable class have some properties in common (in fig. 5.12.a, c_2 and c_3 have property p_2 not defined in c_1), and these properties are defined for all the objects of the transformable class, then, these properties will also be added to the transformable class (property p_2 is added to c_1, fig. 5.12.b). And the new properties added can be propagated to its subclasses (property p_2 is added to c_4, fig. 5.12.c). If all the subclasses of a transformable class are transformable classes, any property defined in one of the subclasses may be added to the transformable class in the root -if the property is defined for all its objects.

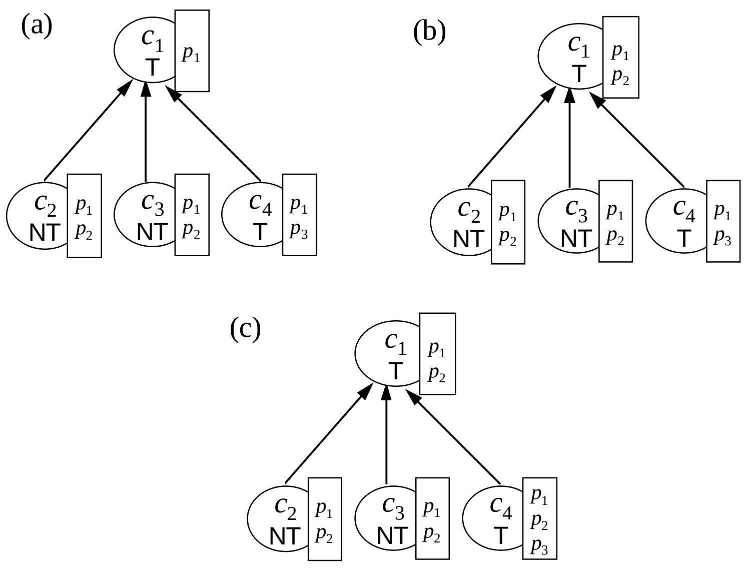


Figure 5.12. Addition of the properties of the subclasses.

Once all the possible properties of the superclasses and subclasses have been added to the transformable classes, the elimination of the properties not defined in all of their subclasses has to be carried out, as shown in figs. 5.11.c and 5.11.f. Therefore, before eliminating properties of a class, all of its subclasses should have been previously transformed.

5.3.3 Inheritance between transformable and non-transformable classes

Once the set of transformable and non-transformable classes has been selected, in order to transform the transformable classes, the inheritance relationships between the selected classes have to be obtained. These inheritance relationships determine the way transformable classes will be transformed.

One possibility is to define the inheritance relationships between the classes selected manually in the moment in which the selection is made -these relationships have to be automatically verified (like it was done in the case of manual integration of derived classes by inheritance) in order to only allow correct definitions. Another possibility is to obtain automatically all the existing inheritance relationships. This case will be further studied here.

5.3.3.1 Characteristics of the class hierarchy

Regarding the problem of obtaining the inheritance relationships between the set of classes selected, like for the problem of integrating automatically by inheritance a derived class in a class hierarchy, two solutions are considered depending on the desired characteristics of the resulting class hierarchy:

- Obtain a class hierarchy closed related to the inheritance relationship, adding to the set of classes selected the additional classes needed (as the Rundensteiner's methodology [Rundensteiner, 1992c] does in the conceptual schema).

- Obtain all the inheritance relationships exclusively considering the set of classes selected.

5.3.3.2 Subsumption relationships

In the case of non-transformable classes, their intension and extension is totally pre-determined: On the other hand, transformable classes have their extension pre-determined, but not their intension. The intension of transformable classes can be modified in order to be adapted to the position of the class in the class hierarchy related to the rest of the classes. Therefore, the relationships between non-transformable classes have to be obtained (or only checked) according to their intension and extension, but relationships between transformable classes or between transformable and non-transformable classes would have to be obtained (or only checked) only according to their extension.

The inheritance relationships between the classes can be obtained using a *subsumes()* function (see section 2.3.2), also based on the relationships already existing in the data dictionary (between the classes selected, or between its corresponding base classes, taking into account the definition of the derived classes). The *subsumes()* function refers to the intension and extension of the classes, but for transformable classes only their extension is considered.

In the case of non-transformable classes the relationships obtained between them will depend exclusively on the *subsumes()* function used. On the other hand, in the case of transformable classes (in relation to other transformable or non-transformable classes) as their intension can be modified, different possibilities may exist.

5.3.3.3 From subsumption to inheritance for transformable classes

If the extension of a transformable class c_1 is subsumed by the extension of other classes (transformable or non-transformable) but it does not subsume any other class in the class hierarchy (fig. 5.13.a), the transformable class can be defined as a subclass of all the classes that subsume it. Therefore, subsumption of extension relationships can be transformed directly into inheritance relationships (as shown there). Also, if the transformable class c_1 is subsumed by some classes and it also subsumes other classes, but all of the classes subsumed by c_1 are subclasses of the classes that subsume it, (represented in fig. 5.13.b), then class c_1 can be defined as a subclass of all the classes that subsume it and a superclass of all the classes subsumed by it (as shown in fig. 5.13.b).

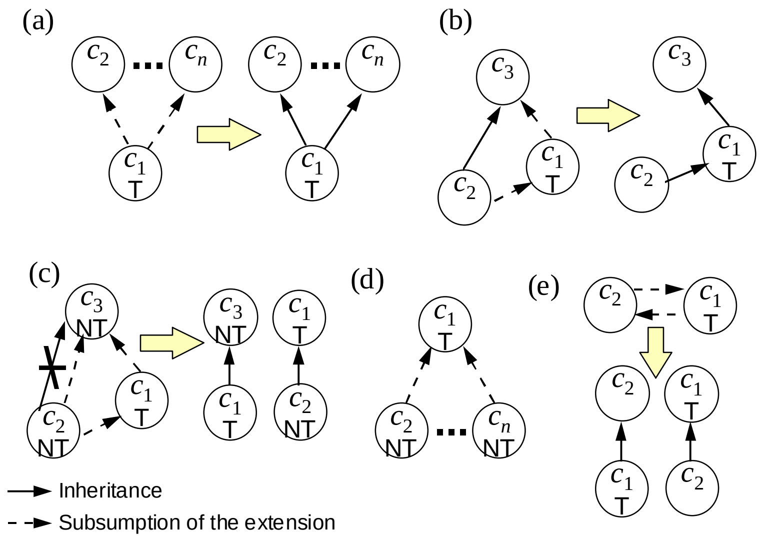


Figure 5.13. Relationships between transformable classes and other classes.

In the two cases represented in figs. 5.13.a and 5.13.b, the inheritance relationships between transformable classes and the rest of classes can be directly defined from the subsumption of extension relationships. In other situations, in the transformation of subsumption relationships between transformable classes and other classes into inheritance relationships, different possibilities may exist. For example, in the case represented in fig. 5.13.c, the transformable class c_1 is subsumed by a non-transformable class c_3; c_1 also subsumes other non-transformable class c_2, however c_2 is not a subclass of class c_3. Therefore, in transforming the subsumption relationships into inheritance relationships different possibilities may exist: to define c_1 as a subclass of c_3, or to define c_1 as a superclass of c_2. Nevertheless it is not possible to define both of them in the same schema because c_3 is not a superclass of c_2. Another possibility in this case, is to consider the transformable class c_1 twice in the class hierarchy, and to generate two different classes in the resulting schema (one from its relationship with c_3, and the other the relation with c_2).

If a transformable class (c_1) subsumes in extension a set of non-transformable classes (fig. 5.13.d) which do not have any properties in common, and it is defined as a superclass of all of them, the result of the transformations on the transformable class would be a class without properties. However, this would not be an acceptable result. Therefore, the transformable class may be defined as a superclass of only a subset of the classes subsumed by it, in this case, different possibilities may be considered (different subsets of classes with properties in common can be defined).

Lastly, if a transformable class (c_1) and another of the selected classes have the same extension (fig. 5.13.e, the classes subsume in extension each other) the transformable class can be defined as a superclass as well as a subclass of the other class in the class hierarchy and both possibilities can be valid, even though, both of them can not be defined in the same class hierarchy. Therefore, one of them has to be selected. Another possibility can be to have the class c_1 twice in the class hierarchy, once as a subclass and another as a superclass of the class which has the same extension and generate two different classes from the class c_1 after the transformations. If both classes c1 and c2 are transformable classes, potentially they will become the same class after transformations.

In order to avoid the situation in which many possibilities of defining inheritance relationships between transformable classes and other classes may exist, different solutions can be considered:

- Forbid the selection of sets of classes in which these situations occur.
- Present all the subsumption relationships between transformable classes and other classes to the application administrator and let him/her select the desired inheritance relationships.
- Define some criteria of automatic selection of the existing subsumption relationships that have to be transformed into inheritance relationships.
- Have the transformable classes included more than once in the class hierarchy and generate different classes from each one of them.

5.3.4 Property relationship closure

A class hierarchy is closed in relation to the property relationship if and only if, for all the classes included in the class hierarchy, all the classes related via a property relationship with them are also included in the class hierarchy.

5.3.4.1 Classes referenced by transformable and non-transformable classes

Given the set of transformable and non-transformable classes selected in order to compose an external schema, if a property of a non-transformable class references some other class (a property relationship exists between them), then, the referenced class has to be included into the set of classes selected in order to compose the external schema. On the other hand, if a transformable class is related by a property relationship with other classes not included in the mentioned set of classes, the transformable class can be transformed in order to avoid the inclusion of the referenced classes into the external schema. This kind of transformation of transformable classes is exclusively due to the set of classes selected to compose the external schema, whereas the transformations presented in point 5.3.2 are due to the position of the transformable class within the class hierarchy.

An example of this kind of transformations can be seen in figs. 5.8 and 5.9. In fig. 5.8 the class CLIENTS references class ADDRESSES (inherits the property Addresses from class PEOPLE). Class CLIENTS is qualified as transformable so, in order to avoid that the referenced class ADDRESSES be included into the external schema, class CLIENTS loses that property (as can be seen in the resulting external schema of fig. 5.9), otherwise, the resulting external schema would have been the one represented in fig. 5.14.

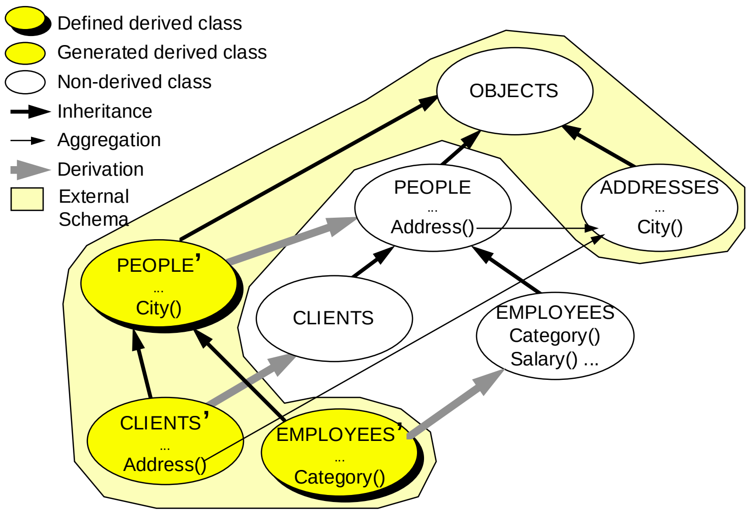


Figure 5.14. Obtained external schema.

5.3.4.2 *Required properties of a class*

A non-transformable class may also reference other classes already included in the set of selected classes -these classes can be transformable or non-transformable. A non-transformable class may reference some of the properties (methods) defined in the classes that it references. These properties may also reference other properties defined in the same class or in other classes -referenced indirectly by the non-transformable class. If a property is referenced directly or indirectly by a non-transformable class, and is defined in a transformable class, then the class resulting by transforming the transformable class has to include the referenced property. Therefore, the properties of transformable classes can be *required* if they are referenced by a non-transformable class, or *non-required* in other case.

Each property in a class (transformable or non-transformable) is related to the set of properties, from the same class or from other classes, that are needed in order to have that property defined. All the properties of non-transformable classes are required properties. The set of properties needed by a required property are also required properties, and the referenced classes in which required properties are defined also have to be included into the external schema at least with the required properties defined. If a required property of a transformable class references a class that has not been initially selected to compose the external schema, it also has to be included into the set of classes. If a property (non-required) of a transformable class is eliminated in a transformation of the class, all the properties of this class or other transformable classes that reference the eliminated property have to be eliminated too. These properties will be non-required. If they were required properties, then the property initially eliminated would have also been required.

The classes referenced by transformable or non-transformable classes, additionally included in the set of classes, can also be considered as transformable or non-transformable. The reason for including these classes in the set of selected classes is that some of their properties are required directly or indirectly by non-transformable classes.

Therefore, if they are considered as transformable classes, all the requirements about them will be fulfilled.

5.3.4.3 Non-transformable classes referencing transformable classes

When a transformable class is transformed, a new class is generated. Therefore, if a non-transformable class references some transformable classes, and the transformable classes are replaced by new classes in the external schema then, the non-transformable class has to be modified in order to replace its references to transformable classes using references to the new generated classes. If references to other classes in non-transformable classes have to be modified, then, new classes have to be generated that replace these non-transformable classes in the external schema.

In the process described above, non-transformable classes may have to be replaced by new classes in the external schema in order to change its references to other new classes generated. In order to avoid this, a simplification of the external schema generation process consists in requiring that any class referenced by a non-transformable class has to be qualified as non-transformable. Therefore, transformable classes referenced by non-transformable classes have to be re-qualified as non-transformable and the classes referenced by (originally or re-qualified) non-transformable classes that were not included initially in the set of selected classes, have to be included qualified as non-transformable classes too. This simplification is a limitation because less classes than in the previous process can be automatically transformed -there will be more classes qualified as non-transformable than before. Therefore, these classes would have to be explicitly defined.

5.3.4.4 Transformation of references to classes

If a non-transformable class or a required property of a transformable class references another class not included in the set of classes selected to compose the external schema, contrary to what has been said before, this second class does not have to be necessarily included in the external schema. If some derived class has been defined from the class originally referenced, and this derived class is included in the set of classes selected, and it also has all the required properties and extension then, the referenced class can be replaced by the derived class in the external schema. If more than one derived class satisfying these conditions are included in the set of classes, all the possibilities can be shown to the application administrator and one of them selected. Another possibility can be to forbid this situation, and only allow one class satisfying these conditions.

In fig. 5.15 an example is shown in which an external schema is to be defined, with the same classes of the original conceptual schema but replacing the class ADDRESSES by a new class ADDRESSES' which does not include the property City. With the behaviour described above, the only class that has to be explicitly defined is class ADDRESSES'. The rest of the classes will be automatically adapted to the new situation producing the desired external schema, as shown in fig 5.16.

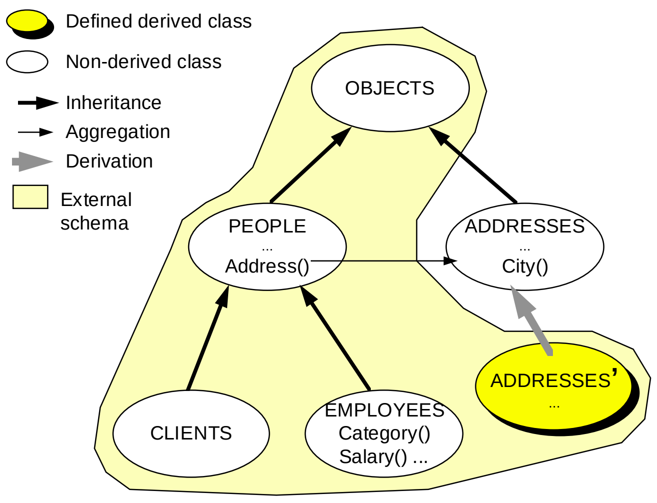


Figure 5.15. References to new classes.

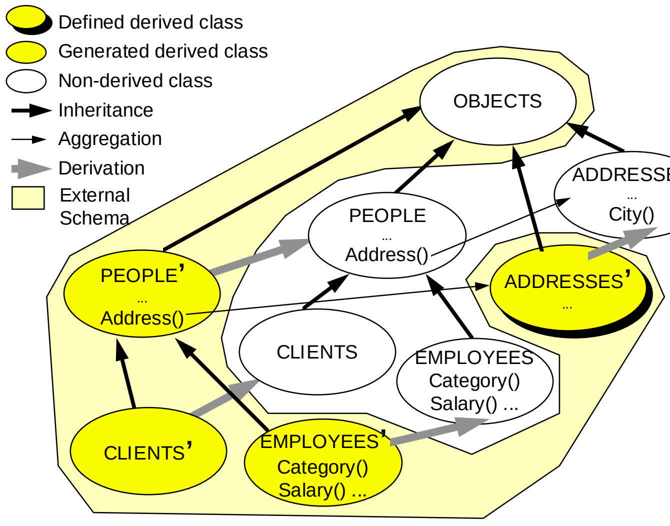


Figure 5.16. Obtained external schema.

5.3.5 Alternatives in the process of definition of external schemas

In previous sections, different alternatives for the definition of external schemas have been considered in different contexts, all of them according to the ANSI/SPARC architecture; in this section, the main ones of these alternatives are presented as a whole.

Organisation of the external schema

- External schemas have to be closed according to the inheritance relationship.
- External schemas do not have to be necessarily closed according to the inheritance relationship.

Qualification of the classes selected

- Qualify the classes selected to compose the external schema as transformable or non-transformable classes.
- Classes selected can not be transformed, therefore all the classes have to be non-transformable.

Order of the transformation of classes
If classes are qualified as transformable or non-transformable, transformable classes can be transformed differently depending upon the way in which they are integrated:

- Gradually integrated in the class hierarchy and transformed as they are integrated (at each moment, in the class hierarchy there is only one transformable class).
- First integrated and then transformed according to their situation (on relation to the rest of transformable or non-transformable classes).

Therefore, different external schema generation algorithms can be defined selecting different combinations of the alternatives presented.

5.4 External schema generation algorithms

In this section two external schema generation algorithms are presented. These algorithms are defined in the framework of the new external schema definition methodology for OODBs put forward in the previous sections. They generate an external schema from a set of classes selected from those existing in the data dictionary.

In both cases the external schema has to be closed with regard to the inheritance relationship. In the basic algorithm all the classes selected have to be non-transformable. The extended algorithm generates an object schema from a set of classes which are qualified as transformable or non-transformable. In the extended algorithm classes are gradually integrated in the class hierarchy.

We have carried out a specification of the external schema definition system by defining a DCM of it, as has been shown in chapter 4. This method is especially useful in the specification of elements of OODBs. The definition of DCMs is a technique which allows the specification of information systems by expressing only their logical component [Olivé, 1989]. If Prolog is used for the construction of the DCM, together with the formal specification, a prototype of the system is also obtained.

In the defined DCM, the external schema generation algorithms are implemented as derived predicates. In DCMs, derived predicates have time T as a component; in this case, time is not relevant for the presentation of the algorithms, so we shall not mention this aspect any further.

5.4.1 External schema generation basic algorithm

Given the set of classes *Cs* selected from the data dictionary, by means of the generation basic algorithm, a minimal object schema *S* = (*Cs'*, *Es*) defined from *Cs*, is generated.

According to the definition of object schema made in section 3.3, the class **objects** has to be included in the set of classes *Cs'*; the object schema *S* has to be valid, and also has to be closed in relation to the inheritance and the property decomposition relationships.

In the defined DCM, the external schema generation algorithm is a derived predicate which can be seen in fig. 5.17.

```
generateExternalSchema(S,Cs,Es,T) :-
       classSetSelection(S,Cs1,T),                          (a)
       includeElement(c(objects),Cs1,Cs2),                    (b)
       propertyDecompositionHierarchyClosure(Cs2,Cs3,T),        (c)
       classInheritanClosure(Cs3,Cs4,Es1,T),                  (d)
       eliminateReduntantEdges(Es1,Es2,T),                      (e)
       classesExistingInDataDictionary(Cs4,Cs,Es2,Es,T),      (f)
       !.
```

Figure 5.17. *GenerateExternalSchema* predicate.

Given the identifier *S* of the schema to be generated, the set of classes *Cs* and the edges *Es* are obtained. First, (fig. 5.17.a) the initial set of classes Cs_1 associated with the identifier *S* of the schema must be defined in the data dictionary. Fig. 5.17.b, the class **objects** must be included in the initial set of classes, being added if it was not previously there (the set of classes Cs_2 is obtained). Fig. 5.17.c, the set obtained must be closed in relation to the property relationship over the referenced classes that have not been included in the initial set must be added (the set of classes Cs_3 is obtained). Fig. 5.17.d, the classes in question together with the set of existing inheritance relationships between them and the necessary additional classes, defined in the data dictionary, must form a closed schema in relation to the inheritance relationship (the set of classes Cs_4 and the set of edges Es_1 are obtained). The schema must be valid in relation to the inheritance relationship, on account of the defined steps redundant inheritance relationships may be generated. For this reason they must be eliminated -fig. 5.17.e, the set of edges Es_2 is obtained. In the process of obtaining a closed schema in relation to the inheritance relationship some derived classes may have been generated. In the case that some class, according to the characteristics of some one of the generated classes, previously existed in the data dictionary, the generated classes are replaced by those classes in the external schema and its generation its cancelled, fig. 5.17.f.

In the following subsections steps c, d and e of the algorithm in fig. 5.17 are set out.

5.4.1.1 Property decomposition closure

Given a set of classes (Cs_1), a new set of classes (Cs_2) is obtained which is the transitive closure in relation to the property relationship of the classes in Cs_1. Every class referenced by a property of a class of the resulting set Cs_2 also belongs to Cs_2.

In fig. 5.18 the predicate that defines this property is presented. To specify it in Prolog, as is indicated in fig. 5.18.a, an auxiliary predicate is defined which uses a kind of incomplete structure called difference-list[1]. Explained in an intuitive fashion, by way of `dl`(Cs_2,Cs_1) it is expressed that the list Cs_2 that must be calculated initially contains the

[1] A difference-list is an incomplete structure that represents the difference between two lists. We represent them by `dl`(*As*,*Bs*), where *As* is the head of the difference-list and *Bs* the tail. `dl`([1,2,3|*Xs*],*Xs*) is the most general difference-list representing the sequence 1,2,3; `dl`(*Xs*,[ ]) is the list *Xs*; and `dl`(*Xs*,*Xs*) is the empty list [ ], [Sterling & Shapiro, 1986]. In the predicates we use the difference-lists indicated explicitly by way of the functor *dl* for reasons of clarity. However, the prototype obtained may be further optimised in run-time if the components of the difference-lists are dealt with directly as arguments in the predicates where they are used (it is achieved that predicates are tail-recursive.)

elements from the list Cs_1 as well as new elements that will be added along the way. By definition $Cs_2 = Cs_1 \cup \{ c_i \mid pr^*(c_j, c_i), c_j \in Cs_1\}$. Therefore, a difference list would be a suitable structure for representing this situation.

In fig. 5.18.b, taking the classes of Cs_1 into consideration, if class c_1, included in the initial set Cs_1, is related by means of a property relationship with class c_2 not included in the set Cs_3 (temporary result, initially was the set Cs_1) c_2 must be included in the temporary result Cs_3 (fig. 5.18.b.1). This operation has to be carried out for the rest of the classes with which c_1 is related by means of a property relationship and also, for the new incorporated class c_2 (fig. 5.18.b.2). The property relationships that exist between class c_1 and other classes are consulted in the data dictionary using the predicate `propertyRel`.

```
propertyDecompositionHierarchyClosure(Cs1,Cs2,T) :-                       (a)
      propertyDecompositionHierarchyClosureDL(Cs1,dl(Cs2,Cs1),T).

propertyDecompositionHierarchyClosureDL([C1|Cs1],dl(Cs2,Cs3),T) :-
      propertyRel(C1,C2,_,T),
      not(includedElement(C2,Cs3)),               (2)             (1)
      !,
propertyDecompositionHierarchyClosureDL([C1,C2|Cs1],dl(Cs2,[C2|Cs3]),
            T).
                                                                     (b)
propertyDecompositionHierarchyClosureDL([_|Cs1],Cs2dl,T) :-
      !,
      propertyDecompositionHierarchyClosureDL(Cs1,Cs2dl,T).         (c)

propertyDecompositionHierarchyClosureDL([],dl(Cs,Cs),_) :-
      !.                                                             (d)
```

Figure 5.18. `propertyDecompositionHierarchyClosure` predicate.

If the check for all the classes directly related by means of the property relationship with a given class has already been carried out (fig. 5.18.c) the same operation has to be done for the rest of the classes of Cs_1. When this operation has been carried out for all the classes of Cs_1 -that is to say, no more classes are left in Cs_1 (fig. 5.18.d), the final resulting set will be the temporary resulting set that had been obtained until this moment (the two components of the difference-list used are unified: `dl`(*Cs,Cs*)).

5.4.1.2 *Class inheritance closure*

Given a set of classes (Cs_1), a new set of classes (Cs_2) and a set of edges (Es) are obtained in such a way that Cs_2 and some Es', $Es' \subseteq Es$, form a minimal object schema defined from Cs_1. In other words, the obtained schema is closed in relation to the inheritance relationship: for each pair of classes c_1, c_2 of Cs_2 a class $c_3 = c_1 \sqcap c_2$ is also included in Cs_2. However, Es does not contain inconsistent edges, all the required edges are included in it, nevertheless it may contain redundant edges.

If a pair of classes c_1, c_2 have previously been included in the same schema (conceptual or external), so will a class $c_3 = c_1 \sqcap c_2$. If this is not so, it is possible that a class $c_1 \sqcap c_2$ has not been defined beforehand, and therefore will have to be generated, being a generated derived class.

The predicate corresponding to the definition of this property is presented in fig. 5.19. As with the predicate in fig. 5.18 difference-lists are used, in this case with two sets to be obtained: that of the classes and that of the edges (fig. 5.19.a.1). Therefore a difference-list is used for each one of them. In the case of the class set, the initial class set Cs_1 is taken as the basis to obtain Cs_2 as a result; and in the case of the edge set, the empty set [] is taken and Es is obtained.

```
classInheritanClosure(Cs1,Cs2,Es,T) :-    (2)                 (1)
       classInheritanClosure( Cs1,Cs1 ,dl(Cs2,Cs1),dl(Es,[]),T).        (a)

classInheritanClosure([C|Cs1],[C|Cs2],Csdl,Esdl,T) :-
       !,                                                               (b)
       classInheritanClosure([C|Cs1],Cs2,Csdl,Esdl,T).

classInheritanClosure([C1|Cs1],[C2|Cs2],dl(Cs3,Cs4),dl(Es1,Es2),T) :-
       classPropertiesIntersection(C1,C2,Ps,T),
       !,
       addLowestCommonSuperclass(C1,C2,Ps, dl(Cs6,Cs4), dl(Cs5,Cs1),          (c)
             dl(Es3,Es2),T),    (2)               (1)           (3)
       classInheritanClosure([C1|Cs5],Cs2,dl(Cs3,Cs6),dl(Es1,Es3),T).

classInheritanClosure([_|Cs1],[],dl(Cs2,Cs3),Esdl,T) :-
       !,                                                               (d)
       classInheritanClosure(Cs1,Cs3,dl(Cs2,Cs3),Esdl,T).

classInheritanClosure([],_,dl(Cs,Cs),dl(Es,Es),_) :-
       !.                                                               (e)
```

Figure 5.19. `classInheritanClosure` predicate.

All the possible pairs of classes included in Cs_1 have to be considered. In order to do so, the set of classes Cs_1 is passed twice as the parameter to the auxiliary predicate in fig. 5.19.a.2. A class from each set will be taken into consideration. The following are the different cases that can be found:

- Both classes taken into consideration are the same class c (fig. 5.19.b), $c = c \sqcap c$, already being included as a class and not being related by way of inheritance to itself. Therefore, we must continue the check for the class c and the rest of the classes of the second set.

- c_1 and c_2 are two different classes (fig. 5.19.c). The class $c_1 \sqcap c_2$ must be included in the resulting set of classes (Cs_4 becomes Cs_6 in fig. 5.19.c.1) and the inheritance relationships existing between c_1, c_2 and $c_1 \sqcap c_2$, which are updated in the resulting set of edges -Es_2 becomes Es_3 in fig. 5.19.c.2-, are defined. The class $c_1 \sqcap c_2$ must be compared with the rest of the classes -not in order to generate new classes (since they would be generated anyway in comparing the original classes of Cs_1), but rather in order to determine the inheritance relationships that exist between it and the rest of the classes. Thus, $c_1 \sqcap c_2$ is added to the set of classes to be taken into consideration (Cs_1 becomes Cs_5 in fig. 5.19.c.3). The predicate that carries out these operations is `addLowestCommonSuperclass`, presented in fig. 5.20. Once these operations have been carried out, the same operation must continue to be carried out for the class c_1 and the set of the remaining classes (Cs_2).

- If, for a class of Cs_1, the comparisons with the rest of the selected classes have been done (the list of the remaining classes is the empty list [] in fig. 5.19.d), the same operation of comparison with the rest of the classes of Cs_1 and the current set of resulting classes Cs_3 has to be carried out.

- When this operation has been carried out for all selected and obtained classes (fig. 5.19.e), the resulting sets of classes and edges will be the resulting temporary sets obtained up until this moment (the two components of the difference-lists used are unified: `dl`(*Cs*,*Cs*), `dl`(*Es*,*Es*)).

```
addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
            dl(Es2,Es1),T) :-
      classProperties(C1,Ps,T),
      subsumesExtension(C1,C2,T),                                    (a)
      includeElement(e(is_a(C2,C1)),Es1,Es2),
      !.
```

```
addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
            dl(Es2,Es1),T) :-
      classProperties(C2,Ps,T),
      subsumesExtension(C2,C1,T),                                    (b)
      includeElement(e(is_a(C1,C2)),Es1,Es2),
      !.
```

```
addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
            dl(Es3,Es1),T) :-
      superclassWithPropertiesInSet(C1,C2,Ps,Cs1,C3,T),
      not(lowerSuperclassWithPropertiesInSet(C1,C2,Ps,Cs1,C3,T)),    (c)
      includeElement(e(is_a(C1,C3)),Es1,Es2),
      includeElement(e(is_a(C2,C3)),Es2,Es3),
      !.
```

```
addLowestCommonSuperclass(C1,C2,Ps,dl(Cs2,Cs1),dl(Cs4,Cs3),
            dl(Es3,Es1),T) :-
      derivedClassWithPropertiesInSet(C1,C2,Ps,Cs1,C3,T),
      modifyDerivedClassExtension(C1,C2,C3,T),
      includeElement(C3,Cs1,Cs2),                                    (d)
      includeElement(C3,Cs3,Cs4),
      includeElement(e(is_a(C1,C3)),Es1,Es2),
      includeElement(e(is_a(C2,C3)),Es2,Es3),
      !.
```

```
addLowestCommonSuperclass(C1,C2,Ps,dl(Cs2,Cs1),dl(Cs4,Cs3),
            dl(Es3,Es1),T) :-
      generateDerivedClass(C3,T),
      classProperties(C3,Ps,T),
      classObjectsUnion(C1,C2,C3,T),
      includeElement(C3,Cs1,Cs2),                                    (e)
      includeElement(C3,Cs3,Cs4),
      includeElement(e(is_a(C1,C3)),Es1,Es2),
      includeElement(e(is_a(C2,C3)),Es2,Es3),
      !.
```

Figure 5.20. `addLowestCommonSuperclass` predicate.

Given two classes (c_1, c_2) belonging to the set of classes (Cs_1), the different possible cases in relation to the class $c_1 \sqcap c_2$ are as follows:

- $c_1 \sqcap c_2 = c_1$ (fig. 5.20.a), in which case the fact that c_2 is a subclass of c_1 must be reflected by adding the corresponding edge, *is_a*(c_2,c_1) to the set of edges Es_1.

- $c_1 \sqcap c_2 = c_2$ (fig. 5.20.b) -similar to the previous case, the edge that must be added in this case is *is_a*(c_1,c_2).

- $c_1 \sqcap c_2 = c_3$ -being that c_3 is different from c_1 and c_2:

    - If c_3 is included in the set of classes Cs_1 selected to form the external schema (fig. 5.20.c), it is only necessary to add the inheritance relationships to the edges set, $is\_a(c_1,c_3)$ and $is\_a(c_2,c_3)$.

    - If c_3 is not included in the set of classes Cs_1, but a derived class has already been defined with the same intension (fig. 5.20.d), this class is modified to fulfil the requirements of the extension and the necessary inheritance relationships are added. This way the generation of multiple classes with the same intension is avoided (as shown in section 3.3.2).

    - Lastly, c_3 must be generated in such a way that it fulfils the required conditions (fig. 5.20.e); and must be added to the set of selected classes, just as the inheritance relationships with c_1 and c_2 must be added to the set of edges.

5.4.1.3 Valid object schema

In order to obtain a closed schema with regard to the inheritance relationship all the possible pairs of classes are considered. If an inheritance relationship exists between two classes the corresponding edge is included in the set of edges. The set of edges resulting from the previous process contains all the inheritance relationships that exist between the set of selected classes -all of them are correct. This set will contain redundant relationships since it will contain as direct relationships some ones that may be obtained in an indirect form. The edges corresponding to these relationships are redundant and must be eliminated in order to have a valid schema. The predicate that carries out this operation is `eliminateRedundantEdges` (fig. 5.21). For each edge of the set that is obtained a check is made to see whether there exists an alternative path between their nodes using the rest of the edges in the set. If such a path does exist the edge is, consequently, redundant.

```
eliminateReduntantEdges(Es1,Es2) :-
      eliminateReduntantEdges(Es1,Es1,Es2,[]).

eliminateReduntantEdges([],_,Es,Es) :-
      !.
eliminateReduntantEdges([E|Es1],Es2,Es3,Es4) :-
      indirectEdge(E,Es2),
      !,
      eliminateReduntantEdges(Es1,Es2,Es3,Es4).
eliminateReduntantEdges([E|Es1],Es2,Es3,Es4) :-
      eliminateReduntantEdges(Es1,Es2,Es3,[E|Es4]).
```

Figure 5.21. `eliminateRedundantEdges` predicate.

5.4.2 External schema generation extended algorithm

In the previous section the external schema generation basic algorithm was presented. In this algorithm all the classes, except the ones generated in order to obtain a closed object schema in respect to the inheritance relationship, must be defined explicitly by the application administrator. In section 5.2.4 a mechanism is proposed in order to make available greater power in the external schema definition. Basically, it consists in offering the possibility of qualifying as transformable or non-transformable each one of the selected classes in order to construct the external schema. Such a qualification is carried out in the moment of the selection of the set of classes -that is to say, it is not related to the class alone, but rather to the class within the selection. Non-transformable classes have to be added to the external schema directly (in the basic algorithm, all classes were non-transformable); transformable classes can be replaced by another existing or newly derived class in the external schema if necessary, the class that substitutes it is required to have the same extension (although possibly a different intension, adapted to the structure imposed by the schema's non-transformable classes).

By using such a mechanism the number of classes that needs to be defined explicitly is reduced -it is enough to define the non-transformable classes that determine the structure of the schema and qualify the rest of the classes we consider necessary to be included as transformable.

Given a set of classes (QCs_1), that have been selected from the data dictionary, qualified as transformable or non-transformable, it is the initial definition of the schema S. By way of the extended generation algorithm, we obtain a new set of classes (Cs_2) and a new set of edges (Es_2) so that S = (Cs_2,Es_2) is an object schema which is valid and closed in relation to the inheritance and property relationships.

```
generateQualifiedExternalSchema(S,Cs,Es,T) :-
    qualifiedClassSetSelection(S,QCs1,T),
    includeElement(q(esc(c(objects),nonTransformable)),QCs1,QCs2),
                                                                   (a)
    qualifiedClasses(QCs2,nonTransformable,CsNT1),
    qualifiedClasses(QCs2,transformable,CsT1),

    %Property Decompostion Hierarchy Closure.
    propertyDecompositionHierarchyClosure(CsNT1,CsNT2,T),
    elementsDifference(CsT1,CsNT2,CsT2),                      (1)
    transfClassesRelatedByPropertyRel(CsT2,CsNT2,CsT3,CsNT3,T),
                                                                   (b)
    classesWithProperties(CsT3,CsTWPs1,T),
    transfClassesPropDecompHierarchyClosure(CsTWPs1,CsT3,CsNT3, (2)
          CsTWPs2,T),

    % Class Inheritance Closure.
    classInheritanClosure(CsNT3,CsNT4,EsNT1,T),
    eliminateReduntantEdges(EsNT1,EsNT2,T),            (1)
                                                                   (c)
    subsumptionIsomorficClasses(CsTWPs2,ICsTWPs,CsT4,T),
    integrationOfTansformableClasses(ICsTWPs,CsNT4,EsNT2,
          Cs3,Es2,T),                                  (2)
    eliminateReduntantEdges(Es2,Es3,T),

    eliminateRedundantTransformableClasses(CsNT3,CsT4,Cs3,Es3,    (d)
          Cs,Es,T),
    !.
```

Figure 5.22. `generateQualifiedExternalSchema` predicate.

The predicate that defines the generation algorithm, which is presented in fig. 5.22, has the parts presented in the following points.

5.4.2.1 Obtain the initial sets of transformable and non-transformable classes

In (fig. 5.22.a), the set of qualified classes QCs_1 associated with the schema S is obtained. Then, the class **objects** must be included, being qualified as non-transformable -QCs_2 is obtained-; and on the basis of this set of qualified classes the sets of non-transformable classes $CsNT_1$ and transformable classes CsT_1 are obtained separately.

5.4.2.2 Property decomposition hierarchy closure

Obtain a schema that is closed in relation to the property relationship (fig. 5.22.b). Each property defined in a class depends on a set of properties from other classes or from the same class, these are the properties used in the property definition.

A property can be an attribute or a method (section 3.1.5). Properties are referred to in the implementation of the methods defined in a class. In the definition of an attribute, class names may be used but not properties; the use of a class name in the definition of an attribute in a class does not mean that all the properties of the referenced class are required, the only properties directly used would be the ones referenced in the implementation of the methods of the class.

A property can be redefined in different classes in the class hierarchy, the set of properties directly used by a property p in a class c, denoted $uses(p,c)$, is defined as follows: $uses(p,c) = \{(p_i,c_j) \mid$ property p_i defined in class c_j is directly referenced in the definition of property p in class $c\}$.

In the same way, the set of properties directly or indirectly used in the definition of property p in class c, denoting $uses^*(p,c)$, is defined as follows: $uses^*(p,c) = uses(p,c) \cup \{(p_i,c_j) \mid ((p_i,c_j) \in uses(p_k,c_l)$ or $(p_i,c_j) \in uses^*(p_k,c_l))$ and $(p_k,c_l) \in uses(p,c)\}$

The property decomposition closure criteria (section 3.2.2) can be further refined not considering all the classes referenced in the definition of some class, rather the properties of those classes that are used. Hence, in order to have an external schema closed in relation to the property relationship, for each one of the properties of the classes selected to compose an external schema, all the properties directly or indirectly used in the definition of these properties have to be included in the external schema. If these properties are defined in classes not included in the external schema, two possibilities are considered in reference to the classes:

- The classes should also be included in the external schema.

- The classes can be redefined (define new derived classes) in order to include exclusively the properties needed. Then, the classes previously included in the external schema that referenced the original classes should also be redefined to reference the new derived classes.

In the algorithm presented in this section, the first option was considered because it is simpler than the second one, and it is enough to show the possibilities offered by the external schema mechanism presented. Therefore, in the present version of the extended algorithm of generation, the condition that all classes referenced by means of property relationship by another class must be non-transformable has been imposed. This is a way of avoiding the need, in transforming a class that is referenced by another, to modify the class that refers to it as well.

Therefore, in order to obtain an object schema closed in relation to the property relationship, the following steps have been defined:

- Adding the necessary non-transformable classes (fig. 5.22.b.1): Using the previously defined predicate `propertyDecompositionHierarchyClosure`, we add to the set of non-transformable classes the classes referenced by the non-transformable classes, obtaining $CsNT_2$. The transformable classes referenced by non-transformable classes have become non-transformable classes; and so, by way of the predicate `elementsDifference` the new set of transformable classes CsT_2 is obtained. The selected transformable classes that are referenced by some other transformable class must be non-transformable. `transfClassesRelatedByPropertyRel` carries out this requalification to thus obtain the sets $CsNT_3$ and CsT_3.

- Transforming the transformable classes by eliminating external references (fig. 5.22.b.2): Any transformable class that references classes not selected in order to compose the schema must be transformed, eliminating such references. By means of the predicate `classesWithProperties`, taking the set of transformable classes CsT_3 as its basis, we obtain the set $CsTWPs_1$ of "transformable classes with the set of properties" for each class. The mentioned references are eliminated by the predicate `transfClassesPropDecompHierarchyClosure`, thus obtaining the new set $CsTWPs_2$, which contains the classes of CsT_3 but at the same time maintains only the references to classes of the schema (due to the requalifications carried out in the previous step, only non-transformable classes are referenced).

5.4.2.3 Class inheritance closure

To obtain a schema that is closed in relation to the inheritance relationship (fig. 5.22.c):

- Obtain the schema formed exclusively by non-transformable classes (fig. 5.22.c.1). The initial schema is obtained by way of the predicate `classInheritanClosure` previously defined in the basic algorithm. After eliminating the redundant edges we obtain the new set of non-transformable classes $CsNT_4$ and the set of edges that are defined between them $EsNT_2$.

- Integration of transformable classes (fig. 5.22.c.2): Given the initial schema obtained in the previous step $S_1 = (CsNT_4, EsNT_2)$, the transformable classes of CsT_3 are integrated in it, the detailed process of which is set out in section 5.4.2.4.

- Unification of transformable classes (fig. 5.22.d): once all the transformable classes have been integrated, under certain conditions, some of them may be transformed and unified with auxiliary classes added to the schema exclusively in order to achieve closure in relation to the inheritance relationship or with other transformable classes, such operation being defined by means of the derived predicate `eliminateRedundantTransformableClasses` presented in section 5.4.2.5.

5.4.2.4 Integration of the transformable classes

The determining component of a transformable class is its extension, in view of the fact that the intension may be transformed in order to adapt to the schema. Therefore, two transformable classes with the same extension definition are potentially the same class. It is for this reason that groups of transformable classes are created whose extensions subsume mutually. Each group is represented by a class that has the extension of the classes of the group and the union of the intensions of the represented classes as its intension. In this way, integrating the representative class of the group, all the classes of the group are integrated. The predicate that carries out this operation is `subsumptionIsomorficClasses` (fig. 5.22.c.2). The new set of transformable classes CsT_4 is obtained along with the properties of these classes and the classes of each group in *ICsTWPs* (each element included has the structure `si`(*c*,*Ps*,*Cs*), -that is to say, the class *c*, its intension *Ps* and represented classes *Cs*). The classes of CsT_4 do not subsume in extension mutually.

Given the schema formed by non-transformable classes $S_1 = (CsNT_4, EsNT_2)$, the transformable classes of CsT_4 -associated classes and properties are in *ICsTWPs*- have to be integrated into it. This operation is carried out integrating the transformable classes one at a time in the schema in question. A transformable class integrated in the schema is considered as non-transformable in the process of integration of the remaining classes.

The transformations that can be carried out in a transformable class can be seen in fig. 5.10. Given a transformable class c_1 to be integrated (fig. 5.10.a), if the transformable class is subsumed in extension by another class c_2 belonging to the schema and, therefore, non-transformable (fig. 5.10.b), then the class c_1 inherits the properties of c_2 and, furthermore, maintains its own properties. However, if it is the class c_1 that subsumes in extension another class c_2 of the schema (fig. 5.10.c), the non-transformable class c_2 conditions the structure of the transformable class c_1, in such a way that any properties which are not defined for c_2 must be eliminated from c_1.

```
transformableClassIntegration(si(C1,Ps1,Cs1),Cs2,Es2,
          Cs3,Es3,T) :-                                    (a)
     includedElement(C2,Cs2),
     subsumesExtension(C1,C2,T),                (1)
     !,
     branchTransfClassInteg(si(C1,Ps1,Cs1),
               Cs2,Es2,Cs3,Es3,T).
```

```
transformableClassIntegration(si(C1,Ps1,Cs1),Cs2,Es2,
          Cs3,Es3,T) :-
     !,
     leafTransfClassInteg(si(C1,Ps1,Cs1),                  (b)
               Cs2,Es2,Cs3,Es3,T).
```

Figure 5.23. `transformableClassIntegration` predicate.

When a transformable class is integrated in the schema, it becomes considered non-transformable. So that a previously transformable class already integrated (c_2) has no effect in the integration of another transformable class (c_1) causing it to lose properties (in the way shown in fig. 5.10.c). In the first place it is necessary to integrate the transformable classes that are not subsumed in extension by other transformable classes (CsT_4 is the set of transformable classes and these do not subsume in extension mutually). The order of integration of the transformable classes is defined in this way in the predicate `integrationOfTansformableClasses` (used in fig. 5.22.c.2) which proceeds by selecting transformable classes which are not subsumed by any of the remaining transformable classes and carries out the integration by way of `transformableClassIntegration` predicate, presented in fig. 5.23.

5.4.2.4.1 Transformable class with subclasses in the schema

If the transformable class to be integrated (c_1) subsumes in extension any of the classes (c_2) currently integrated in the schema (fig. 5.10.d), as would be the case in fig. 5.23.a where the condition denoted by (1) is fulfilled, the predicate which defines the manner of integration in this case is `branchTransfClassInteg`. This predicate, for each one of the classes (c_2) of the schema subsumed in extension by the transformable class to be integrated (c_1), carries out the integration operations defined by the predicate `integrateTransformableClass` set out in fig. 5.24.

```
integrateTransformableClass(si(C1,Ps1,Cs1),C2,Cs2,Es2,Cs3,Es3,T) :-
         subsumingSuperclassPropertyUnion(C1,C2,Cs2,Es2,Ps3,T),
         elementsUnion(Ps1,Ps3,Ps4),                                  (a)

         classProperties(C2,Ps2,T),
         elementsIntersection(Ps2,Ps4,Ps5),                           (b)

         defineTransformableClass(si(C1,Ps5,Cs1),C3,T),
         includeElement(C3,Cs2,Cs4),
         classInheritanClosure([C3],Cs2,dl(Cs3,Cs4),dl(Es3,Es2),T),   (c)
         !.
```

Figure 5.24. `integrateTransformableClass` predicate.

A transformable class (c_1) has a set of properties (Ps_1) associated with it and represents the set of transformable classes with identical extension (Cs_1), (represented by si(c_1,Ps_1,Cs_1) in fig. 5.24). c_1 subsumes in extension the class c_2, which is included in the schema S_2 = (Cs_2, Es_2). The predicate `integrateTransformableClass` defines the integration of the class c_1 into the schema S_2 to generate a new schema S_3 = (Cs_3, Es_3). The class c_1 will be a superclass of c_2 and a subclass of those superclasses of c_2 that subsume in extension c_1. The intension of c_1 will be conditioned by the intensions of the superclasses that are obtained just as it will by the intension of c_2.

In fig. 5.24.a, we obtain in Ps_3 the properties of the superclasses of c_2 that subsume c_1. Such properties must be inherited by c_1: united to the properties Ps_1 that c_1 had, Ps_4 is obtained. The intension of c_2 is Ps_2 (Ps_2 contains the properties of the set Ps_3 since they are properties of the superclasses of c_2, inherited by c_2). Therefore, the set of properties which the transformable class c_1 has, having been transformed previously, is Ps_5, the intersection of Ps_2 and Ps_4 (fig. 5.24.b). This would be the case in the example shown in fig. 5.10.d where c_1 is affected by transformations imposed by the superclasses found, just as it is by the subclass c_2. Once we have the properties of the transformable class, we proceed with its definition and inclusion in the schema (fig. 5.24.c). The definition is carried out by way of the predicate `defineTransformableClass` which is presented in fig. 5.25. Given the transformable class c_1 and its characteristics, it returns the transformed class c_3. The integration in the schema of the transformed class is carried out by means of the auxiliary part of `classInheritanClosure` set out in fig. 5.24 which finds the inheritance relationships of the class c_3 that is to be integrated with the rest of the existing classes in the schema $S_2 = (Cs_2,Es_2)$, to obtain a schema $S_3 = (Cs_3,Es_3)$.

```
defineTransformableClass(si(C1,Ps1,Cs1),C2,T) :-
      classProperties(C2,Ps1,T),
      subsumesExtension(C2,C1,T),          (a)
      subsumesExtension(C1,C2,T),
      !,
      associateClassesByDerivation(C2,Cs1,T).

defineTransformableClass(si(C1,Ps1,Cs1),C2,T) :-
      generateDerivedClass(C2,T),
      classProperties(C2,Ps1,T),           (b)          (c)
      classObjects(C1,Os1,T),
      classObjects(C2,Os1,T),
      !,
      associateClassesByDerivation(C2,Cs1,T).
```

Figure 5.25. `defineTransformableClass` predicate

By way of the predicate `defineTransformableClass` (fig. 5.25) we obtain a transformed class from the transformable class. If a class with the required characteristics already existed in the data dictionary (fig. 5.25.a) this will be the obtained class. Otherwise (fig. 5.25.b) it defines the new class in the data dictionary. Once it has the class (existing or generated), it adds to the data dictionary the fact that a derivation relationship exists between this class and the set of transformable classes (Cs_1) that are represented by c_1 (fig. 5.25.c).

This process of transformation of a transformable class c_1 is carried out for each class of the schema that is subsumed by c_1. In the schema, the only classes subsumed by c_1 will be the set of non-transformable classes that made up the initial schema; this is due to the order of integration of the transformable classes that is followed.

5.4.2.4.2 Transformable class without subclasses in the schema

If the class c_1 does not subsume any class of the schema (fig. 5.25.b), the integration would be brought about by way of `leafTransfClassInteg` which is set out in fig. 5.26.

```
leafTransfClassInteg(si(C1,Ps1,Cs1),Cs2,Es2,Cs3,Es3,T) :-
      subsumingClassPropertyUnion(C1,Cs2,Ps2,T),
      elementsUnion(Ps1,Ps2,Ps3),                                (a)

      defineTransformableClass(si(C1,Ps3,Cs1),C2,T),
      includeElement(C3,Cs2,Cs4),                                (b)
      classInheritanClosure([C2],Cs2,dl(Cs3,Cs4),dl(Es3,Es2),T),
      !.
```

Figure 5.26. `leafTransfClassInteg` predicate.

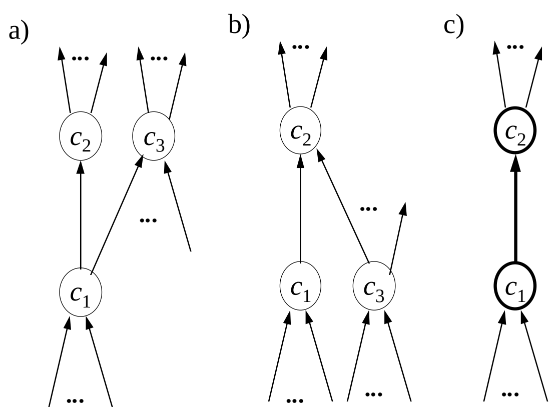
Corresponding to the example in fig. 5.10.e, the transformation is carried out by adding to the transformable class (c_1) the properties of the classes that subsume it in extension (fig. 5.26.a), being defined and integrated into the schema afterwards (fig. 5.26.b).

With the defined integration process indirect inheritance relationships are represented as edges, the elimination of such redundant edges being defined in the predicate `eliminateReduntantEdges` in fig. 5.21.

5.4.2.5 Unification of transformable classes

Once we have all the transformable classes integrated, under certain conditions, some of these classes may be transformed and unified with auxiliary classes that have been added to the schema exclusively in order to achieve the closure in relation to the inheritance relationship and even with other transformable classes. Before describing the unification process, we shall consider the conditions that must be satisfied.

An edge $e = is\_a(c_1,c_2)$ has *exclusive nodes* in a set of edges *Es,* if it is the only edge included in *Es* that has the class c_1 as its first node and also is the only one that has the class c_2 as its second node -in other words, the edge *e* is the only one that arrives at c_2 and is also the only one that starts from c_1.



```
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
        includedElement(e(is_a(C1,C2)),Es),
        includedElement(e(is_a(C1,C3)),Es),
        compareClasses(C3,C2,ne),                (a)
        !,
        fail.
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
        includedElement(e(is_a(C1,C2)),Es),
        includedElement(e(is_a(C3,C2)),Es),
        compareClasses(C3,C1,ne),                (b)
        !,
        fail.
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
        includedElement(e(is_a(C1,C2)),Es),      (c)
        !.
```

Figure 5.27. Exclusive nodes edge.

In fig. 5.27 the predicate that determines whether an edge is of exclusive nodes is presented together with its graphic representation. Only in the case represented in fig. 5.27.c, does the indicated edge satisfy the described requirements.

Should c_1 be a class that has been added in order to achieve the closure of the schema in relation to the inheritance relationship (a class included in the set *CsAdded* in fig. 5.28), should c_2 be an originally transformable or an already transformed class (included in *CsTnew*), and should an edge of exclusive nodes exist between them: then, the predicate `eliminateRedundantTransformableClasses`, presented in fig. 5.28, joins both classes in a new transformed class, updating the schema as may be appropriate and repeats this operation for each existing case. This predicate is made up of two parts: primarily, fig. 5.28.a, the various sets of classes (*CsTnew* and *CsAdded*) are determined and afterwards, fig. 5.28.b, the added classes are analysed by ascertaining whether any edge of exclusive nodes exists with the described conditions and carrying out the unification.

```
eliminateRedundantTransformableClasses(CsNT,CsT,Cs1,Es1,
            Cs2,Es2,T) :-
      elementsDifference(Cs1,CsNT,Cs_notNT),
      elementsDifference(Cs_notNT,CsT,Cs_notNTnotT),
      obtainedFromTransformableClasses(Cs_notNTnotT,CsT,CsTrans,T),
      elementsUnion(CsT,CsTrans,CsTnew),
      elementsDifference(Cs_notNTnotT,CsTrans,CsAdded),
      eliminateRedundantTransformableClasses(CsAdded,CsTnew,
                  dl(Cs2,Cs1),dl(Es2,Es1),T).
```

(a)

```
eliminateRedundantTransformableClasses([C1|CsAdded],CsTnew,
            dl(Cs2,Cs1),dl(Es2,Es1),T) :-
      exclusiveNodesEdge(e(is_a(C1,C2)),Es1),
      includedElement(C2,CsTnew),
      !,
      unifyTransformableAndAddedClasses(C2,C1,C3,T),
      unifyClassesInSchema(C2,C1,C3,Cs1,Es1,Cs3,Es3,T),
      eliminateRedundantTransformableClasses(CsAdded,CsTnew,
                  dl(Cs2,Cs3),dl(Es2,Es3),T).
eliminateRedundantTransformableClasses([_|CsAdded],CsTnew,
            Csdl,Esdl,T) :-
      !,
      eliminateRedundantTransformableClasses(CsAdded,CsTnew,
                  Csdl,Esdl,T).
eliminateRedundantTransformableClasses([],_,dl(Cs,Cs),
            dl(Es,Es),_) :-
      !.
```

(b)

Figure 5.28. `eliminateRedundantTransformableClasses` predicate.

5.5 Conclusions

Independently of the external schema generation algorithm considered, the main characteristics of this proposal are:

- Two phase integration: First, derived classes are integrated directly into the data dictionary by derivation relationships. Then, they are integrated into the external schema, but this time considering only the set of selected classes and using inheritance relationships.

- The new concepts of transformable and non-transformable classes simplify the external schema definition process, because they permit the direct generation of the classes needed in the external schema, avoiding more complex definitions.

- This methodology respects the ANSI/SPARC three-level schema architecture.

- According to the classification of external schema definition methodologies presented in chapter 2, the new methodology belongs to the second group: defined external schemas are not necessarily subschemas of the conceptual schema.

- In the examples presented, derived classes were always defined according to object-preserving semantics. This is not a limitation of the new methodology: derived classes defined according to object-generating semantics can also be handled in the same way.

- With the definition of this methodology, in particular using the derivation relationship to integrate derived classes, the object-orientation paradigm is not changed. The derivation relationship does not appear in object schemas for end users, it is only used in the data dictionary.

In [Santos, 1995] it is stated that: “By relating virtual (derived) and corresponding root (base) classes through the *may_be* relationship, we avoid the creation of auxiliary intermediate classes which have to be generated to allow the integration of virtual classes into the inheritance hierarchy as described in [Rundensteiner, 1992b]. From a philosophical point of view, integrating virtual classes into a single inheritance hierarchy amounts to define a single taxonomy which encompasses the whole set of concepts (in the spirit of the KL-ONE classification algorithm), whereas relating such classes to the inheritance hierarchy through an orthogonal relation gives these classes a different conceptual status, making them not concepts themselves, but different points of views (possibly exceptional and therefore involving type incompatibility) of existing concepts instead.”

In the external schema definition methodology proposed here, using the derivation relationship, derived classes have this “different conceptual status” but only in the data dictionary and not in end-user’s schemas.

6 Definition of derived classes

In the definition of derived classes three main issues have to be resolved such as:

- The integration of derived classes with other classes in an object schema.
- The definition of classes with new objects.
- The transmission of modifications between the objects in derived classes and the objects from which they have been defined.

A solution to the first issue has been proposed in chapter 5. In this chapter, the other two remaining issues are further studied.

6.1 Derived classes

Non-derived classes are defined during the initial definition of the conceptual schema. Derived classes are classes which are defined from previously existing classes (derived or non-derived) using object-oriented queries; derived classes are defined during the lifetime of the database in order to be included in some external schema or in the conceptual schema.

In order to adapt to final users' needs, the information contained in the conceptual schema's classes must be re-organised in the form of new classes: external schemas may contain conceptual schema classes as well as new derived classes.

6.1.1 Base classes and base objects

The classes from which a derived class is directly defined are its base classes. Derived classes offer a new interface of access to the information contained in their base classes; derived classes share the data stored in the database with their base classes. In no case may a derived class contain information that has not been obtained from its base classes.

The objects contained in a derived class are *derived objects*. The objects in base classes that participate in the definition of a derived object are its *base objects*.

6.1.2 Object-preserving vs. object-generating semantics

A derived class may be defined either by object-preserving semantics, if it only contains objects of its base classes; or by object-generating semantics, if it contains new objects generated from the objects of its base classes. Defining derived classes by object-generating semantics makes it possible to carry out sophisticated reorganisations of existing information which would otherwise be impossible -i.e. transformation of values into objects, or aggregation of objects to form a new concept.

If a derived class represents a concept previously defined in object form, it will have to be defined by object-preserving semantics: the derived class defines a new interface for its objects. If a derived class represents a concept not previously defined in object form, it will have to be defined by object-generating semantics.

We consider it necessary to be able to define derived classes using both semantics since, in some situations, it is necessary to carry out reorganisations of the information that could not be accomplished defining derived classes exclusively by object-preserving semantics; besides, if the derived class defines a new interface over a concept previously defined in object form, it will have to be defined by object-preserving semantics in order to keep this information.

6.1.3 The derivation relationship

A *derivation relationship* is defined between a derived class and the set of its base classes. The derivation relationship defines how to obtain a derived class from its base classes; it establishes the correspondence between the base objects and the derived objects.

The derivation relationship is used to integrate the derived classes into the data dictionary. Derived classes are related by means of the derivation relationship to its base classes. This relationship is different from the inheritance relationship and aggregation relationship found in the object orientation paradigm and they lie on an orthogonal dimension: the derivation dimension, part of the point of view dimension in terms of the ANSI/SPARC framework [ANSI/X3/SPARC, 1986].

Unlike other authors who define similar relationships ("view derivation" [Bertino, 1992], "may_be" [Santos *et al.*, 1994], "derived-from" [Kim & Kelley, 1995]), this derivation relationship does not appear in either the conceptual schema or the external schemas, only in the data dictionary; and it is not necessary to extend the object orientation paradigm in order to include it.

6.1.4 An example

Let us take as an example the classes laid out in fig. 6.1, based on the examples used in [Abiteboul & Bonner, 1991; Heuer & Sander, 1991]. Initially, we have the non-derived class PEOPLE, this class containing the property Hobbies which returns a set of names of hobbies for each object. From the class PEOPLE the derived classes HOBBIES and HOBBIES' are defined by object-generating semantics, such classes representing the existing hobbies in object form according to different criteria, as we shall see in the next section.

The class PEOPLE' has been defined by object-preserving semantics. This class represents the same concept represented by the initial class PEOPLE but is adapted to the new representation provided by the derived class HOBBIES with which it is related by way of an aggregation relationship. The class MATCHES has been defined by object-generating semantics, and represents the different matches that can be defined between objects of the class PEOPLE' with which certain hobbies are associated -that is to say, it represents a new concept defined on the basis of existing information.

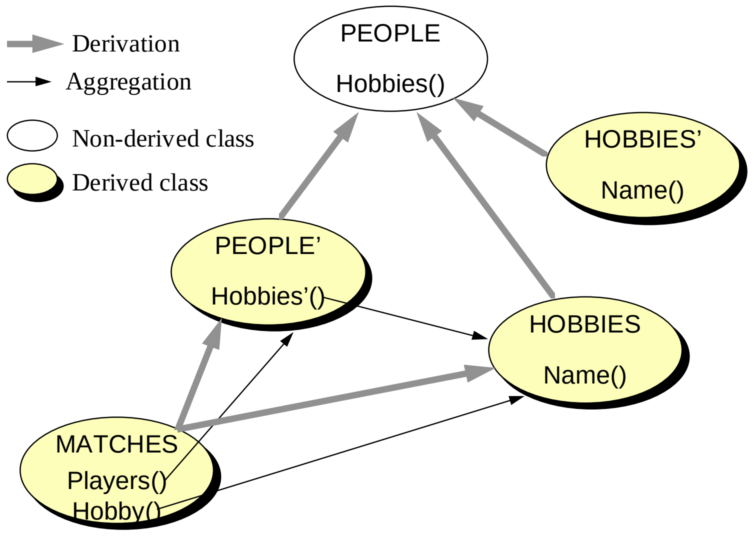


Figure 6.1. Definition of derived classes.

Therefore, this example shows the definition of classes by object-preserving semantics, in the case of class PEOPLE', as well as the definition of classes by object-generating semantics: transformation of values into objects in the definition of the classes HOBBIES or HOBBIES', and aggregation of objects in order to form a new concept in the case of the class MATCHES.

6.2 Derived class object identifiers

Each base object or derived object is represented by its identifier. "An object identifier has no intrinsic meaning -and derives its meaning *only* from its relationship to values or other object identifiers in a given database instance. In particular, then, if an object identifier is considered independently from its associated database instance, then it conveys essentially no information other than its identity as distinct from all other object identifiers" [Hull et *al.*, 1991]. The relationships of the object identifier with values or other object identifiers are defined by the properties that are applicable to the object.

PEOPLE			
oid	**...**	**Age**	**Hobbies**
p_1		31	{Tennis, Football, Driving}
p_2		16	{Reading, Chess}
p_3		18	{Chess, Tennis}
p_4		27	{Chess, Tennis}

Figure 6.2. Non-derived class PEOPLE.

Let us consider the example set out in fig. 6.1. In fig. 6.2 the elements of the non-derived class PEOPLE that are relevant to the example we are using are represented in table form. Every object is represented by one row in the table, and for each one we have its identifier and the properties Age and Hobbies. The use of tables in the various examples is only a form of representation.

6.2.1 Object-preserving semantics

In the example of fig. 6.1 the only class defined by object-preserving semantics is the class PEOPLE'. As shown in fig. 6.3, the object identifier of each object of the new class PEOPLE' is the same as the identifier of its corresponding base object of the non-derived class PEOPLE of fig. 6.2.

PEOPLE'		
oid	**Base Objects**	**...**
p_1	p_1	
p_2	p_2	
p_3	p_3	
p_4	p_4	

Figure 6.3. Representation of the objects of the class PEOPLE'.

The objects of a class defined by object-preserving semantics are objects already existing in its base classes.

6.2.2 Object-generating semantics

As presented in chapter 2, section 2.3.4, some systems generate new object identifiers defined as a combination of values and object identifiers of other objects. It can be considered that the object identifier of a derived object is generated from a set of its attributes; in [Abiteboul & Bonner, 1991] these attributes are called core attributes. This is also our point of view.

The core attributes do not necessarily form part of the interface of the derived class -they may be internal properties. In the case that a derived class is defined by object-preserving semantics, for each object of the derived class it may be considered that there exists an internal core attribute that returns the identifier of its base object.

There will never be two objects in a derived class having identical values in their respective core attributes. Therefore, the objects of a derived class may be represented either by way of their identifier or by way of their core attributes: there is a bijective relationship between object identifiers and core attributes for each derived class.

A class is defined to be *value-identifiable* if its objects can be identified using a set of its attributes that only can be values (not object identifiers) [Schewe et *al.*, 1992]. In the same way, we define a class as *attribute-identifiable* if its objects can be identified using a set of its attributes (without type restriction). Derived classes are attribute-identifiable: their objects can also be identified by their core attributes.

Therefore, the representation of the classes shown in the example of fig. 6.1 is the following: from the class PEOPLE the class HOBBIES is defined in such a manner that an object is generated for each different value in the set returned by the Hobbies property for any object, carrying out a transformation of values into objects. In fig. 6.4 a representation of the result obtained can be seen.

HOBBIES				
oid	**Core Attributes**	**Base Objects**	**...**	**Name**
h_{11}	Tennis	$\{p_1, p_3, p_4\}$		Tennis
h_{12}	Football	$\{p_1\}$		Football
h_{13}	Driving	$\{p_1\}$		Driving
h_{14}	Reading	$\{p_2\}$		Reading
h_{15}	Chess	$\{p_2, p_3, p_4\}$		Chess

Figure 6.4. Representation of the objects of the class HOBBIES.

If two objects of the class PEOPLE have some value in common in the set returned by the Hobbies property, both of them will be base objects of the corresponding derived object (generated from that value). In general, all the base objects from which the same values for the core attributes are obtained are included in the set of base objects for the corresponding derived object that is generated.

In fig. 6.5 the representation of the result obtained for the class HOBBIES' is shown. The identifier of the base object and the hobby name have been defined as core attributes. In this way a new object is generated for each value in the set returned by the property Hobbies in each one of the objects of the class PEOPLE.

HOBBIES’				
oid	**Core Attributes**	**Base Objects**	**...**	**Name**
h_{21}	p_1, Tennis	p_1		Tennis
h_{22}	p_1, Football	p_1		Football
h_{23}	p_1, Driving	p_1		Driving
h_{24}	p_2, Reading	p_2		Reading
h_{25}	p_2, Chess	p_2		Chess
h_{26}	p_3, Chess	p_3		Chess
h_{27}	p_3, Tennis	p_3		Tennis
h_{28}	p_4, Chess	p_4		Chess
h_{29}	p_4, Tennis	p_4		Tennis

Figure 6.5. Representation of the objects of the class HOBBIES'.

The class PEOPLE' represented in fig. 6.6 has been defined by object-preserving semantics, the only core attribute being the base object identifier.

PEOPLE’				
oid	**Core Attributes**	**Base Objects**	**...**	**Hobbies’**
p_1	p_1	p_1		$\{h_{11}, h_{12}, h_{13}\}$
p_2	p_2	p_2		$\{h_{14}, h_{15}\}$
p_3	p_3	p_3		$\{h_{15}, h_{11}\}$
p_4	p_4	p_4		$\{h_{15}, h_{11}\}$

Figure 6.6. Representation of the objects of the class PEOPLE’.

The class MATCHES in fig. 6.7, has been defined by object-generating semantics by object association. It corresponds to the possible Tennis or Chess matches that may be defined between two different objects of PEOPLE’ with those hobbies; the order of the players is irrelevant.

MATCHES					
oid	**Core Attributes**	**Base Objects**	**...**	**Hobby**	**Players**
m_1	$h_{11}, \{p_1, p_3\}$	$h_{11}, \{p_1, p_3\}$		h_{11}	$\{p_1, p_3\}$
m_2	$h_{11}, \{p_1, p_4\}$	$h_{11}, \{p_1, p_4\}$		h_{11}	$\{p_1, p_4\}$
m_3	$h_{11}, \{p_3, p_4\}$	$h_{11}, \{p_3, p_4\}$		h_{11}	$\{p_3, p_4\}$
m_4	$h_{15}, \{p_2, p_3\}$	$h_{15}, \{p_2, p_3\}$		h_{15}	$\{p_2, p_3\}$
m_5	$h_{15}, \{p_2, p_4\}$	$h_{15}, \{p_2, p_4\}$		h_{15}	$\{p_2, p_4\}$
m_6	$h_{15}, \{p_3, p_4\}$	$h_{15}, \{p_3, p_4\}$		h_{15}	$\{p_3, p_4\}$

Figure 6.7. Representation of the objects of the class MATCHES.

6.2.3 Classes containing objects already generated

In previous systems, the most commonly used ways of sharing objects between derived classes were the same as the ways of sharing objects between non-derived classes or derived and non-derived classes, namely:

- Inheritance: defining a derived class as a subclass of a previously existing derived class.

- Derivation: defining a new class with object-preserving semantics from another derived class.

Usually, in order to generate new identifiers, Skolem functors are used [Hull et *al.*, 1991]; in general, a distinct Skolem functor is used for each new derived class. Given the values of the set of core attributes, the functor generates a new object identifier: the identifiers of new generated objects are function of the core attributes and the derived class; therefore, derived classes defined by object-generating semantics can not share objects with previously existing classes.

We propose to define the identifiers of the new objects only as function of their respective core attributes. Thus, two different derived classes defined by object-generating semantics can have objects in common without being related by inheritance or derivation: having the same set of core attributes. Independently of their respective names, attributes are uniquely identified in the data dictionary.

As mentioned in chapter 3, section 3.1.5, for simplicity, we assume that all properties in the data dictionary have unique property names. To ensure uniqueness of properties, a unique property identifier can be associated to each newly defined property; therefore, two properties that have the same property name could thus be distinguished internally based on their identifier.

Therefore, the generation of identifiers of objects in a derived class is not dependent upon the derived class itself, it only depends on the core attributes selected.

For example, if a new derived class HOBBIES'' representing the hobbies that are related with people older than 20 is defined by object-generating semantics (its base class is the class PEOPLE), it will contain a subset of the objects of class HOBBIES (fig. 6.4) independently of which class have been defined before (HOBBIES'' or HOBBIES), as shown in fig. 6.8.

HOBBIES''				
oid	**Core Attributes**	**Base Objects**	**...**	**Name**
h_{11}	Tennis	$\{p_1, p_4\}$		Tennis
h_{12}	Football	$\{p_1\}$		Football
h_{13}	Driving	$\{p_1\}$		Driving
h_{15}	Chess	$\{p_4\}$		Chess

Figure 6.8. Representation of the objects of the class HOBBIES''.

6.3 Definition of the objects in derived classes

6.3.1 Derived classes definition predicates

In order to define the derived classes, derived predicates can be defined in Prolog as part of the corresponding DCM of an external schema definition system, as it is shown in this section.

The definition of the derived classes has been carried out by means of the predicate `derivedClass`; in this predicate `hobbies` corresponds to class HOBBIES, `hobbies_` to class HOBBIES', etc. The creation of new objects is controlled by the predicate `newObject`.

The definition of the class HOBBIES can be seen in fig. 6.9. For each object of the class PEOPLE its set of hobbies is obtained (property hobbies); and for each one of the names of its hobbies a new object is generated by the predicate `newObject` (if not generated before). In order to generate a new object it has to be given the derived class name (i.e. `hobbies`), the names -identifiers- of the core attributes that define de derived class (i.e. `coreAttNames([prop(hobbyName)])`), the values of the core attributes defined (i.e.

coreAttValues([value(Name)])) and the set of base objects that take part in the definition of the derived object (i.e. baseObjects([object(P)])).

```
derivedClass(hobbies,T) :-
      classObject(people,P,T),
      objectProperty(P,hobbies,set(Hs),T),
      includedElement(value(Name),Hs),
      newObject(hobbies,
            coreAttNames([prop(hobbyName)]),
            coreAttValues([value(Name)]),
            baseObjects([object(P)]),T),
      fail.
derivedClass(hobbies,_) :-
      !.
```

Figure 6.9. Definition of objects of class HOBBIES.

The definition of the class HOBBIES' can be seen in fig. 6.10. In this case, two core attributes have been defined: the hobby name, and the object of class PEOPLE where this hobby is defined (i.e. coreAttNames([prop(hobbyPer),prop(hobbyName)])).

```
derivedClass(hobbies_,T) :-
      classObject(people,P,T),
      objectProperty(P,hobbies,set(Hs),T),
      includedElement(value(Name),Hs),
      newObject(hobbies_,
            coreAttNames([prop(hobbyPer),prop(hobbyName)]),
            coreAttValues([object(P),value(Name)]),
            baseObjects([object(P)]),T),
      fail.
derivedClass(hobbies_,_) :-
      !.
```

Figure 6.10. Definition of objects of class HOBBIES'.

The class HOBBIES'' represented in fig. 6.8 is defined in fig. 6.11. As can be seen there, with regard to the core attributes used, the definition of this class is the same as the definition of class HOBBIES in fig. 6.10. Thus, the same object identifiers as before are generated.

```
derivedClass(hobbies__,T) :-
      classObject(people,P,T),
      objectProperty(P,age,value(Age),T),
      Age > 20,
      objectProperty(P,hobbies,set(Hs),T),
      includedElement(value(Name),Hs),
      newObject(hobbies,
            coreAttNames([prop(hobbyName)]),
            coreAttValues([value(Name)]),
            baseObjects([object(P)]),T),
      fail.
derivedClass(hobbies__,_) :-
      !.
```

Figure 6.11. Definition of objects of class HOBBIES''.

The class PEOPLE' is defined by object-preserving semantics. Its definition can be seen in fig. 6.12. In this case, the only core attribute defined is the associated object of its base class PEOPLE; therefore, the same object identifier is used for the corresponding object of the new class.

```
derivedClass(people_,T) :-
     classObject(people,P,T),
     newObject(people_,
          coreAttNames([prop(person)]),
          coreAttValues([object(P)]),
          baseObjects([object(P)]),T),
     fail.
derivedClass(people_,_) :-
     !.
```

Figure 6.12. Definition of objects of class PEOPLE'.

The definition of class MATCHES can be seen in fig. 6.13.

```
derivedClass(matches,T) :-
     classObject(hobbies,H,T),
     objectProperty(H,name,value(Name),T),
     includedElement(value(Name),
          [value(tennis),value(chess)]),
     classObject(people_,P1,T),
     objectProperty(P1,hobbies_,set(Hs1),T),
     includedElement(object(H),Hs1),
     classObject(people,P2,T),
     P1 <> P2,
     objectProperty(P2,hobbies_,set(Hs2),T),
     includedElement(object(H),Hs2),
     newObject(matches,
          coreAttNames([prop(hobby),prop(players)]),
          coreAttValues([object(H),
               set([object(P1),object(P2)])]),
          baseObjects([object(H),
               set([object(P1),object(P2)])]),T),
     fail.
derivedClass(matches,_) :-
     !.
```

Figure 6.13. Definition of objects of class MATCHES.

6.3.2 Definition of properties of derived objects

The properties of the derived objects are defined from the base objects and the core attributes using the predicate `objectProperty`. The base objects and core attributes are accessible through predicates from the identifier of the derived object. In fig. 6.14, some examples of object property definition for classes HOBBIES (*i*) and MATCHES (*ii*, *iii*) can be seen.

```
i)   objectProperty(H,hobbyName,N,T) :-
          classObject(hobbies,H,T),
          objectCoreAttributes(H,[N],T).

ii)  objectProperty(M,hobby,H,T) :-
          classObject(matches,M,T),
          objectBaseObjects(M,[H|_],T).

iii) objectProperty(M,players,Ps,T) :-
          classObject(matches,M,T),
          objectBaseObjects(M,[_,Ps],T).
```

Figure 6.14. Definition of properties of derived objects.

6.3.3 Kinds of derivation relationships

The base objects of a derived object are all those objects upon which it is defined. The set of objects that participate in the definition of the identity of a derived object is a subset of its set of base objects. An example of this is given in figs. 6.15 (representation) and 6.16 (definition): the class PEOPLE'' is defined with the property Hobbies' defined as was made before for the class PEOPLE', and also the property OtherHobbies, which relates each person with the names of hobbies which the person in question does not practise but are practised by other people who have some hobby in common with that person. This class is defined by object-preserving semantics from the class PEOPLE', its only core attribute being the identifier of the object of the base class from which it receives its identity.

PEOPLE''					
oid	**Core Attributes**	**Base Objects**	**...**	**Hobbies'**	**OtherHobbies**
p_1	p_1	p_1, $\{p_3, p_4\}$, $\{h_{15}\}$		$\{h_{11}, h_{12}, h_{13}\}$	{Chess}
p_2	p_2	p_2, $\{p_3, p_4\}$, $\{h_{11}\}$		$\{h_{14}, h_{15}\}$	{Tennis}
p_3	p_3	p_3, $\{p_1, p_2\}$, $\{h_{12}, h_{13}, h_{14}\}$		$\{h_{15}, h_{11}\}$	{Footbal,Driving, Reading}
p_4	p_4	p_4, $\{p_1, p_2\}$, $\{h_{12}, h_{13}, h_{14}\}$		$\{h_{15}, h_{11}\}$	{Footbal,Driving, Reading}

Figure 6.15. More than one base object by object-preserving semantics.

For the definition of the class PEOPLE'', each derived object is related to the object from which it receives its identity and also to the objects that participate in the definition of its properties.

```
derivedClass(people__,T) :-
      classObject(people_,P,T),
      newObject(people__,coreAttributes([object(P)]),
            baseObjects([object(P)]),T),
      fail.
derivedClass(people__,T) :-
      classObject(people_,P1,T),
      objectProperty(P1,hobbies_,set(Hs1),T),
      classObject(people_,P2,T),
      P1 <> P2,
      objectProperty(P2,hobbies_,set(Hs2),T),
      intersection(Hs1,Hs2,[_|_]),
      difference(Hs2,Hs1,Hs3),
      newObject(people__,
            coreAttNames([prop(person)]),
            coreAttValues([object(P1)]),
            baseObjects([object(P1),set([object(P2)]),
                  set(Hs3)]),T),
      fail.
derivedClass(people__,_) :-
      !.
```

Figure 6.16. Definition of objects of class PEOPLE''.

The derivation relationship establishes the correspondence between the base objects and the objects of the derived class. Some of the base objects participate in the formation of the identity of the derived objects: the values as well as the identifiers of the objects returned by the properties that form the core attributes are obtained from a subset of the base objects. Some base objects are only used in order to define properties of the derived objects. Therefore, two kinds of derivation relationship exist: derivation of identity and derivation of value. A base class is related through a *derivation relationship of identity* to a derived class if the objects of the base class participate in the definition of the identity of the objects of the derived class, a *derivation relationship of value* existing between a base class and a derived class only if the objects of the base class do not participate in the definition of the identity of the derived objects.

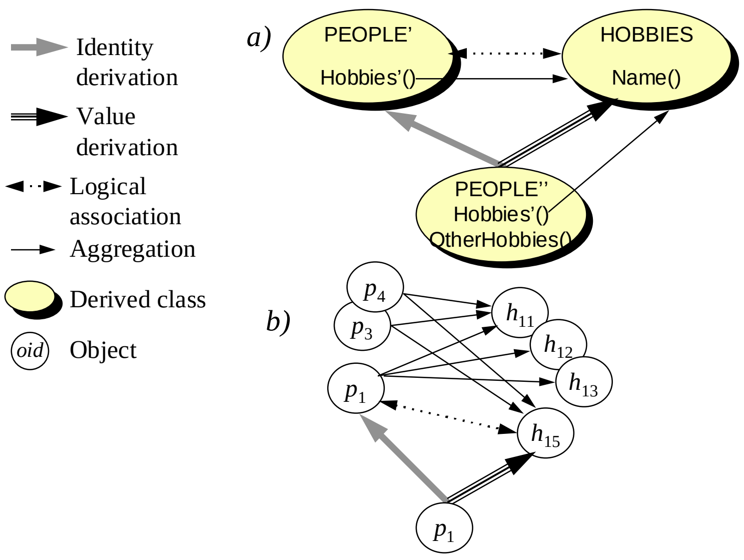


Figure 6.17. Kinds of derivation relationship.

An example of these two kinds of relationship is shown in fig. 6.17 for the definition of the class PEOPLE'' represented in fig. 6.15. In fig. 6.17.a these relationships are depicted on a class level, and in fig. 6.17.b on occurrence level for a derived object.

In addition to the derivation relationships mentioned, in fig. 6.17 there is an explicit representation of the relationship expressed by way of the conditions defined in order to associate an object of the class PEOPLE' with the objects of the class HOBBIES upon whose basis the property OtherHobbies is defined -distinct from those already related through the aggregation relationship. This relationship is termed *logical association*.

6.4 Transmission of modifications

6.4.1 Dynamic derivation relationship

Adapting the definitions of "static view" and "dynamic view" made by Gottlob, Paolini & Zicari [Gottlob et *al*., 1988], a *static derivation relationship* is defined between the set of base classes and a derived class, and establishes the correspondence between the base objects and the derived objects. The derived class defined by a static derivation relationship is a *static derived class*.

A *dynamic derivation relationship* is made up of a static derivation relationship and a *translator* or *update policy* that determines how to transmit the modifications that are made to the objects of the derived class into modifications to the objects in the base class. The derived class defined by a dynamic derivation relationship is a *dynamic derived class*. In most of the cases, the modifications made to the objects of derived classes may be transmitted in various manners to the objects of the base classes, some such manners possibly bringing about side-effects.

6.4.2 Connection between base and derived objects

In order to transmit the modifications from the objects in the derived classes to the objects in the base classes upon which those derived classes are defined a connection must exist between them, in such a way that, given a derived object, the base objects upon whose basis that derived object has been defined are obtained. This connection is the derivation relationship of identity and value presented in section 6.3.3.

If the definition of the derived classes is carried out exclusively by means of object-preserving semantics, the base objects related by way of derivation of identity are obtained directly, with no need to use additional data structures to implement this relationship [Scholl & Schek, 1991; Kim & Kelley, 1995; Ra & Rundensteiner, 1995].

In the system proposed by Kifer, Kim & Sagiv [Kifer et *al*., 1992], this limitation is given more flexibility by defining a new object-generation mechanism where the core attributes may only be identifiers of objects. By using this identifier generation mechanism direct access is gained to the base object identifiers upon which the identity depends.

In none of the mentioned cases is there access to the base objects related exclusively through a derivation relationship of value; for this reason the modifications that affect these objects cannot be transmitted. The set of base objects from which a derived object is defined contains the set of objects that participate in the generation of the identifier. If we only take the latter set into consideration, we will not have all the information available which is necessary to access all the base objects that take part in the definition of the derived object.

Regarding the problem of the transmission of modifications, the solutions which are put forward by other authors (automatic transmission limiting the definition of the derived classes, and transmission of modifications through methods of the derived classes) were presented in chapter 2, section 2.3.5. In the next section, our proposal is presented: transmission of modifications through the derivation relationship.

6.4.3 Transmission of modifications through the derivation relationship

Derived classes are defined directly or indirectly over non-derived classes. In a schema evolution environment it is sometimes necessary to change the definition of the classes in such a way that derived classes become non-derived classes and vice-versa. If the transmission of modifications from a derived class to its base classes has been defined in the methods of the derived class (as in the approaches of section 2.3.5.2) and it is decided to redefine this derived class as a non-derived class, it will be necessary to modify the methods of the derived classes. In such a context, if a derived class becomes non-derived, it is important that only the derivation relationships between the classes must change -not the classes themselves. Thus, our approach is to define the transmission of modifications outside the class: in the derivation relationship.

The derivation relationship defines the correspondence that exists between the base objects and the derived objects. The transmission of modifications between base objects and derived objects (in both directions) is also defined in the derivation relationship.

The modifications of base objects or derived objects cause a change in the relationships that are initially established between them. For each of the methods of the base classes or the derived class which carries out modifications (modification of property values, creation or deleting of objects), the way in which the modifications in question affect the various elements that participate in the derivation relationship must be defined. Such definition is made in the derivation relationship. For each method of modification of the base classes or the derived class, an associated operation is defined in the derivation relationship which is run each time the corresponding method is used, each modification method having a defined *operation consistency relation* [Bratsberg, 1992] in the derivation relationship, which is responsible for maintaining the consistency between the base classes and the derived class.

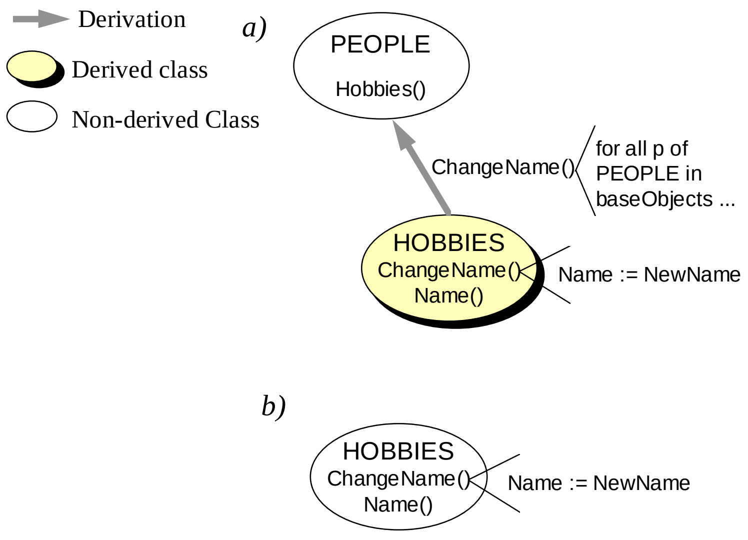


Figure 6.18. Representation of a derived class.

In fig. 6.18 an example of such a proposal is presented. The derived class HOBBIES has been defined from the non-derived class PEOPLE by way of the derivation relationship, fig. 6.18.a, assuming that in the derived class HOBBIES a method ChangeName is defined to change the value of the attribute Name; this method is defined the same way as in a base class. In the derivation relationship an operation is defined that is associated with ChangeName which carries out the transmission of the necessary modifications to the values of Hobbies in the affected objects of PEOPLE, as well as the corresponding changes in the relationship that is established between the base objects and the derived object if any. Similarly, if there is a method that has been defined to change the value of the attribute Hobbies in the base class PEOPLE, then in the derivation relationship the ways in which the possible changes might affect objects of class HOBBIES are defined by means of an operation. The class HOBBIES has the characteristics of a base class; yet, on the other hand, its definition makes it derived.

In order to transform the class HOBBIES into a non-derived class, fig. 6.18.b, the only part affected is that of derivation relationship but not the definition of its properties, i.e. the method ChangeName.

A first version of the modification operations in the derivation relationship can be defined automatically (following the criteria presented in section 2.3.5.1 “Automatic transmission of modifications”); these operations can be further refined by the application administrator.

The value of the core attributes can be modified. The result of modifying the core attributes of an object will depend on the semantics of the modification operations as defined in the associated operation consistency relations.

6.4.4 Operation consistency relations

The concept of operation consistency relation set out by Bratsberg in [Bratsberg, 1992] has been adapted to our environment, fulfilling the initially defined requirements.

In [Bratsberg, 1992] an approach to class evolution is put forward. The evolution of the classes’ intensions and extensions are considered separately. If we consider the extension alone, a class (origin) is related by means of *extent propagation link* with another class (destination) if inserting a new object in the origin class means that it will also have to be included in the destination class. The destination class contains all the objects of the origin class, and may also contain further objects not added in the origin class. The objects in both classes may have common properties as well as properties particular to each respective class.

Associated with the extent propagation link that unites two classes, for each one of the classes *attribute consistency relations* are defined which describe all the attributes of one class that are dependent on the attributes of the other. These relations may be either: derivable if it is possible to obtain the attributes of one class from the attributes of the other; or non-derivable if the attributes are dependent but cannot be derived. The derivation relationships existing between the attributes define the manner of transmission of modifications between classes. In [Monk, 1994] a similar approach is taken exclusively in the case of derivable attributes.

The operation consistency relations are defined as a means of implementing the attribute consistency relations, in order to maintain the consistency when the derived attributes are materialised -that is to say, when the base attributes and the derived attributes happen to be stored separately. They also present an alternative to the attribute consistency relations in order to maintain the consistency between the objects of the base class without describing the dependencies between the attributes. This second focus of operation consistency relations has been considered in the present paper -not at implementation level, but on a conceptual level only.

6.5 Conclusions

The definition of derived classes by object-preserving and object-generating semantics makes it possible to offer a new interface to previously existing objects or to carry out sophisticated re-organisations of existing information: transforming values into objects, aggregating objects to form a new concept. In order to carry out this kind of transformations the identifiers of the new objects have to be generated from identifiers of other objects as well as from values of attributes. This fact, already pointed out by other authors, has been further studied in this chapter.

In some cases, derived classes defined with object-generating semantics have objects in common between them. In previous systems, the only way of sharing these objects was to define inheritance relationships between the classes. A mechanism of generation of object identifiers has been proposed that avoids this requirement.

The derivation relationship is defined between a derived class and the set of its base classes. It is used to integrate the derived classes into the data dictionary. The connections between the derived objects and its respective base objects are defined using derivation relationships of identity and of value.

The derivation relationship is also used to define the way of transmission of modifications between derived classes and their respective base classes. In a schema evolution environment it is sometimes necessary to change the definition of the classes in such a way that derived classes become non-derived classes and vice-versa. In our approach, the set of defined classes in the data dictionary does not change -all that changes is its manner of definition: that is to say, the derivation relationships defined among its classes.

7 External schemas in a schema-evolution environment

External schemas are derived from the database conceptual schema. They can be used to simulate changes to the database conceptual schema. From our point of view, external schemas can contain conceptual schema classes as well as derived classes directly or indirectly defined from conceptual schema classes. Derived classes can be defined by object-preserving or object-generating semantics. In this chapter, this fact is contrasted to the interpretation of the concept of information capacity of object schemas made by other authors.

Sometimes the final users' information requirements change: they need new information which cannot be derived from the information previously contained in the database. The solution that we propose here is the definition of derived classes that can contain non-derived information: partially derived classes. This possibility presents additional problems which are dealt with in an integrated way in this chapter.

7.1 Information in object schemas

As pointed out in [Hull, 1986]: "A central issue in the area of databases is that of data 'relativism', that is, the general activity of structuring the same data in different ways." The problem of determining the *relative information capacity* of schemas has been studied for different models (for relational schemas, both with and without key dependencies [Hull, 1986]; for structures built recursively using some data constructs: set, tuple and union of types [Hull & Yap, 1984; Abiteboul & Hull, 1988]; for a model with complex types and constraints [Miller et *al.*, 1994]). This topic has special importance in the areas of definition of external schemas and integration of different schemas. It has to be further studied for object-oriented models.

The information contained in an object schema is represented by its classes -by their intension and extension. In object schemas, the concept of *data relativism* is implemented defining external schemas and derived classes.

Studying this topic in depth is out of the scope of this work. It can be the topic of another thesis. In this section, only a short introduction to it is made; specifically, only the different possibilities of definition of derived classes in order to be included into external schemas are studied.

7.1.1 Information in external schemas

External schemas are derived from the database conceptual schema. Each external schema describes the part of the information of the conceptual schema appropriate to the group of users to whom it is addressed.

An external schema may include classes defined in the conceptual schema just as it may also contain derived classes -directly or indirectly defined on the basis of conceptual schema classes- that, from our point of view, do not necessarily need to be included in the conceptual schema. Derived classes are defined and included in the data dictionary. The information represented with derived classes is already represented in the conceptual schema.

The information contained in any external schema has to be also contained in the conceptual schema. If an object is to be created or modified in an external schema, the information necessary to obtain this object from the objects into the conceptual schema has to be also added to the conceptual schema (problem of transmission of modifications, see chapter 6, section 6.4).

7.1.2 Simulating conceptual schema transformations using external schemas

In [Tresch & Scholl, 1993] the use of external schemas in order to avoid re-organisations of the conceptual schema was proposed in the following terms:

> Schema transformations usually follow from evolutionary changes of the logical object structure, that is, the database schema.
>
> (...) Schema transformations can be classified according to their impact on the object modelling capacity [Abiteboul & Hull, 1988]:
>
> - *Capacity preserving* transformations do not affect the modelling possibilities. That is, the same potential set of objects can be represented after transformation.
> - *Capacity reducing* transformations reduce the modelling possibilities, such that with these transformations, information is lost.
> - Finally, *capacity augmenting* transformations enhance the information contents of the schema.
>
> (...) Can reorganisations always be avoided? In most cases: schema transformations that are capacity preserving and capacity reducing can always be avoided. However, the latter can produce some nondeterminism. Unfortunately, capacity augmenting transformations require some propagation to the physical level.
>
> (...) The transformation from an object-oriented modelling to a value-oriented modelling of the same situation is an example of a capacity reducing transformation.
>
> (...) Notice that object-to-value transformation is capacity reducing, and therefore, the information is lost. (...) Consequently, the reverse transformation, value-to-object, is capacity augmenting.
>
> Transformations that are capacity augmenting cannot be simulated using views (external schemas).

In these definitions, it can be seen that the information capacity of a schema is directly related to the potential set of objects that can be represented in that schema. According to them, an external schema can not include derived classes defined by object-generating semantics.

In [Hull, 1986], different measures of relative information capacity are defined for relational databases based in the following definition: "Suppose that ***P*** and ***Q*** are two relational database schemata. Speaking informally, we say that ***Q*** *dominates* ***P*** if there are functions σ and τ such that (*i*) σ maps the family of instances of ***P*** into the family of instances of ***Q***, (*ii*) τ maps the family of instances of ***Q*** into the family of instances of ***P***, and (*iii*) the composition of σ followed by τ is the identity on the family of instances of ***P***." The different measures of information capacity are obtained by making certain restrictions on the maps σ and τ.

Therefore, the question is, can an external schema include a derived class defined by object-generating semantics from classes included into the conceptual schema? Our answer to this question, contrary to Tresch & Scholl [Tresch & Scholl, 1993], is "yes".

In order to show the reasons for this answer, in fig. 7.1 an example of a definition of an external schema is presented. It is based on the set of derived classes defined in the previous chapter, in fig. 6.1. The non-derived class PEOPLE has the property Hobbies which returns a set of names of hobbies for each object. From the class PEOPLE the derived class HOBBIES is defined by object-generating semantics; this class represents the existing hobbies in object form. The class PEOPLE' has been defined by object-preserving semantics from the class PEOPLE. This class represents the same concept represented by the initial class PEOPLE but being adapted to the new representation provided by the derived class HOBBIES with which it is related by way of an aggregation relationship.

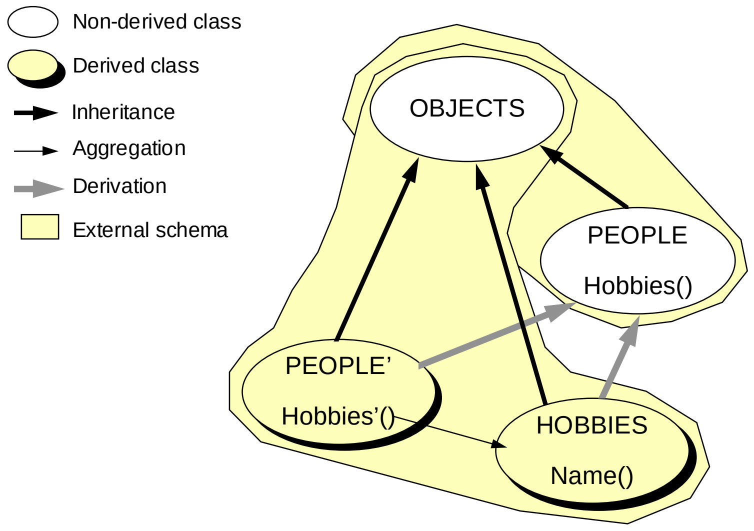


Figure 7.1. Derived classes defined with object-preserving and object-generating semantics.

Taking into account the schema composed by the class PEOPLE and the new external schema composed by the classes PEOPLE' and HOBBIES of fig. 7.1, if a restriction is defined between the classes PEOPLE' and HOBBIES in such a way that no objects can exist in class HOBBIES without being related by aggregation with some object of class PEOPLE', the information capacity of both schemas is the same.

According to the definition of dominance of schemas [Hull, 1986] presented before, the schemas of fig. 7.1 dominate each other, their information capacity is equivalent, the new external schema is defined from the original schema, all its possible instances can be obtained from the original schema; and all the information that can be contained in the original schema can be obtained from the new external schema.

If such a restriction was not defined between the classes PEOPLE' and HOBBIES, the class HOBBIES could contain objects which would not have any relationship with values of the property hobbies of class PEOPLE. Therefore, this schema would dominate the original one, but the opposed affirmation would not be true.

Can an external schema be defined that only contains the class HOBBIES? This class has been defined by object-generating semantics, each one of its objects is related with values of attributes of objects included in its base class. If a new object is to be added to the class HOBBIES, a new value has to be added to the corresponding attribute of some object in class PEOPLE. If allowed, the update policy from class HOBBIES to class PEOPLE will be defined in the operations of the derivation relationship (see chapter 6, section 6.4.3). The modelling possibilities of a class are given by its structural part (the set of attributes of its objects), and also by its behavioural part (the set of methods that can be applied to its objects). Therefore, if the class HOBBIES is defined consistently, the answer to this question is “yes”.

From our point of view, a class or a set of classes defined by object-generating semantics does not necessarily contain more information than their base classes. We agree with Tresch & Scholl [Tresch & Scholl, 1993] in that the operations upon the conceptual schema in order to obtain an external schema must be capacity preserving or capacity reducing. In other words, these transformations do not affect or reduce the modelling possibilities of the conceptual schema. It will be this way if all the possible instances of external schemas can be obtained from the instances of the conceptual schema by the derivation relationship.

7.2 Non-derived information in classes

The base classes of a derived class may be other derived classes and/or non-derived classes. The information contained in the non-derived classes is obtained directly from the database. By applying the transitivity of the derivation relationship, the information of the derived classes is calculated based on the data stored in the database, the derived classes offering a new interface for such data. Along the derivation dimension, the derived classes share the data stored in the database with their base classes. In no case may the derived classes contain information that has not been obtained from their base classes.

Sometimes the end-users’ information requirements change. They need new information which cannot be derived from the information previously contained in the database; and this new information must be incorporated into the database. Adding the new information might either call for the definition of new non-derived classes or the modification of previously existing classes so that it can be included. Modifying a previously existing class may turn out to be troublesome in some cases, if new classes have been derived from it or if there are programs which use it.

7.2.1 Partially derived classes

Conceptually, a solution to support schema evolution with some capacity augmenting transformations consists in allowing derived classes that may contain non-derived information -that is to say, partially derived classes. The non-derived information is added to the information obtained from the base classes, thus avoiding the need to modify the definition of other classes.

The definition of partially derived classes allows us to define derived classes through capacity augmenting transformations from the base classes without the rest of the existing classes being affected. The base classes and the partially derived classes share the information that may be contained in both classes. The additional information that cannot be contained in the base classes is contained exclusively in the non-derived parts of the partially derived classes.

The intension of a class is made up of the set of properties of that class. The extension is the set of occurrences of the class, the set of objects included in the class. In a partially derived class non-derived elements may be defined in the class's intension just as in its extension.

7.2.1.1 Non-derived elements in the intension

Several systems allow non-derived elements to be defined in the intension of the partially derived classes [Bertino, 1992; Ra & Rundensteiner, 1995; Naja & Mouaddib, 1995; Bertino et *al*., 1996], partially derived classes with non-derived properties can be defined. Such properties may be initialised with default values.

A further essential transformation is the possibility of modifying the type associated with a property in such a way that it has a greater information capacity, i.e. a simple property becomes multi-valued in the partially derived class, or a property defined as an integer in the base class becomes real in the new class. The initial value of the generalised property of the partially derived class's objects is the value obtained from the base objects. This information may be extended at a later stage. The non-derived information defined in the intension must be stored somewhere for each one of the occurrences of the new defined class.

The ability to define partially derived classes with non-derived properties simplifies the execution of some schema evolution operations but, as we will shown in the following point, there are requirements of evolution which are not catered for by this type of transformation.

7.2.1.2 Non-derived elements in the extension

We come across an example of these additional requirements in [Bertino et *al*., 1996]: a derived class is defined by object-preserving semantics which hides a constraint defined in the corresponding base class. For this reason, the derived class could contain objects that cannot be included in the base class. However, although the insertion of such objects into the derived class looks valid to an user of the class, this insertion would not be possible unless a capacity augmenting mechanism were in place.

We come across a similar situation in the example of fig. 7.1, where the class HOBBIES has been defined by object-generating semantics from the property Hobbies of the class PEOPLE. By way of this transformation the representation of the hobby concept has changed. It has turned from the value of an object's multi-valued property into an object. Having the hobby concept represented as a value, we will only be able to store a hobby if it happens to appear in an occurrence of PEOPLE. Once the class HOBBIES is defined, if a restriction exists which specifies a similar dependency between the occurrences of the class HOBBIES and the occurrences of the new class PEOPLE', there will be no problem. If such a restriction does not exist, or if an external schema is defined that contains the class HOBBIES but not the class PEOPLE', in these schemas it would be possible to insert new objects in the class HOBBIES without being associated with any occurrence in the class PEOPLE'. The problem is that such an operation is not supported by the base schema.

7.2.1.3 Local extension of a partially derived class

The problem is the same in both cases: a class has been created by way of a transformation which increases the information capacity with respect to the base classes. So, the base classes cannot provide support for all the information that the new defined class may contain. One solution to this problem consists in allowing the definition of partially derived classes with non-derived *local extensions*. Based on the same term defined by Bratsberg [Bratsberg, 1992], a partially derived class's local extension contains the non-derived elements that are defined both in the class's intension as well as its extension.

Let us consider a concrete example for the class HOBBIES. Let us suppose it were necessary to deal with hobbies regardless of whether they were associated with some occurrences of the class PEOPLE'. We define HOBBIES as a partially derived class. If we introduce a new hobby whose core attribute value is Basketball, such an object will be created in the local extension of the class HOBBIES. In fig. 7.2 it can be seen that this object has been created and has no associated base objects.

HOBBIES				
oid	**Core Attributes**	**Base Objects**	**...**	**Name**
h_{11}	Tennis	$\{p_1, p_3, p_4\}$		Tennis
h_{12}	Football	$\{p_1\}$		Football
h_{14}	Reading	$\{p_2\}$		Reading
h_{15}	Chess	$\{p_2, p_3, p_4\}$		Chess
h_{16}	Basketball	▨		Basketball
h_{13}	Driving	▨		Driving

Figure 7.2. Representation of the class HOBBIES with non-derived objects.

If another final user modifies the objects in the class PEOPLE and it turns out that no object remains that has an associated concrete hobby, the object of the class HOBBIES corresponding to the hobby in question would cease to exist. If the class HOBBIES is handled independently, this could be undesired behaviour. Let us assume that Driving ceases to be a hobby of anybody. In fig. 7.2, it can be seen that the object in question has not been eliminated from the class HOBBIES, but instead has moved to the local extension (it has no associated base objects). In the transmission of the corresponding modifications, the object-generated in the class HOBBIES has not been deleted, but, rather, has been transformed into a non-derived object.

In the same way that a derived object may become non-derived (i.e. the object that corresponds to Driving in the example), a non-derived object may become derived. If, following a modification in the class PEOPLE, it turns out that Basketball is defined as the value of the property Hobbies of one of the objects, a new object will not be generated in the class HOBBIES but, instead the non-derived object will become derived, and associated with its corresponding base objects.

The extension of a partially defined class is made up of the derived extension obtained from the base classes and the local extension. The objects of the local extension may go to form part of the derived extension; and the objects of the derived extension may pass over to the local extension.

7.2.2 Extent propagation links

As pointed out in section 6.4.3 and 6.4.4, in [Bratsberg, 1992] an approach to class evolution is put forward. The evolution of the classes' intensions and extensions are considered separately. If we consider the extension alone, a class (origin) is related by means of extent propagation link with another class (destination) if inserting a new object in the origin class means that it will also have to be included in the destination class. The destination class contains all the objects of the origin class, and may also contain further objects not added in the origin class. The objects in both classes may have common properties as well as properties particular to each respective class.

If we consider the extension of the classes, the various possibilities which are allowed by Bratsberg [Bratsberg, 1992] are set out in parts (*a*), (*b*) and (*c*) of fig. 7.3: i.e. in part (*a*) class c_1 is the destination of an extent propagation link whose origin is class c_2. As can be seen, class c_1 contains all the extension of class c_2, but not necessarily all its intension. In the case depicted in part (*d*) extent propagation link cannot be used, since none of the classes contains all the occurrences of the other class, although both classes possess common elements.

In order to depict the situation in part (*d*) of fig. 7.3 with terminology resembling that used in [Bratsberg, 1992] we define a new relation: the *conditional extent propagation link*, which asserts that the occurrences of the origin class may become occurrences of the destination class under certain conditions. This new relationship may also be defined between classes related by means of extent propagation links shown in parts (*a*) and (*b*) of fig. 7.3.

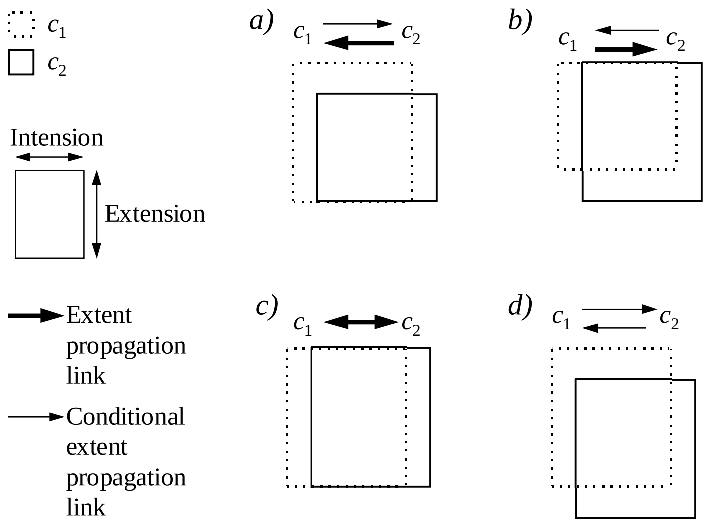


Figure 7.3. Extent propagation between classes.

The extent propagation links set out in [Bratsberg, 1992] define the flow of copies of object identifiers between classes. For the derived classes defined by object-generating semantics we extend this definition, along with the definition of the conditional extent propagation link, so that instead of defining the flow of object identifiers, they define the flow of information between classes. Our derivation relationship introduced in chapter 6, defined between a derived or partially derived class and its base classes, corresponds to a set of conditional extent propagation links. The conditional extent propagation links define the flow of information from the base classes to the derived class and vice-versa.

7.3 A schema-evolution environment

By definition, the information contained in the external schemas has to be obtained from the conceptual schema. The conceptual schema has to include all the non-derived classes defined in the data dictionary, and it can also contain derived classes. In the cases in which non-derived information (in the form of new non-derived classes or partially derived classes) is required in order to satisfy the end-users' new information requirements, this information can not be included into an external schema without being previously included into the conceptual schema.

7.3.1 Test environment

In the external schema design process, continually new non-derived information may be required to be subsequently rejected; this means that the conceptual schema has to be continually modified until a final version of the external schema is achieved.

In order to avoid the continual modification of the conceptual schema, the availability of a *test environment* is very useful. In this environment, *temporal external schemas* can be defined that include non-derived information without having the conceptual schema affected.

When a temporal external schema is accepted by the end-users, this schema has to become a real external schema: the conceptual schema has to be modified, if necessary, in order to include the non-derived information of the new schema.

Therefore, the definition of partially-derived classes or new non-derived classes will only affect the conceptual schema if the decision is made to include them in some external schema, in other case, they will be exclusively defined in the test environment of the data dictionary.

7.3.2 Evolution of the conceptual schema

Given the conceptual schema, all the external schemas defined over it, and a temporal external schema with non-derived information, in order to transform the temporal external schema into an external schema the following transformations have to be carried out:

- The conceptual schema is modified in order to embody the non-derived information.

- The temporal external schema becomes an external schema, derived from the conceptual schema: the non-derived information becomes derived.

- The previously existing external schemas are not affected by this transformation of the conceptual schema.

The most immediate way of achieving this would be adding the new non-derived or partially-derived classes to the conceptual schema, as it is made by Ra & Rundensteiner [Ra & Rundensteiner, 1995] for derived classes with non-derived properties. But in this solution, complex class hierarchies would be obtained with classes which are not strictly necessary.

For example, in fig. 7.1 only the new derived classes PEOPLE' and HOBBIES should be included into the conceptual schema because they dominate the schema composed exclusively by class PEOPLE; the class PEOPLE would become a derived class. In the solution proposed by Ra & Rundensteiner, the class PEOPLE would also be included into the conceptual schema.

As it has been shown in chapter 6, section 6.4.3, derived classes can be transformed directly into non-derived classes and vice versa; the only element that has to be changed is the derivation relationship.

In some cases, the classes that will have to be added to the conceptual schema will not be classes belonging to the temporal external schema considered, but new classes defined from classes previously existing in the conceptual schema as well as from classes in the temporal external schema. All this process can be carried out manually, but we think that most of it can be carried out automatically. This is one of ours topics of further study.

7.3.3 Non-side effect external schemas

The concept of *non-side effect external schema* (non-side effect view) is defined by Gentile & Zicari [Gentile & Zicari, 1994] as an external schema (view) which is re-computed dynamically so that conceptual schema modifications are (whenever possible) “filtered out” from applications using the external schema (view).

In the system proposed by Gentile & Zicari, external schemas have to be transformed in order to be adapted to the new conceptual schema obtained after the modifications. In the system proposed in the present work, external schemas are always non-side effect; even in this case, previously existing external schemas do not have to be modified after changes in the conceptual schema have been carried out.

The classes included in external schemas are classes of the conceptual schema or classes derived directly or indirectly from classes of the conceptual schema. If some class of the conceptual schema, which is a base class of some derived class or is included in an external schema has to be modified, a new class is defined and the previously existing class is not affected by this modification. The only thing that can change is its definition, it can become a derived class. If the class is no longer included into the conceptual schema it will remain defined in the data dictionary. Therefore, the rest of classes defined over it will not be affected by this change.

7.4 Conclusions

External schemas can contain derived classes defined by object-preserving as well as by object-generating semantics.

Derived classes offer a new interface of access to the information contained in their base classes. In a schema evolution environment we consider it necessary to be able to define partially derived classes: derived classes which can also contain non-derived elements. If some class with non-derived information needs to be incorporated into some external schema, the non-derived information will have to be included into the conceptual schema. The same derivation operations discussed in chapter 6 can be used to produce partially derived classes.

Preceding systems of definition of derived classes [Bertino, 1992; Ra & Rundensteiner, 1995; Naja & Mouaddib, 1995; Bertino et *al*., 1996] only allow partially derived classes to be defined by having non-derived elements in the intension: non-derived properties may be defined. For this situation, a way of propagating the transformations to the conceptual schema is presented in [Ra & Rundensteiner, 1995]. In the system proposed here, partially derived classes can be defined with non-derived elements in the intension as well as in the extension of the class.

Other papers on schema evolution [Skarra & Zdonik, 1986; Andany et *al*., 1991; Tresch, 1991; Tresch & Scholl, 1992; Monk & Sommerville, 1993; Brèche et *al*., 1995; Ferrandina et *al*., 1995] offer such mechanisms as lazy conversion, or class or schema versioning, with elements common to the derivation relationship; but they do not offer the possibilities that derived and partially derived class definition affords.

In order to avoid unnecessary modifications of the conceptual schema the use of a test environment for the definition of temporal external schemas has been proposed. When an external schema with non-derived information is to be defined, the conceptual schema has to be modified in order to include the non-derived information of the new schema.

External schemas defined in the environment presented here are non-side effect external schemas, they are not affected by modifications carried out in the conceptual schema.

8 Conclusions

Given the importance and usefulness of having an external schema definition mechanism in OODB, this thesis has gone deeply into this subject. A new external schema definition methodology has been presented that fulfils the ANSI/SPARC three-level schema architecture. Most of the concepts used in this methodology are not new -however, what is new is the defined combination and also the consequent results.

8.1 Main results

The main contributions of this thesis are the following:

- Proposal of a new external schema definition methodology according to the ANSI/SPARC framework, which in OODBs had not been taken into account. This methodology considerably simplifies the process of definition of external schemas. The defined methodology is not particular to any object oriented model, it uses common concepts to most of the existing object oriented models.

- The new external schema definition methodology offers a solution to the problem of integration of derived classes with the rest of the existing classes in object schemas. Integration is made in two phases: first, into the data dictionary; then, into the defined external schema. The solution put forward reduces the number of auxiliary classes that must be generated in order to carry out the integration in respect to other alternative solutions

- Definition of the new concepts of transformable and non-transformable classes. Transformable classes can be automatically modified, hence avoiding the need to explicitly define all the classes that we want to include into the external schema. It has been shown how the transformable and non-transformable class concepts simplify the external schema definition process.

- Study of the definition of classes with object-preserving and object-generating semantics, in particular, study of the problem of generation of object identifiers. Both object definition semantics are dealt with in an integrated way. New objects may be defined from the association of previously existing objects, just as from values of existing objects' properties -transformation of values into objects. Unlike previous works, a distinction has been made between the identifier generation mechanism and the mechanism for maintaining the connection between the objects of the base classes and the objects of the derived classes.

- A solution for the problem of transmission of modifications between base and derived classes has been proposed. The solution that we have put forward is an adaptation to our environment of the proposal made in [Bratsberg, 1992] which solves some of the inconveniences of other alternatives. The derivation relationship contains the initial definition of the objects of the derived class as well as a set of operation consistency relations that define how the relationships between the base objects and the derived objects change with the modification operations.

- In order to satisfy the end-user's requirements of new information in a schema evolution environment we have proposed the definition of partially derived classes: classes that may contain derived information as well as non-derived information. The base classes are updated with all the new information that they can contain arising out of the partially derived classes. The partially derived classes may contain non-derived information in the intension and moreover, unlike other systems, in the extension. The objects of the partially derived classes may cease to be derived or become derived in response to the modifications.

- The information contained in the external schemas must be a subset of the information contained in the conceptual schema. If we want to include partially derived classes in some external schema, the non-derived information contained in such classes must be included in the conceptual schema -that is to say, the conceptual schema must evolve. In order to avoid unnecessary modifications of the conceptual schema the use of a test environment for the definition of temporal external schemas has been proposed.

- The definition of DCMs using Prolog in order to specify different aspects of OODBs has been proposed. The result of the specification process using this technique is an executable prototype of the system. The use of this technique has been proposed mainly due to the difficulty of building prototypes of the mentioned elements over commercial OODBs.

8.2 Future work

Some of the topics of interest for future work are as follows:

- Study the subsumption relationships between classes. Define a *subsumes()* function for a specific object model, i.e. BLOOM.

- Study the problem of determining the relative information capacity for object-oriented models, that is to say, study the subsumption relationships between object schemas.

- If some temporal external schema containing non-derived information (new non-derived classes and/or partially derived classes) is to be defined as an external schema, the conceptual schema has to be modified in order to embody the non-derived information. Algorithms that automatically carry out this process are going to be proposed.

- Extend the external schema definition methodology in order to adapt it to use the concepts of richer object oriented models as BLOOM [Castellanos et *al*., 1992].

Appendix A. Conceptual schema definition DCM

The following is the source code of the conceptual schema definition DCM developed:

```
/*********************************************************************
**********************************************************************
                              CS_DEF.DAT
       Input file.
**********************************************************************
*********************************************************************/

csdefine(type("person",[]))
csdefine(typeproperty("person","name",[type("string")]))
csdefine(typeproperty("person","address",[setclass("addresses")]))
csdefine(type("employee",["person"]))
csdefine(typeproperty("employee","category",[type("string")]))
csdefine(typeproperty("employee","salary",[type("integer")]))
csdefine(type("address",[]))
csdefine(typeproperty("address","city",[type("string")]))
csdefine(class("people","person",[]))
csdefine(class("clients","person",["people"]))
csdefine(class("employees","employee",["people"]))
csdefine(class("addresses","address",[]))
```

```
/*********************************************************************
**********************************************************************
                              CS.DEF
       Output file.
**********************************************************************
*********************************************************************/

csproperty(50,"name",86)
csproperty(51,"address",87)
csproperty(52,"category",88)
csproperty(53,"salary",89)
csproperty(54,"city",90)
cstype(37,"any",[],[],77)
cstype(38,"real",[37],[],78)
cstype(39,"integer",[38],[],79)
cstype(40,"character",[39],[],80)
cstype(41,"string",[37],[],81)
cstype(43,"person",[37],[51,50],83)
cstype(44,"employee",[43],[51,50,53,52],84)
cstype(45,"address",[37],[54],85)
cstypeproperty(43,50,[t(41)],91)
cstypeproperty(43,51,[c_(49)],92)
cstypeproperty(44,52,[t(41)],93)
cstypeproperty(44,53,[t(39)],94)
cstypeproperty(45,54,[t(41)],95)
cstypeproperty(44,50,[t(41)],112)
cstypeproperty(44,51,[c_(49)],112)
csclass(42,"objects",37,[],[49,46],82)
csclass(46,"people",43,[42],[48,47],96)
csclass(47,"clients",43,[46],[],97)
csclass(48,"employees",44,[46],[],98)
csclass(49,"addresses",45,[42],[],99)
```

```
/*********************************************************************
**********************************************************************
                              CS_GLDOM.PRO
**********************************************************************
*********************************************************************/

global domains

       file           = idFile; timeFile; objectFile

       id             = integer
       idList         = id*
       time           = integer

       domain         = t(id); c(id); t_(id); c_(id)
       signature      = domain*
       signatures     = signature*

       propertyName   = symbol
       typeName       = symbol
       className      = symbol
```

```
/*******************************************************************
********************************************************************
                                CS_GLPRE.PRO
********************************************************************
*******************************************************************/

global predicates

/*******************************************************************
        Access to base facts.
*******************************************************************/
        propertyName(id,propertyName,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)

        typeName(id,typeName,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)
        typeSupertypes(id,idList,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)

        typeAllProperties(id,idList,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)

        typePropertyAtT(id,id,time) -
                (i,i,i)(i,i,o)(i,o,i)(i,o,o)
                (o,i,i)(o,i,o)(o,o,i)(o,o,o)
        typeProperty(id,id,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)
        typePropertySignature(id,id,signature,time) -
                (i,i,i,i)(i,i,o,i)(i,o,i,i)(i,o,o,i)
                (o,i,i,i)(o,i,o,i)(o,o,i,i)(o,o,o,i)
        newTypeProperty(id,id,signature,time) -
                (i,i,i,i)

        className(id,className,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)
        classSuperclasses(id,idList,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)
        classType(id,id,time) -
                (i,i,i)(i,o,i)(o,i,i)(o,o,i)

        assertzTmpIdList(idList) -
                (i)
        retractTmpIdList(idList) -
                (i)(o)

/*******************************************************************
        Inconsistency verification.
*******************************************************************/

        inconsistency(time) -
                (i)
        inconsistency2(time) -
                (i)

/*******************************************************************
        Signatures.
*******************************************************************/

        signatureExtrictSubdomains(signature,signature,time) -
                (i,i,i)

/*******************************************************************
        Types.
*******************************************************************/

        supertype(id,id,time) -
                (i,i,i)
        directSupertype(id,id,time) -
                (i,i,i)
        indirectSupertype(id,id,time) -
                (i,i,i)
        supertypeOfSomeOne(id,idList,time) -
                (i,i,i)
        redundantSupertypes(idList,time) -
                (i,i)
        typeInheritanceWithSomeOne(id,idList,time) -
                (i,i,i)
        typeInheritance(id,id,time) -
                (i,i,i)
        commonTypeProperty(id,id,id,time) -
```

```
                (i,i,i,i)
        sameTypeOrSubtype(id,idList,time) -
                (i,i,i)
        deriveTypeProperties(id,idList,time) -
                (i,o,i)
        definedLowestCommonSupertype(id,id,time) -
                (i,i,i)
        lowestCommonSupertype(id,id,id,time) -
                (i,i,i,i)(i,i,o,i)
        greatestCommonSubtype(id,id,id,time) -
                (i,i,i,i)(i,i,o,i)

/*********************************************************************
        Classes.
*********************************************************************/

        superclass(id,id,time) -
                (i,i,i)
        directSuperclass(id,id,time) -
                (i,i,i)
        indirectSuperclass(id,id,time) -
                (i,i,i)
        superclassOfSomeOne(id,idList,time) -
                (i,i,i)
        redundantSuperclasses(idList,time) -
                (i,i)
        classInheritanceWithSomeOne(id,idList,time) -
                (i,i,i)
        classInheritance(id,id,time) -
                (i,i,i)
        commonClassProperty(id,id,id,time) -
                (i,i,i,i)
        classesTypes(idList,idList,time) -
                (i,o,i)
        deriveClassSubclasses(id,idList,time) -
                (i,o,i)
        definedLowestCommonSuperclass(id,id,time) -
                (i,i,i)
        lowestCommonSuperclass(id,id,id,time) -
                (i,i,i,i)(i,i,o,i)
        greatestCommonSubclass(id,id,id,time) -
                (i,i,i,i)(i,i,o,i)

/*********************************************************************
        System primitives
*********************************************************************/

    /*-------------------------------------------------------
        Generate system identifier.
    -------------------------------------------------------*/
        generateId(id) -
                (o)

    /*-------------------------------------------------------
        Get system time.
    -------------------------------------------------------*/
        now(time) -
                (o)

/*********************************************************************
        Id lists.
*********************************************************************/

    /*-------------------------------------------------------
        An Id is member of a list of Ids.
    -------------------------------------------------------*/
        memberId(id,idList) -
                (i,i)

    /*-------------------------------------------------------
        Add a new Id to an Id list only if it is not included
        yet.
    -------------------------------------------------------*/
        addNewId(idList,id,idList) -
                (i,i,o)

    /*-------------------------------------------------------
        Intersection of two lists of identifiers.
    -------------------------------------------------------*/
        intersectionIdList(idList,idList,idList) -
                (i,i,o)
```

```
/*----------------------------------------------------
   Equal lists.
------------------------------------------------------*/
   equalIdList(idList,idList) -
          (i,i)

/*----------------------------------------------------
   All the elements of a list are contained in another
   list.
------------------------------------------------------*/
   allContainedIdList(idList,idList) -
          (i,i)
```

```
/*******************************************************************
********************************************************************
                             CS.PRO
       Conceptual Schema Definition.
********************************************************************
*******************************************************************/

code = 4096

include "cs_gldom.pro"

domains
       supertypes            = typeName*
       superclasses          = className*

       domainName            = type(typeName);
                               class(className);
                               settype(typeName);
                               setclass(className)

       signatureName         = domainName*

       csDefinitionTerm      = type(typeName,supertypes);
                               typeProperty(typeName,propertyName,signatureName);
                               class(className,typeName,superclasses);
                               consulterror;
                               error

database
       tmpIdList(idList)

       error(csDefinitionTerm)

       csDefine(csDefinitionTerm)

       csDefineClassName(id,className)
       csDefineTypeName(id,typeName)
       csDefineClass(id,className,id,idList,time)
       csDefineType(id,typeName,idList,time)

       csProperty(id,propertyName,time)
       csType(id,typeName,idList,idList,time)
       csTypeProperty(id,id,signature,time)
       csClass(id,className,id,idList,idList,time)


include "cs_glpre.pro"


predicates
       conceptualSchemaDefinition
       systemComponentDefinition
       nameDefinition
       typeDefinition
       supertypeNames2Ids(supertypes,idList)
       typeRoot(idList,idList)
       propertyDefinition
       typePropertyDefinition
       signatureNames2Ids(signatureName,signature)
       classDefinition
       superclassNames2Ids(superclasses,idList)
       classRoot(idList,idList)
       removeDefinitionTerms
       deriveDirectSubclassRelationship
       deriveAllTypeProperties


goal
       makewindow(1,7,7,"",0,0,25,80),

       write("\n  Defining the Conceptual Schema..."),
       conceptualSchemaDefinition,

       write("\n\n\n  To finish press the space bar, please."),
       readchar(_),
       removewindow.


clauses
```

```
/******************************************************************
       Access to base facts.
*******************************************************************/

       propertyName(IdP,Property,T) :-
              csProperty(IdP,Property,T1),
              T1 <= T.

       typeName(IdT,Type,T) :-
              csDefineType(IdT,Type,_,T1),
              T1 <= T.
       typeSupertypes(IdT,IdTs,T) :-
              csDefineType(IdT,_,IdTs,T1),
              T1 <= T.

       typeAllProperties(IdT,IdPs,T) :-
              csType(IdT,_,_,IdPs,T1),
              T1 <= T.

       typePropertyAtT(IdT,IdP,T) :-
              csTypeProperty(IdT,IdP,_,T).
       typeProperty(IdT,IdP,T) :-
              csTypeProperty(IdT,IdP,_,T1),
              T1 <= T.
       typePropertySignature(IdT,IdP,Signature,T) :-
              csTypeProperty(IdT,IdP,Signature,T1),
              T1 <= T.
       newTypeProperty(IdT,IdP,Signature,T) :-
              assertz(csTypeProperty(IdT,IdP,Signature,T)),
              !.

       className(IdC,Class,T) :-
              csDefineClass(IdC,Class,_,_,T1),
              T1 <= T.
       classSuperclasses(IdC,IdCs,T) :-
              csDefineClass(IdC,_,_,IdCs,T1),
              T1 <= T.
       classType(IdC,IdT,T) :-
              csDefineClass(IdC,_,IdT,_,T1),
              T1 <= T.


       assertzTmpIdList(IdList) :-
              assertz(tmpIdList(IdList)),
              !.
       retractTmpIdList(IdList) :-
              retract(tmpIdList(IdList)),
              !.


/******************************************************************
       Conceptual Schema definition.
*******************************************************************/

       conceptualSchemaDefinition :-
              assertz(error(consulterror)),
              consult("cs_def.dat"),
              retract(error(consulterror)),

              systemComponentDefinition,
              nameDefinition,
              typeDefinition,
              propertyDefinition,
              typePropertyDefinition,
              classDefinition,
              not(error(_)),
              now(T),
              not(inconsistency(T)),
              deriveDirectSubclassRelationship,
              deriveAllTypeProperties,
              now(T2),
              not(inconsistency2(T2)),

              removeDefinitionTerms,
              save("cs.def"),
              write("\n\n  Conceptual Schema defined."),
              !.
       conceptualSchemaDefinition :-
              error(consulterror),
              write("\n\n  DE000: Sintax error in the input file."),
              !.
       conceptualSchemaDefinition :-
               !.

    /*---------------------------------------------------
       Remove temporal definition terms in order to
       mantain just the Conceptual Schema definition terms.
    ---------------------------------------------------*/
       removeDefinitionTerms :-
               retract(csDefine(_)),
               fail.
       removeDefinitionTerms :-
               retract(csDefineClassName(_,_)),
               fail.
       removeDefinitionTerms :-
               retract(csDefineTypeName(_,_)),
               fail.
       removeDefinitionTerms :-
               retract(csDefineClass(_,_,_,_,_)),
               fail.
       removeDefinitionTerms :-
               retract(csDefineType(_,_,_,_)),
               fail.
       removeDefinitionTerms :-
               !.


/*********************************************************************
       System components definition.
*********************************************************************/

       systemComponentDefinition :-
               now(T1),
               generateId(IdA),
               assertz(csDefineType(IdA,any,[],T1)),
               assertz(csDefineTypeName(IdA,any)),

               now(T2),
               generateId(IdR),
               assertz(csDefineType(IdR,real,[IdA],T2)),
               assertz(csDefineTypeName(IdR,real)),

               now(T3),
               generateId(IdI),
               assertz(csDefineType(IdI,integer,[IdR],T3)),
               assertz(csDefineTypeName(IdI,integer)),

               now(T4),
               generateId(IdCh),
               assertz(csDefineType(IdCh,character,[IdI],T4)),
               assertz(csDefineTypeName(IdCh,character)),

               now(T5),
               generateId(IdS),
               assertz(csDefineType(IdS,string,[IdA],T5)),
               assertz(csDefineTypeName(IdS,string)),

               now(T6),
               generateId(IdCl),
               assertz(csDefineClass(IdCl,objects,IdA,[],T6)),
               assertz(csDefineClassName(IdCl,objects)),
               !.


/*********************************************************************
       Temporal definition of names, this way there can be crossed
       references.
*********************************************************************/

       nameDefinition :-
               csDefine(type(Name,A_)),
               not(error(_)),
               assertz(error(type(Name,A_))),
               not(csDefineTypeName(_,Name)),
               retract(error(type(Name,A_))),
               generateId(IdT),
               assertz(csDefineTypeName(IdT,Name)),
               fail.
       nameDefinition :-
               csDefine(class(Name,A_,B_)),
               not(error(_)),
               assertz(error(class(Name,A_,B_))),
               not(csDefineClassName(_,Name)),
               retract(error(class(Name,A_,B_))),
               generateId(IdC),
               assertz(csDefineClassName(IdC,Name)),
               fail.
       nameDefinition :-
               error(Term),
               write("\n\n  DE001: Type or class name already defined in:\n  ",Term),
               retract(error(Term)),
               assertz(error(error)),
               !.
       nameDefinition :-
               !.


/*****************************************************************
       Initial Type definition.
*****************************************************************/

       typeDefinition :-
               csDefine(type(Name,Supertypes)),
               csDefineTypeName(IdT,Name),
               not(error(_)),
               assertz(error(type(Name,Supertypes))),
                       supertypeNames2Ids(Supertypes,IdTsTMP),
                       typeRoot(IdTsTMP,IdTs),
               retract(error(type(Name,Supertypes))),
               now(T),
               assertz(csDefineType(IdT,Name,IdTs,T)),
               fail.
       typeDefinition :-
               error(type(_,_)),
               error(Term),
               write("\n\n  DE002: Supertype name not defined in:\n  ",Term),
               !.
       typeDefinition :-
               !.

    /*--------------------------------------------------
       Conversion from a list of supertype names to a list
       of Ids.
    --------------------------------------------------*/
       supertypeNames2Ids([],[]) :-
               !.
       supertypeNames2Ids([Name|L1],[IdT|L2]) :-
               csDefineTypeName(IdT,Name),
               supertypeNames2Ids(L1,L2).

    /*--------------------------------------------------
       Add the type "any" as the parent of a type if it
       hasn't any one specified.
    --------------------------------------------------*/
       typeRoot([],[IdA]) :-
               csDefineTypeName(IdA,any),
               !.
       typeRoot(IdTs,IdTs) :-
               !.


/*****************************************************************
       Initial Property definition.
*****************************************************************/

       propertyDefinition :-
               csDefine(typeproperty(_,Name,_)),
               not(csProperty(_,Name,_)),
               now(T),
               generateId(IdP),
               assertz(csProperty(IdP,Name,T)),
               fail.
       propertyDefinition :-
               !.


/*****************************************************************
       Initial Type Property definition.
*****************************************************************/

       typePropertyDefinition :-
               csDefine(typeproperty(Type,Property,Signature)),
               not(error(_)),
               assertz(error(typeproperty(Type,Property,Signature))),
                       csDefineTypeName(IdT,Type),
                       csProperty(IdP,Property,_),
                       signatureNames2Ids(Signature,IdSs),
               retract(error(typeproperty(Type,Property,Signature))),
               now(T),
               assertz(csTypeProperty(IdT,IdP,IdSs,T)),
               fail.
       typePropertyDefinition :-
               error(typeproperty(_,_,_)),
               error(Term),
               write("\n\n  DE003: Type or class not defined in:\n  ",Term),
               !.
       typePropertyDefinition :-
               !.

    /*-------------------------------------------------------
       Conversion from a signature list of domain names to
       a signature list of Ids.
    -------------------------------------------------------*/
       signatureNames2Ids([],[]) :-
               !.
       signatureNames2Ids([type(Name)|L1],[t(IdT)|L2]) :-
               csDefineTypeName(IdT,Name),
               signatureNames2Ids(L1,L2).
       signatureNames2Ids([settype(Name)|L1],[t_(IdT)|L2]) :-
               csDefineTypeName(IdT,Name),
               signatureNames2Ids(L1,L2).
       signatureNames2Ids([class(Name)|L1],[c(IdC)|L2]) :-
               csDefineClassName(IdC,Name),
               signatureNames2Ids(L1,L2).
       signatureNames2Ids([setclass(Name)|L1],[c_(IdC)|L2]) :-
               csDefineClassName(IdC,Name),
               signatureNames2Ids(L1,L2).


/*********************************************************************
       Initial Class definition.
*********************************************************************/

       classDefinition :-
               csDefine(class(Class,Type,Superclasses)),
               csDefineClassName(IdC,Class),
               not(error(_)),
               assertz(error(class(Class,Type,Superclasses))),
                       csDefineTypeName(IdT,Type),
                       superclassNames2Ids(Superclasses,IdCsTMP),
                       classRoot(IdCsTMP,IdCs),
               retract(error(class(Class,Type,Superclasses))),
               now(T),
               assertz(csDefineClass(IdC,Class,IdT,IdCs,T)),
               fail.
       classDefinition :-
               error(class(_,_,_)),
               error(Term),
               write("\n\n  DE004: Type or superclass name not defined in:\n  ",Term),
               !.
       classDefinition :-
               !.

    /*-------------------------------------------------------
       Conversion from a list of superclass names to a
       list of Ids.
    -------------------------------------------------------*/
       superclassNames2Ids([],[]) :-
               !.
       superclassNames2Ids([Name|L1],[IdC|L2]) :-
               csDefineClassName(IdC,Name),
               superclassNames2Ids(L1,L2).

    /*-------------------------------------------------------
       Add the class "objects" as the parent of a type if
       it hasn't any one specified.
    -------------------------------------------------------*/
       classRoot([],[IdCl]) :-
               csDefineClassName(IdCl,objects),
               !.
       classRoot(IdCs,IdCs) :-
               !.
```

```
/*********************************************************************
       Subclass relationship derivation.
*********************************************************************/

       deriveDirectSubclassRelationship :-
              csDefineClass(IdC,Class,IdT,IdUs,T),
              now(T1),
              deriveClassSubclasses(IdC,IdDs,T1),
              assertz(csClass(IdC,Class,IdT,IdUs,IdDs,T)),
              fail.
       deriveDirectSubclassRelationship :-
              !.


/*********************************************************************
       Type properties derivation.
*********************************************************************/

       deriveAllTypeProperties :-
              csDefineType(IdT,Type,IdTs,T),
              now(T1),
              deriveTypeProperties(IdT,IdPs,T1),
              assertz(csType(IdT,Type,IdTs,IdPs,T)),
              fail.
       deriveAllTypeProperties :-
              !.


include "cs_incon.pro"
include "cs_signa.pro"
include "cs_types.pro"
include "cs_class.pro"

include "oom_syst.pro"
include "oom_list.pro"
```

```
/*******************************************************************
*******************************************************************
                             CS_INCON.PRO
       Inconsistency verification.
*******************************************************************
*******************************************************************/

clauses

    /*-----------------------------------------------------
       A property can only be defined once in a type.
    -----------------------------------------------------*/
       inconsistency(T) :-
              typePropertyAtT(IdT,IdP,T1),
              typePropertyAtT(IdT,IdP,T2),
              T1 <> T2,
              T1 <= T,
              T2 <= T,

              typeName(IdT,Type,T),
              propertyName(IdP,Property,T),
              write("\n\n  IN001: Property ",Property,
                     " defined more than once in type ",Type,"."),
              !.

    /*-----------------------------------------------------
       A type can't be its own supertype.
    -----------------------------------------------------*/
       inconsistency(T) :-
              typeName(IdT,Type,T),
              supertype(IdT,IdT,T),
              write("\n\n  IN002: Type ",Type," can't be its own supertype."),
              !.

    /*-----------------------------------------------------
       There can't be redundant supertypes in a type
       definition.
    -----------------------------------------------------*/
       inconsistency(T) :-
              typeSupertypes(IdT,IdTs,T),
              redundantSupertypes(IdTs,T),
              typeName(IdT,Type,T),
              write("\n\n  IN003: Type ",Type," has redundant supertypes."),
              !.

    /*-----------------------------------------------------
       A property can't be defined in two subtypes not
       related by inheritance.
    -----------------------------------------------------*/
       inconsistency(T) :-
              typeProperty(IdT1,IdP,T),
              typeProperty(IdT2,IdP,T),
              IdT1 <> IdT2,
              not(typeInheritance(IdT1,IdT2,T)),
              not(commonTypeProperty(IdT1,IdT2,IdP,T)),

              typeName(IdT1,Type1,T),
              typeName(IdT2,Type2,T),
              propertyName(IdP,Property,T),
              write("\n\n  IN004: Property ",Property," is defined in types ",
                     Type1," and ",Type2,
                     ",\n  but they are not related by inheritance."),
              !.

    /*-----------------------------------------------------
       The domains of the property functions redefined in
       a subtype must be contained within the domains of
       the respective property functions of the supertype.
    -----------------------------------------------------*/
       inconsistency(T) :-
              typePropertySignature(IdT1,IdP,Signature1,T),
              typePropertySignature(IdT2,IdP,Signature2,T),
              IdT1 <> IdT2,
              supertype(IdT2,IdT1,T),
              not(signatureExtrictSubdomains(Signature1,Signature2,T)),

              typeName(IdT1,Type1,T),
              typeName(IdT2,Type2,T),
              propertyName(IdP,Property,T),
              write("\n\n  IN005: Property ",Property," is defined in types ",
                Type1," and ",Type2,",\n  related by inheritance, but the ",
                "respective domains are not extrict\n  subdomains."),
        !.

/*-------------------------------------------------------
   A class can't be its own superclass.
-------------------------------------------------------*/
   inconsistency(T) :-
        className(IdC,Class,T),
        superclass(IdC,IdC,T),
        write("\n\n  IN011: Class ",Class," can't be its own superclass."),
        !.

/*-------------------------------------------------------
   There can't be redundant superclasses in a class
   definition.
-------------------------------------------------------*/
   inconsistency(T) :-
        classSuperclasses(IdC,IdCs,T),
        redundantSuperclasses(IdCs,T),

        className(IdC,Class,T),
        write("\n\n  IN012: Class ",Class," has redundant superclasses."),
        !.

/*-------------------------------------------------------
   All Superclass types of a class have to be
   supertypes or the same type that the one of the
   class.
-------------------------------------------------------*/
   inconsistency(T) :-
        className(IdCobjects,objects,T),
        classType(IdC,IdT,T),
        IdC <> IdCobjects,
        classSuperclasses(IdC,IdCs,T),
        classesTypes(IdCs,IdTs,T),
        not(sameTypeOrSubtype(IdT,IdTs,T)),

        className(IdC,Class,T),
        typeName(IdT,Type,T),
        write("\n\n  IN013: Some superclass of ",Class,
                " has an incompatible type\n  with ",Type,"."),
        !.

/*-------------------------------------------------------
   If two classes share properties, they have to be
   inherited from a common class.
-------------------------------------------------------*/
   inconsistency(T) :-
        classType(IdC1,IdT1,T),
        classType(IdC2,IdT2,T),
        IdC1 <> IdC2,
        typeProperty(IdT1,IdP,T),
        typeProperty(IdT2,IdP,T),
        not(classInheritance(IdC1,IdC2,T)),
        not(commonClassProperty(IdC1,IdC2,IdP,T)),

        className(IdC1,Class1,T),
        className(IdC2,Class2,T),
        propertyName(IdP,Property,T),
        write("\n\n  IN014: Property ",Property," is defined in classes ",
                Class1," and ",Class2,
                ",\n  but they are not related by inheritance."),
        !.

/*-------------------------------------------------------
   Type Closure: "A type specialization hierarchy is
   said to be closed under intersection, if and only
   if for any two types t1, t2, there exists a third
   type t which has exactly all properties common to
   t1 and t2: t = intersection(t1,t2)".
-------------------------------------------------------*/
   inconsistency2(T) :-
        typeName(IdT1,Type1,T),
        typeName(IdT2,Type2,T),
        IdT1 <> IdT2,
        not(definedLowestCommonSupertype(IdT1,IdT2,T)),

        write("\n\n  IN006: Type hierarchy not closed, the lowest common \n",
                "  supertype of ",Type1," and ",Type2," does not exist."),
        !.
```

```
/*----------------------------------------------------
          Class Hierarchy Closure: "A class hierarchy is said
          to be closed under intersection if and only if for
          any two classes C1 and C2, there must exist a third
          class C3 which type is the intersection of C1 and
          C2 types, and its content contents the union of the
          respective contents". C3 = intersection(C1,C2).
-----------------------------------------------------*/
   inconsistency2(T) :-
          className(IdC1,Class1,T),
          className(IdC2,Class2,T),
          IdC1 <> IdC2,
          not(definedLowestCommonSuperclass(IdC1,IdC2,T)),

          write("\n\n  IN015: Class hierarchy not closed, the lowest common \n",
                 "  superclass of ",Class1," and ",Class2," does not exist."),
          !.
```

```
/*****************************************************************
******************************************************************
                             CS_SIGNA.PRO
       Signatures.
******************************************************************
*****************************************************************/

predicates
       subdomain(signature,signature,time)
       extrictSubdomainCondition(signature,signature,time)

       signaturesIntersection(signatures,signature,time)
       domainIntersection(signature,signature,signature,time)
       domainUnion(signature,signature,signature,time)

clauses

   /*---------------------------------------------------
       The first signature domains are extrict subdomains
       of the second signature given ones.
    ---------------------------------------------------*/
       signatureExtrictSubdomains(Signature1,Signature2,T) :-
              subdomain(Signature1,Signature2,T),
              extrictSubdomainCondition(Signature1,Signature2,T),
              !.

   /*---------------------------------------------------
       The first domain is subdomain of the second one.
    ---------------------------------------------------*/
       subdomain([],[],_) :-
              !.
       subdomain([Domain|L1],[Domain|L2],T) :-
              !,
              subdomain(L1,L2,T).
       subdomain([t(D1)|L1],[t(D2)|L2],T) :-
              supertype(D2,D1,T),
              !,
              subdomain(L1,L2,T).
       subdomain([t_(D1)|L1],[t_(D2)|L2],T) :-
              supertype(D2,D1,T),
              !,
              subdomain(L1,L2,T).
       subdomain([t(D1)|L1],[t_(D1)|L2],T) :-
              !,
              subdomain(L1,L2,T).
       subdomain([t(D1)|L1],[t_(D2)|L2],T) :-
              supertype(D2,D1,T),
              !,
              subdomain(L1,L2,T).
       subdomain([c(D1)|L1],[c(D2)|L2],T) :-
              superclass(D2,D1,T),
              !,
              subdomain(L1,L2,T).
       subdomain([c_(D1)|L1],[c_(D2)|L2],T) :-
              superclass(D2,D1,T),
              !,
              subdomain(L1,L2,T).
       subdomain([c(D1)|L1],[c_(D1)|L2],T) :-
              !,
              subdomain(L1,L2,T).
       subdomain([c(D1)|L1],[c_(D2)|L2],T) :-
              superclass(D2,D1,T),
              !,
              subdomain(L1,L2,T).

   /*---------------------------------------------------
       If the first domain is subdomain of the second one,
       and this condition is true, the first subdomain is
       EXTRICT subdomain of the second one.
    ---------------------------------------------------*/
       extrictSubdomainCondition([D|L1],[D|L2],T) :-
              extrictSubdomainCondition(L1,L2,T),
              !.
       extrictSubdomainCondition([t(D1)|_],[t(D2)|_],T) :-
              supertype(D2,D1,T),
              !.
       extrictSubdomainCondition([t_(D1)|_],[t_(D2)|_],T) :-
              supertype(D2,D1,T),
              !.
       extrictSubdomainCondition([t(D1)|_],[t_(D1)|_],_) :-
```

```
            !.
    extrictSubdomainCondition([t(D1)|_],[t_(D2)|_],T) :-
            supertype(D2,D1,T),
            !.
    extrictSubdomainCondition([c(D1)|_],[c(D2)|_],T) :-
            superclass(D2,D1,T),
            !.
    extrictSubdomainCondition([c_(D1)|_],[c_(D2)|_],T) :-
            superclass(D2,D1,T),
            !.
    extrictSubdomainCondition([c(D1)|_],[c_(D1)|_],_) :-
            !.
    extrictSubdomainCondition([c(D1)|_],[c_(D2)|_],T) :-
            superclass(D2,D1,T),
            !.
    extrictSubdomainCondition([_|L1],[_|L2],T) :-
            extrictSubdomainCondition(L1,L2,T).

/*------------------------------------------------------
    Given a set of signatures, obtains the signature
    intersection of them.
 ------------------------------------------------------*/
    signaturesIntersection([Signature],Signature,_) :-
            !.
    signaturesIntersection([Sig1|S],Signature,T) :-
            signaturesIntersection(S,Sig2,T),
            domainIntersection(Sig1,Sig2,Signature,T).

 /*-----------------------------------------------------
    Given two domains, gets the domain intersection
    of them.
 ------------------------------------------------------*/
    domainIntersection([],[],[],_) :-
            !.
    domainIntersection([D|L1],[D|L2],[D|L3],T) :-
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([t(D1)|L1],[t(D2)|L2],[t(D3)|L3],T) :-
            greatestCommonSubtype(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([t_(D1)|L1],[t_(D2)|L2],[t_(D3)|L3],T) :-
            greatestCommonSubtype(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([t(D1)|L1],[t_(D2)|L2],[t(D3)|L3],T) :-
            greatestCommonSubtype(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([t_(D1)|L1],[t(D2)|L2],[t(D3)|L3],T) :-
            greatestCommonSubtype(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([c(D1)|L1],[c(D2)|L2],[c(D3)|L3],T) :-
            greatestCommonSubclass(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([c_(D1)|L1],[c_(D2)|L2],[c_(D3)|L3],T) :-
            greatestCommonSubclass(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([c(D1)|L1],[c_(D2)|L2],[c(D3)|L3],T) :-
            greatestCommonSubclass(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).
    domainIntersection([c_(D1)|L1],[c(D2)|L2],[c(D3)|L3],T) :-
            greatestCommonSubclass(D1,D2,D3,T),
            !,
            domainIntersection(L1,L2,L3,T).

 /*-----------------------------------------------------
    Given two domains, gets the domain union of them.
 ------------------------------------------------------*/
    domainUnion([],[],[],_) :-
            !.
    domainUnion([D|L1],[D|L2],[D|L3],T) :-
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([t(D1)|L1],[t(D2)|L2],[t(D3)|L3],T) :-
            lowestCommonSupertype(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([t_(D1)|L1],[t_(D2)|L2],[t_(D3)|L3],T) :-
            lowestCommonSupertype(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([t(D1)|L1],[t_(D2)|L2],[t_(D3)|L3],T) :-
            lowestCommonSupertype(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([t_(D1)|L1],[t(D2)|L2],[t_(D3)|L3],T) :-
            lowestCommonSupertype(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([c(D1)|L1],[c(D2)|L2],[c(D3)|L3],T) :-
            lowestCommonSuperclass(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([c_(D1)|L1],[c_(D2)|L2],[c_(D3)|L3],T) :-
            lowestCommonSuperclass(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([c(D1)|L1],[c_(D2)|L2],[c_(D3)|L3],T) :-
            lowestCommonSuperclass(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
    domainUnion([c_(D1)|L1],[c(D2)|L2],[c_(D3)|L3],T) :-
            lowestCommonSuperclass(D1,D2,D3,T),
            !,
            domainUnion(L1,L2,L3,T).
```

```
/******************************************************************
*******************************************************************
                             CS_TYPES.PRO
       Types.
*******************************************************************
******************************************************************/

predicates
       deriveTypeProperty(id,time)
       typeProperties(id,id,signature,time)
       typeInheritedProperties(id,id,signature,time)
       inheritedProperty(id,id,time)
       propertySignaturesIntersection(idList,id,signature,time)
       signaturesOfTypes(idList,id,signatures,time)
       typePropertiesDomainUnion(id,id,id,idList,time)
       lowerCommonSupertype(id,id,id,time)
       greaterCommonSubtype(id,id,id,time)


clauses

    /*-----------------------------------------------------
       The first type is supertype of the second one.
    -------------------------------------------------------*/
       supertype(IdT1,IdT2,T) :-
              directSupertype(IdT1,IdT2,T),
              !.
       supertype(IdT1,IdT2,T) :-
              indirectSupertype(IdT1,IdT2,T),
              !.

    /*-----------------------------------------------------
       The first type is direct supertype of the second
       one.
    -------------------------------------------------------*/
       directSupertype(IdT1,IdT2,T) :-
              typeSupertypes(IdT2,IdTs,T),
              memberId(IdT1,IdTs),
              !.

    /*-----------------------------------------------------
       The first type is indirect supertype of the second
       one.
    -------------------------------------------------------*/
       indirectSupertype(IdT1,IdT2,T) :-
              typeSupertypes(IdT2,IdTs,T),
              supertypeOfSomeOne(IdT1,IdTs,T),
              !.

    /*-----------------------------------------------------
       A type is supertype of some one of the types included
       in the list.
    -------------------------------------------------------*/
       supertypeOfSomeOne(IdT1,[IdT2|_],T) :-
              supertype(IdT1,IdT2,T),
              !.
       supertypeOfSomeOne(IdT,[_|L],T) :-
              supertypeOfSomeOne(IdT,L,T).

    /*-----------------------------------------------------
       There is some redundant supertype in a list of types.
    -------------------------------------------------------*/
       redundantSupertypes([Id|L],_) :-
              memberId(Id,L),
              !.
       redundantSupertypes([Id|L],T) :-
              typeInheritanceWithSomeOne(Id,L,T),
              !.
       redundantSupertypes([_|L],T) :-
              redundantSupertypes(L,T).

    /*-----------------------------------------------------
       There is some inheritance relationship between a
       given type and any one of a list.
    -------------------------------------------------------*/
       typeInheritanceWithSomeOne(IdT1,[IdT2|_],T) :-
              typeInheritance(IdT1,IdT2,T),
              !.
       typeInheritanceWithSomeOne(IdT,[_|L],T) :-
              typeInheritanceWithSomeOne(IdT,L,T).
```

```
/*----------------------------------------------------
   There is an inheritance relationship between the
   given types.
------------------------------------------------------*/
   typeInheritance(IdT1,IdT2,T) :-
          supertype(IdT1,IdT2,T),
          !.
   typeInheritance(IdT1,IdT2,T) :-
          supertype(IdT2,IdT1,T),
          !.

/*----------------------------------------------------
   The property is defined in a type common to both
   of the types given.
------------------------------------------------------*/
   commonTypeProperty(IdT1,IdT2,IdP,T) :-
          typeProperty(IdT3,IdP,T),
          IdT1 <> IdT3,
          IdT2 <> IdT3,
          supertype(IdT3,IdT1,T),
          supertype(IdT3,IdT2,T),
          !.

/*----------------------------------------------------
   A type is subtype of all the types of a list, or
   it's the same type.
------------------------------------------------------*/
   sameTypeOrSubtype(IdT,[IdT],_) :-
          !.
   sameTypeOrSubtype(IdT,[IdT2],T) :-
          supertype(IdT2,IdT,T),
          !.
   sameTypeOrSubtype(IdT,[IdT|L],T) :-
          !,
          sameTypeOrSubtype(IdT,L,T).
   sameTypeOrSubtype(IdT,[IdT2|L],T) :-
          supertype(IdT2,IdT,T),
          !,
          sameTypeOrSubtype(IdT,L,T).


/*----------------------------------------------------
   Given a type obtains the list of properties that it
   has.
------------------------------------------------------*/
   deriveTypeProperties(IdT,IdPs,T) :-
          assertzTmpIdList([]),
          deriveTypeProperty(IdT,T),
          retractTmpIdList(IdPs),
          !.

/*----------------------------------------------------
   Given a type stores the list of properties that it
   has.
------------------------------------------------------*/
   deriveTypeProperty(IdT,T) :-
          typeProperties(IdT,IdP,_,T),
          retractTmpIdList(IdPs),
          assertzTmpIdList([IdP|IdPs]),
          fail.
   deriveTypeProperty(_,_) :-
          !.

/*----------------------------------------------------
   Properties of a type (local or inherited).
------------------------------------------------------*/
   typeProperties(IdT,IdP,Signature,T) :-
          typePropertySignature(IdT,IdP,Signature,T).
   typeProperties(IdT,IdP,Signature,T) :-
          typeInheritedProperties(IdT,IdP,Signature,T).

/*----------------------------------------------------
   Inherited properties of a type.
------------------------------------------------------*/
   typeInheritedProperties(IdT,IdP,Signature,T) :-
          typeSupertypes(IdT,IdTs,T),
          propertyName(IdP,_,T),
          inheritedProperty(IdT,IdP,T),
          propertySignaturesIntersection(IdTs,IdP,Signature,T),
          newTypeProperty(IdT,IdP,Signature,T).
```

```
/*----------------------------------------------------
    A property is inherited.
 ------------------------------------------------------*/
    inheritedProperty(IdT,IdP,T) :-
           not(typeProperty(IdT,IdP,T)),
           typeProperty(IdT2,IdP,T),
           supertype(IdT2,IdT,T),
           !.

/*----------------------------------------------------
    Intersection of the signatures of a property for a
    set of types.
 ------------------------------------------------------*/
    propertySignaturesIntersection(IdTs,IdP,Signature,T) :-
           signaturesOfTypes(IdTs,IdP,Signatures,T),
           signaturesIntersection(Signatures,Signature,T).

/*----------------------------------------------------
    Given a set of types and a property, returns the
    set of signatures of this property for the different
    types.
 ------------------------------------------------------*/
    signaturesOfTypes([],_,[],_) :-
           !.
    signaturesOfTypes([IdT|L],IdP,[Signature|S],T) :-
           typeProperties(IdT,IdP,Signature,T),
           signaturesOfTypes(L,IdP,S,T),
           !.
    signaturesOfTypes([_|L],IdP,S,T) :-
           signaturesOfTypes(L,IdP,S,T).

/*---------------------------------------------------
    The lowest common supertype of two given types is
    defined and is supertype of both of them or one of
    them.
 ------------------------------------------------------*/
    definedLowestCommonSupertype(IdT,IdT,_) :-
           !.
    definedLowestCommonSupertype(IdT1,IdT2,T) :-
           supertype(IdT1,IdT2,T),
           !.
    definedLowestCommonSupertype(IdT1,IdT2,T) :-
           supertype(IdT2,IdT1,T),
           !.
    definedLowestCommonSupertype(IdT1,IdT2,T) :-
           typeAllProperties(IdT1,IdPs1,T),
           typeAllProperties(IdT2,IdPs2,T),
           intersectionIdList(IdPs1,IdPs2,IdPsI),
           typeAllProperties(IdT3,IdPs3,T),
           equalIdList(IdPs3,IdPsI),
           supertype(IdT3,IdT1,T),
           supertype(IdT3,IdT2,T),
           typePropertiesDomainUnion(IdT1,IdT2,IdT3,IdPsI,T),
           !.

/*---------------------------------------------------
    The domains of the third type properties are the
    union of the respective property domains of the
    first and second given types.
 ------------------------------------------------------*/
    typePropertiesDomainUnion(_,_,_,[],_) :-
           !.
    typePropertiesDomainUnion(IdT1,IdT2,IdT3,[IdP|L],T) :-
           typePropertySignature(IdT1,IdP,Signature1,T),
           typePropertySignature(IdT2,IdP,Signature2,T),
           typePropertySignature(IdT3,IdP,Signature3,T),
     domainUnion(Signature1,Signature2,Signature3,T),
           typePropertiesDomainUnion(IdT1,IdT2,IdT3,L,T).

/*---------------------------------------------------
    Given two types gets the defined lowest common
    supertype of them.
 ------------------------------------------------------*/
    lowestCommonSupertype(IdT,IdT,IdT3,_) :-
           !,
           IdT3 = IdT.
    lowestCommonSupertype(IdT1,IdT2,IdT3,T) :-
           supertype(IdT1,IdT2,T),
           !,
           IdT3 = IdT1.
    lowestCommonSupertype(IdT1,IdT2,IdT3,T) :-
           supertype(IdT2,IdT1,T),
           !,
           IdT3 = IdT2.
    lowestCommonSupertype(IdT1,IdT2,IdT3,T) :-
           typeName(IdT3,_,T),
           supertype(IdT3,IdT1,T),
           supertype(IdT3,IdT2,T),
           not(lowerCommonSupertype(IdT1,IdT2,IdT3,T)),
           !.

/*-------------------------------------------------------
    Given two types and a proposed lowest common
    supertype, cheks if a lower supertype exists.
 -------------------------------------------------------*/
    lowerCommonSupertype(IdT1,IdT2,IdT3,T) :-
           typeName(IdT4,_,T),
           supertype(IdT4,IdT1,T),
           supertype(IdT4,IdT2,T),
           supertype(IdT3,IdT4,T),
           !.

/*-------------------------------------------------------
    Given two types gets the defined greatest common
    subtype of them.
 -------------------------------------------------------*/
    greatestCommonSubtype(IdT,IdT,IdT3,_) :-
           !,
           IdT3 = IdT.
    greatestCommonSubtype(IdT1,IdT2,IdT3,T) :-
           supertype(IdT2,IdT1,T),
           !,
           IdT3 = IdT1.
    greatestCommonSubtype(IdT1,IdT2,IdT3,T) :-
           supertype(IdT1,IdT2,T),
           !,
           IdT3 = IdT2.
    greatestCommonSubtype(IdT1,IdT2,IdT3,T) :-
           typeName(IdT3,_,T),
           supertype(IdT1,IdT3,T),
           supertype(IdT2,IdT3,T),
           not(greaterCommonSubtype(IdT1,IdT2,IdT3,T)),
           !.

/*-------------------------------------------------------
    Given two types and a proposed greatest common
    subtype, cheks if a greater subtype exists.
 -------------------------------------------------------*/
    greaterCommonSubtype(IdT1,IdT2,IdT3,T) :-
           typeName(IdT4,_,T),
           supertype(IdT1,IdT4,T),
           supertype(IdT2,IdT4,T),
           supertype(IdT4,IdT3,T),
           !.
```

```
/*********************************************************************
**********************************************************************
                             CS_CLASS.PRO
       Classes.
**********************************************************************
*********************************************************************/

predicates
       deriveClassSubclass(id,time)
       lowerCommonSuperclass(id,id,id,time)
       greaterCommonSubclass(id,id,id,time)


clauses

    /*-----------------------------------------------------
       The first class is superclass of the second one.
    -------------------------------------------------------*/
    superclass(IdC,IdC2,T) :-
              directSuperclass(IdC,IdC2,T),
              !.
    superclass(IdC,IdC2,T) :-
              indirectSuperclass(IdC,IdC2,T),
              !.

    /*-----------------------------------------------------
       A class is direct superclass of another.
    -------------------------------------------------------*/
       directSuperclass(IdC1,IdC2,T) :-
              classSuperclasses(IdC2,Superclasses,T),
              memberId(IdC1,Superclasses),
              !.

    /*-----------------------------------------------------
       A class is indirect superclass of another.
    -------------------------------------------------------*/
       indirectSuperclass(IdC1,IdC2,T) :-
              classSuperclasses(IdC2,Superclasses,T),
              superclassOfSomeOne(IdC1,Superclasses,T),
              !.

    /*-----------------------------------------------------
       A class is superclass of some one of the classes
       included in the list.
    -------------------------------------------------------*/
       superclassOfSomeOne(IdC1,[IdC2|_],T) :-
              superclass(IdC1,IdC2,T),
              !.
       superclassOfSomeOne(IdC,[_|L],T) :-
              superclassOfSomeOne(IdC,L,T).

    /*-----------------------------------------------------
       There is some redundant superclass in a list of
       classes.
    -------------------------------------------------------*/
       redundantSuperclasses([IdC|L],_) :-
              memberId(IdC,L),
              !.
       redundantSuperclasses([IdC|L],T) :-
              classInheritanceWithSomeOne(IdC,L,T),
              !.
       redundantSuperclasses([_|L],T) :-
              redundantSuperclasses(L,T).

    /*-----------------------------------------------------
       There is some inheritance relationship between a
       given type and any one of a list.
    -------------------------------------------------------*/
       classInheritanceWithSomeOne(IdC1,[IdC2|_],T) :-
              classInheritance(IdC1,IdC2,T),
              !.
       classInheritanceWithSomeOne(IdC,[_|L],T) :-
              classInheritanceWithSomeOne(IdC,L,T).

    /*-----------------------------------------------------
       There is an inheritance relationship between the
       given classes.
    -------------------------------------------------------*/
       classInheritance(IdC1,IdC2,T) :-
              superclass(IdC1,IdC2,T),
        !.
classInheritance(IdC1,IdC2,T) :-
        superclass(IdC2,IdC1,T),
        !.

/*-----------------------------------------------------
   The property is defined in a class common to both
   of the classes given.
-------------------------------------------------------*/
   commonClassProperty(IdC1,IdC2,IdP,T) :-
        typeProperty(IdT3,IdP,T),
        classType(IdC3,IdT3,T),
        IdC1 <> IdC3,
        IdC2 <> IdC3,
        superclass(IdC3,IdC1,T),
        superclass(IdC3,IdC2,T),
        !.

/*-----------------------------------------------------
   Get the types of the classes given.
-------------------------------------------------------*/
   classesTypes([],[],_) :-
        !.
   classesTypes([IdC|LC],[IdT|LT],T) :-
        classType(IdC,IdT,T),
        classesTypes(LC,LT,T).

/*-----------------------------------------------------
   Given a class obtains the list of its direct
   subclasses.
-------------------------------------------------------*/
   deriveClassSubclasses(IdC,IdDs,T) :-
        assertzTmpIdList([]),
        deriveClassSubclass(IdC,T),
        retractTmpIdList(IdDs),
        !.

/*-----------------------------------------------------
   Given a class stores the list of its direct
   subclasses.
-------------------------------------------------------*/
   deriveClassSubclass(IdC1,T) :-
        classSuperclasses(IdC2,IdCs,T),
        memberId(IdC1,IdCs),
        retractTmpIdList(IdDs),
        assertzTmpIdList([IdC2|IdDs]),
        fail.
   deriveClassSubclass(_,_) :-
        !.

/*-----------------------------------------------------
   The lowest common superclass of two given classes
   is defined and is superclass of both of them or one
   of them.
-------------------------------------------------------*/
   definedLowestCommonSuperclass(IdC,IdC,_) :-
        !.
   definedLowestCommonSuperclass(IdC1,IdC2,T) :-
        superclass(IdC1,IdC2,T),
        !.
   definedLowestCommonSuperclass(IdC1,IdC2,T) :-
        superclass(IdC2,IdC1,T),
        !.
   definedLowestCommonSuperclass(IdC1,IdC2,T) :-
        classType(IdC1,IdT1,T),
        classType(IdC2,IdT2,T),
        lowestCommonSupertype(IdT1,IdT2,IdT3,T),
        classType(IdC3,IdT3,T),
        superclass(IdC3,IdC1,T),
        superclass(IdC3,IdC2,T),
        !.

/*-----------------------------------------------------
   Given two classes gets the defined lowest common
   superclass of them.
-------------------------------------------------------*/
   lowestCommonSuperclass(IdC,IdC,IdC3,_) :-
        !,
        IdC3 = IdC.
   lowestCommonSuperclass(IdC1,IdC2,IdC3,T) :-
        superclass(IdC1,IdC2,T),
        !,
        IdC3 = IdC1.
   lowestCommonSuperclass(IdC1,IdC2,IdC3,T) :-
        superclass(IdC2,IdC1,T),
        !,
        IdC3 = IdC2.
   lowestCommonSuperclass(IdC1,IdC2,IdC3,T) :-
        className(IdC3,_,T),
        superclass(IdC3,IdC1,T),
        superclass(IdC3,IdC2,T),
        not(lowerCommonSuperclass(IdC1,IdC2,IdC3,T)),
        !.

/*----------------------------------------------------
   Given two classes and a proposed lowest common
   superclass, cheks if a lower superclass exists.
----------------------------------------------------*/
   lowerCommonSuperclass(IdC1,IdC2,IdC3,T) :-
        className(IdC4,_,T),
        superclass(IdC4,IdC1,T),
        superclass(IdC4,IdC2,T),
        superclass(IdC3,IdC4,T),
        !.

/*----------------------------------------------------
   Given two classes gets the defined greatest common
   subclass of them.
----------------------------------------------------*/
   greatestCommonSubclass(IdC,IdC,IdC3,_) :-
        !,
        IdC3 = IdC.
   greatestCommonSubclass(IdC1,IdC2,IdC3,T) :-
        superclass(IdC2,IdC1,T),
        !,
        IdC3 = IdC1.
   greatestCommonSubclass(IdC1,IdC2,IdC3,T) :-
        superclass(IdC1,IdC2,T),
        !,
        IdC3 = IdC2.
   greatestCommonSubclass(IdC1,IdC2,IdC3,T) :-
        className(IdC3,_,T),
        superclass(IdC1,IdC3,T),
        superclass(IdC2,IdC3,T),
        not(greaterCommonSubclass(IdC1,IdC2,IdC3,T)),
        !.

/*----------------------------------------------------
   Given two classes and a proposed greatest common
   subclass, cheks if a greater subclass exists.
----------------------------------------------------*/
   greaterCommonSubclass(IdC1,IdC2,IdC3,T) :-
        className(IdC4,_,T),
        superclass(IdC1,IdC4,T),
        superclass(IdC2,IdC4,T),
        superclass(IdC4,IdC3,T),
        !.
```

```
/*********************************************************************
**********************************************************************
                              OOM_SYST.PRO
System primitives.
               generateId(id)
               now(time)
**********************************************************************
*********************************************************************/

domains
    idReg      = id(id)
    timeReg    = time(time)


predicates
       lastId(id)
       newId(id)

       lastTime(time)
       newTime(time)


clauses

    /*=====================================================
       Generate a new identifier.
    =====================================================*/
       generateId(Id) :-
               lastId(Id1),
               Id = Id1 + 1,
               newId(Id),
               !.

    /*-----------------------------------------------------
       Get the last generated Id.
    -----------------------------------------------------*/
       lastId(Id) :-
               openread(idFile,"oomid.dat"),
               readdevice(Old),
               readdevice(idFile),
               readterm(idReg,id(Id)),
               closefile(idFile),
               readdevice(Old),
               !.
       lastId(0).

    /*-----------------------------------------------------
       Store the last generated Id.
    -----------------------------------------------------*/
       newId(Id) :-
               openwrite(idFile,"oomid.dat"),
               writedevice(Old),
               writedevice(idFile),
               write("id(",Id,")\n"),
               closefile(idFile),
               writedevice(Old).

    /*=====================================================
       Get the time.
    =====================================================*/
       now(Time) :-
               lastTime(T1),
               Time = T1 + 1,
               newTime(Time),
               !.

    /*-----------------------------------------------------
       Get the last generated time.
    -----------------------------------------------------*/
       lastTime(T) :-
               openread(timeFile,"oomtime.dat"),
               readdevice(Old),
               readdevice(timeFile),
               readterm(timeReg,time(T)),
               closefile(timeFile),
               readdevice(Old),
               !.
       lastTime(0).
```

```
/*-----------------------------------------------------
   Store the last generated Id.
-------------------------------------------------------*/
   newTime(T) :-
           openwrite(timeFile,"oomtime.dat"),
           writedevice(Old),
           writedevice(timeFile),
           write("time(",T,")\n"),
           closefile(timeFile),
           writedevice(Old).
```

```
/******************************************************************
*******************************************************************
                             OOM_LIST.PRO
       Id lists.
              memberId(id,idList)
              addNewId(idList,id,idList)
*******************************************************************
******************************************************************/

clauses

    /*-----------------------------------------------------
       An Id is member of a list of Ids.
    -------------------------------------------------------*/
       memberId(X,[X|_]) :-
              !.
       memberId(X,[_|L]) :-
              memberId(X,L).

    /*-----------------------------------------------------
       Add a new Id to an Id list only if it is not included
       yet.
    -------------------------------------------------------*/
       addNewId(L,Id,L) :-
              memberId(Id,L),
              !.
       addNewId(L,Id,[Id|L]) :-
              !.

    /*-----------------------------------------------------
       Intersection of two lists of identifiers.
    -------------------------------------------------------*/
       intersectionIdList([],_,[]) :-
              !.
       intersectionIdList([X|L1],L2,[X|L3]) :-
              memberId(X,L2),
              !,
              intersectionIdList(L1,L2,L3).
       intersectionIdList([_|L1],L2,L3) :-
              intersectionIdList(L1,L2,L3).

    /*-----------------------------------------------------
       Equal lists.
    -------------------------------------------------------*/
       equalIdList(L1,L2) :-
              allContainedIdList(L1,L2),
              allContainedIdList(L2,L1).

    /*-----------------------------------------------------
       All the elements of a list are contained in another
       list.
    -------------------------------------------------------*/
       allContainedIdList([],_) :-
              !.
       allContainedIdList([X|L1],L2) :-
              memberId(X,L2),
              allContainedIdList(L1,L2).
```

Appendix B. OODB definition DCM

The following is the source code of the OODB definition DCM developed. It includes a simplified version of the previous DCM.

```
/*******************************************************************
********************************************************************
                             OOSCHE.DEF
       Input file.
********************************************************************
*******************************************************************/


dbbaseproperty(p("address"))
dbbaseproperty(p("name"))
dbbaseproperty(p("category"))
dbbaseclass(c("objects"))
dbbaseclass(c("people"))
dbbaseclass(c("addresses"))
dbbaseclass(c("clients"))
dbbaseclass(c("employees"))
dbdirectclassproperty(c("people"),p("address"))
dbdirectclassproperty(c("employees"),p("category"))
dbdirectclassproperty(c("clients"),p("name"))
dbdirectinheritance(c("people"),c("objects"))
dbdirectinheritance(c("addresses"),c("objects"))
dbdirectinheritance(c("clients"),c("people"))
dbdirectinheritance(c("employees"),c("people"))
dbdirectaggregation(c("people"),c("addresses"),p("address"))


/*******************************************************************
********************************************************************
                             OOINST.DEF
       Input file.
********************************************************************
*******************************************************************/

dbbaseobject(o("o1"))
dbbaseobject(o("o2"))
dbbaseobject(o("o3"))
dbbaseobject(o("o4"))
dbbaseobject(o("o5"))
dbdirectclassobject(c("people"),o("o1"))
dbdirectclassobject(c("clients"),o("o2"))
dbdirectclassobject(c("employees"),o("o2"))
dbdirectclassobject(c("addresses"),o("o3"))
dbdirectclassobject(c("employees"),o("o4"))
dbdirectclassobject(c("addresses"),o("o5"))
dbdirectobjectproperty(o("o1"),p("address"),o([o("o3")]))
dbdirectobjectproperty(o("o2"),p("address"),o([o("o3"),o("o5")]))
dbdirectobjectproperty(o("o4"),p("address"),o([o("o5")]))
dbdirectobjectproperty(o("o2"),p("category"),v([v("boss")]))
dbdirectobjectproperty(o("o4"),p("category"),v([v("technician")]))


/*******************************************************************
********************************************************************
                             OODERI.DEF
       Input file.
********************************************************************
*******************************************************************/

dbdefinedderivedclass(c("employees_"))
dbdirectderivation(c("employees_"),[c("employees")],preservation)
dbdirectclassproperty(c("employees_"),p("name"))
dbdirectclassproperty(c("employees_"),p("address"))
dbdirectaggregation(c("employees_"),c("addresses"),p("address"))
```

```
/*******************************************************************
********************************************************************
                             OOSELE.DEF
       Input file.
********************************************************************
*******************************************************************/

dbclasssetselection("es1",[c("objects"),c("clients"),c("employees_")])
dbqualifiedclasssetselection("es2",[q(esc(c("objects"),nontransformable)),q(esc(c("clien
ts"),transformable)),q(esc(c("employees_"),nontransformable)),q(esc(c("addresses"),trans
formable)),q(esc(c("people"),transformable))])
dbqualifiedclasssetselection("es3",[q(esc(c("objects"),nontransformable)),q(esc(c("clien
ts"),transformable)),q(esc(c("employees_"),transformable)),q(esc(c("addresses"),transfor
mable)),q(esc(c("people"),nontransformable))])
```

```
/*******************************************************************
********************************************************************
                              OODB.DAT
       Output file.
********************************************************************
*******************************************************************/

rbaseproperty(p("address"),1)
rbaseproperty(p("name"),2)
rbaseproperty(p("category"),3)
rbaseclass(c("objects"),4)
rbaseclass(c("people"),5)
rbaseclass(c("addresses"),6)
rbaseclass(c("clients"),7)
rbaseclass(c("employees"),8)
rdirectclassproperty(c("people"),p("address"),9)
rdirectclassproperty(c("employees"),p("category"),10)
rdirectclassproperty(c("clients"),p("name"),11)
rdirectclassproperty(c("employees_"),p("name"),12)
rdirectclassproperty(c("employees_"),p("address"),13)
rdirectinheritance(c("people"),c("objects"),14)
rdirectinheritance(c("addresses"),c("objects"),15)
rdirectinheritance(c("clients"),c("people"),16)
rdirectinheritance(c("employees"),c("people"),17)
rdirectaggregation(c("people"),c("addresses"),p("address"),18)
rdirectaggregation(c("employees_"),c("addresses"),p("address"),19)
rbaseobject(o("o1"),20)
rbaseobject(o("o2"),21)
rbaseobject(o("o3"),22)
rbaseobject(o("o4"),23)
rbaseobject(o("o5"),24)
rdirectclassobject(c("people"),o("o1"),25)
rdirectclassobject(c("clients"),o("o2"),26)
rdirectclassobject(c("employees"),o("o2"),27)
rdirectclassobject(c("addresses"),o("o3"),28)
rdirectclassobject(c("employees"),o("o4"),29)
rdirectclassobject(c("addresses"),o("o5"),30)
rdirectobjectproperty(o("o1"),p("address"),o([o("o3")]),31)
rdirectobjectproperty(o("o2"),p("address"),o([o("o3"),o("o5")]),32)
rdirectobjectproperty(o("o4"),p("address"),o([o("o5")]),33)
rdirectobjectproperty(o("o2"),p("category"),v([v("boss")]),34)
rdirectobjectproperty(o("o4"),p("category"),v([v("technician")]),35)
rdefinedderivedclass(c("employees_"),36)
rdirectderivation(c("employees_"),[c("employees")],preservation,37)
rclasssetselection("es1",[c("objects"),c("clients"),c("employees_")],38)
rqualifiedclasssetselection("es2",[q(esc(c("objects"),nontransformable)),q(esc(c("client
s"),transformable)),q(esc(c("employees_"),nontransformable)),q(esc(c("addresses"),transf
ormable)),q(esc(c("people"),transformable))],39)
rqualifiedclasssetselection("es3",[q(esc(c("objects"),nontransformable)),q(esc(c("client
s"),transformable)),q(esc(c("employees_"),transformable)),q(esc(c("addresses"),transform
able)),q(esc(c("people"),nontransformable))],40)
rindirectinheritance(c("clients"),c("objects"),41)
rindirectinheritance(c("employees"),c("objects"),42)
rinheritedaggregation(c("clients"),c("addresses"),p("address"),43)
rinheritedaggregation(c("employees"),c("addresses"),p("address"),44)
rallclassproperties(c("objects"),[],45)
rallclassproperties(c("people"),[p("address")],46)
rallclassproperties(c("addresses"),[],47)
rallclassproperties(c("clients"),[p("name"),p("address")],48)
rallclassproperties(c("employees"),[p("category"),p("address")],49)
rallclassproperties(c("employees_"),[p("name"),p("address")],50)
rallclassobjects(c("objects"),[o("o1"),o("o2"),o("o3"),o("o4"),o("o5")],51)
rallclassobjects(c("people"),[o("o1"),o("o2"),o("o4")],52)
rallclassobjects(c("addresses"),[o("o3"),o("o5")],53)
rallclassobjects(c("clients"),[o("o2")],54)
rallclassobjects(c("employees"),[o("o2"),o("o4")],55)
rallclassobjects(c("employees_"),[o("o4")],56)
rtime(56)
```

```
%*******************************************************************
%*******************************************************************
%       Domains.
%*******************************************************************
%*******************************************************************

domains

        comparation    = eq;
                         ne

        element        = p(property);
                         c(class);
                         o(object);
                         v(value);
                         e(edge);
                         q(esClass);
                         cp(element,elements);      %Class with properties.
                         si(element,elements,elements)
                              %Subsumption Isomorfic classes with properties.

        elements       = element*
        difList        = dl(elements,elements)

        property       = symbol
        class          = symbol
        object         = symbol
        value          = symbol
        schema         = symbol
        edge           = reference is_a(element,element)
        esClass        = esc(element,classQuality)  % External Schema Class

        classQuality   = transformable;
                         nonTransformable;
                         any

        evaluation     = o(elements);
                         v(elements)

        derSemantics   = preservation;
                         generation

        time           = integer
```

```
/*******************************************************************
********************************************************************
                              OODB.PRO
********************************************************************
*******************************************************************/

include "domains.pro"
include "elemlist.pro"


predicates
                      run

                      addTime
                      now(time)

                      inheritanceInconsistency
       nondeterm      inheritanceInconsistency(symbol)
                      findIndirectInheritance(element)
       nondeterm      thereIsInheritance(element,element)
       nondeterm      loopInInheritance(element,elements)
       nondeterm      inheritance(element,element)

                      classInconsistency
       nondeterm      classInconsistency(symbol)
                      findClassProperties
       nondeterm      hasClassProperty(element,element)
       nondeterm      superclassWithProperties(element,element,elements,element)
       nondeterm      class(element)
                      compareClasses(element,element,comparation)
       nondeterm      classProperties(element,elements)
       nondeterm      classProperty(element,element)
       nondeterm      classPropertiesIntersection(element,element,elements)

                      objectInconsistency
       nondeterm      objectInconsistency(symbol)
                      findClassObjects
       nondeterm      containsClassObject(element,element)
       nondeterm      objectClassProperty(element,element,element)
                      objects(elements)
                      values(elements)
       nondeterm      classObjects(element,elements)
                      subsumesExtension(element,element)

                      aggregationInconsistency
       nondeterm      aggregationInconsistency(symbol)
                      findInheritedAggregation
       nondeterm      thereIsAggregation(element,element,element)
       nondeterm      refinedAggregation(element,element,element)
       nondeterm      aggregation(element,element,element)

                      derivationInconsistency
       nondeterm      derivationInconsistency(symbol)
                      findDerivedClassObjects
                      classes(elements)
       nondeterm      loopInDerivation(element,elements)

                      classSetInconsistency
       nondeterm      classSetInconsistency(symbol)

                      qualifiedClassSetInconsistency
       nondeterm      qualifiedClassSetInconsistency(symbol)
                      rightQualifiedClasses(elements)
                      qualifiedClasses(elements,classQuality,elements)

       nondeterm      characteristicFunction(element,element)
```

```
database - oodb

        dbBaseProperty(element)
        dbBaseClass(element)
        dbDirectClassProperty(element,element)
        dbDirectInheritance(element,element)
        dbDirectAggregation(element,element,element)
        dbBaseObject(element)
        dbDirectClassObject(element,element)
        dbDirectObjectProperty(element,element,evaluation)
        dbDefinedDerivedClass(element)
        dbDirectDerivation(element,elements,derSemantics)
        dbGeneratedDerivedClass(element)
        dbClassSetSelection(schema,elements)
        dbQualifiedClassSetSelection(schema,elements)
        dbExternalSchema(schema,elements,elements)

        dbIndirectInheritance(element,element)
        dbInheritedAggregation(element,element,element)
        dbAllClassProperties(element,elements)
        dbAllClassObjects(element,elements)


database - roodb

        rBaseProperty(element,time)
        rBaseClass(element,time)
        rDirectClassProperty(element,element,time)
        rDirectInheritance(element,element,time)
        rDirectAggregation(element,element,element,time)
        rBaseObject(element,time)
        rDirectClassObject(element,element,time)
        rDirectObjectProperty(element,element,evaluation,time)
        rDefinedDerivedClass(element,time)
        rDirectDerivation(element,elements,derSemantics,time)
        rGeneratedDerivedClass(element,time)
        rClassSetSelection(schema,elements,time)
        rQualifiedClassSetSelection(schema,elements,time)
        rExternalSchema(schema,elements,elements,time)

        rIndirectInheritance(element,element,time)
        rInheritedAggregation(element,element,element,time)
        rAllClassProperties(element,elements,time)
        rAllClassObjects(element,elements,time)

        rTime(integer)


goal
        run.


clauses

        run :-
                consult("oosche.def",oodb),
                consult("ooinst.def",oodb),
                consult("ooderi.def",oodb),
                consult("oosele.def",oodb),

                not(inheritanceInconsistency),
                not(classInconsistency),
                not(aggregationInconsistency),
                not(objectInconsistency),
                not(derivationInconsistency),
                not(classSetInconsistency),
                not(qualifiedClassSetInconsistency),

                addTime,
                save("oodb.dat",roodb),
                write("\n\nOK."),
                !.
```

```
addTime :-
        dbBaseProperty(P),
        now(T),
        assertz(rBaseProperty(P,T)),
        fail.
addTime :-
        dbBaseClass(C),
        now(T),
        assertz(rBaseClass(C,T)),
        fail.
addTime :-
        dbDirectClassProperty(C,P),
        now(T),
        assertz(rDirectClassProperty(C,P,T)),
        fail.
addTime :-
        dbDirectInheritance(C1,C2),
        now(T),
        assertz(rDirectInheritance(C1,C2,T)),
        fail.
addTime :-
        dbDirectAggregation(C1,C2,P),
        now(T),
        assertz(rDirectAggregation(C1,C2,P,T)),
        fail.
addTime :-
        dbBaseObject(O),
        now(T),
        assertz(rBaseObject(O,T)),
        fail.
addTime :-
        dbDirectClassObject(C,O),
        now(T),
        assertz(rDirectClassObject(C,O,T)),
        fail.
addTime :-
        dbDirectObjectProperty(O,P,E),
        now(T),
        assertz(rDirectObjectProperty(O,P,E,T)),
        fail.
addTime :-
        dbDefinedDerivedClass(C),
        now(T),
        assertz(rDefinedDerivedClass(C,T)),
        fail.
addTime :-
        dbDirectDerivation(C,Cs,S),
        now(T),
        assertz(rDirectDerivation(C,Cs,S,T)),
        fail.
addTime :-
        dbGeneratedDerivedClass(C),
        now(T),
        assertz(rGeneratedDerivedClass(C,T)),
        fail.
addTime :-
        dbClassSetSelection(S,Cs),
        now(T),
        assertz(rClassSetSelection(S,Cs,T)),
        fail.
addTime :-
        dbQualifiedClassSetSelection(S,Cs),
        now(T),
        assertz(rQualifiedClassSetSelection(S,Cs,T)),
        fail.
addTime :-
        dbExternalSchema(S,Cs,Es),
        now(T),
        assertz(rExternalSchema(S,Cs,Es,T)),
        fail.
addTime :-
        dbIndirectInheritance(C1,C2),
        now(T),
        assertz(rIndirectInheritance(C1,C2,T)),
        fail.
addTime :-
        dbInheritedAggregation(C1,C2,P),
        now(T),
        assertz(rInheritedAggregation(C1,C2,P,T)),
        fail.
        addTime :-
                dbAllClassProperties(C,Ps),
                now(T),
                assertz(rAllClassProperties(C,Ps,T)),
                fail.
        addTime :-
                dbAllClassObjects(C,Os),
                now(T),
                assertz(rAllClassObjects(C,Os,T)),
                fail.
        addTime :-
                !.


        %*****************************************************
        % now(T) :- Obtain the time.
        %
        now(T) :-
                retract(rTime(T1)),
                !,
                T = T1 + 1,
                assertz(rTime(T)).
        now(1) :-
                !,
                assertz(rTime(1)).


%*******************************************************************
%       Inheritance.
%*******************************************************************

        %*****************************************************
        % inheritanceInconsistency(N) :- Inheritance
        %       inconsistencies in the repository.
        %
        inheritanceInconsistency :-
                inheritanceInconsistency(N),
                write("\nInheritance Inconsistency: ",N,"\n"),
                !.
        inheritanceInconsistency(a01) :-
                dbDirectInheritance(C,_),
                not(dbBaseClass(C)).
        inheritanceInconsistency(a02) :-
                dbDirectInheritance(_,C),
                not(dbBaseClass(C)).
        inheritanceInconsistency(a03) :-
                dbBaseClass(C),
                dbDirectInheritance(C,C).
        inheritanceInconsistency(a04) :-
                dbDirectInheritance(C1,C2),
                loopInInheritance(C2,[C1]).
        inheritanceInconsistency(a05) :-
                dbBaseClass(C1),
                findIndirectInheritance(C1),
                dbDirectInheritance(C1,C2),
                dbIndirectInheritance(C1,C2).
        inheritanceInconsistency(a06) :-
                not(dbBaseClass(c(objects))).
        inheritanceInconsistency(a07) :-
                dbBaseClass(C),
                compareClasses(C,c(objects),ne),
                not(inheritance(C,c(objects))).


        %*****************************************************
        % findIndirectInheritance :- Find indirect inheritance
        %       relationships.
        %
        findIndirectInheritance(C1) :-
                dbBaseClass(C1),
                dbBaseClass(C2),
                compareClasses(C1,c(objects),ne),
                compareClasses(C1,C2,ne),
                not(dbDirectInheritance(C2,C1)),
                not(dbIndirectInheritance(C2,C1)),
                dbDirectInheritance(C1,C3),
                thereIsInheritance(C3,C2),
                not(dbIndirectInheritance(C1,C2)),
                assertz(dbIndirectInheritance(C1,C2)),
                fail.
        findIndirectInheritance(_) :-
```

```
       !.


%****************************************************
% thereIsInheritance(Class1,Class2) :- Class1 and
%      Class2 are related by inheritance.
%
thereIsInheritance(C1,C2) :-
       dbDirectInheritance(C1,C2).
thereIsInheritance(C1,C2) :-
       dbDirectInheritance(C1,C3),
       thereIsInheritance(C3,C2).


%****************************************************
% loopInInheritance(Class,Classes) :- There is a loop
%      in inheritance relationships: we have arrived to
%      Class visiting Classes.
%
loopInInheritance(C,Cs) :-
       includedElement(C,Cs).
loopInInheritance(C1,Cs) :-
       dbDirectInheritance(C1,C2),
       loopInInheritance(C2,[C1|Cs]).


%****************************************************
% inheritance(Class1,Class2) :- Class1 is subclass of
%      Class2.
%
inheritance(C1,C2) :-
       dbDirectInheritance(C1,C2).
inheritance(C1,C2) :-
       dbIndirectInheritance(C1,C2).
```

```
%******************************************************************
%      Classes.
%******************************************************************

       %****************************************************
       % classInconsistency(N) :- Class inconsistencies in the
       %      repository.
       %
       classInconsistency :-
              classInconsistency(N),
              write("\nClass Inconsistency: ",N,"\n"),
              !.
       classInconsistency(b01) :-
              dbDirectClassProperty(C,_),
              not(class(C)).
       classInconsistency(b02) :-
              dbDirectClassProperty(_,P),
              not(dbBaseProperty(P)).
       classInconsistency(b03) :-
              dbBaseClass(C1),
              dbBaseClass(C2),
              compareClasses(C1,C2,ne),
              dbDirectClassProperty(C1,P),
              dbDirectClassProperty(C2,P).
       classInconsistency(b04) :-
              findClassProperties,
              dbBaseClass(C1),
              dbBaseClass(C2),
              compareClasses(C1,C2,ne),
              not(inheritance(C1,C2)),
              not(inheritance(C2,C1)),
              classPropertiesIntersection(C1,C2,Ps),
              not(superclassWithProperties(C1,C2,Ps,_)).


       %****************************************************
       % findClassProperties :- Find all the properties of
       %      classes.
       %
       findClassProperties :-
              class(C),
              findall(P,hasClassProperty(C,P),Ps2),
              nonDuplicatedElements(Ps2,Ps1),
              assertz(dbAllClassProperties(C,Ps1)),
              fail.
```

```
findClassProperties :-
        !.


%****************************************************
% hasClassProperty(Class,Property) :- Class has Property.
%
hasClassProperty(C,P) :-
        dbDirectClassProperty(C,P).
hasClassProperty(C1,P) :-
        dbDirectClassProperty(C2,P),
        inheritance(C1,C2).


%****************************************************
% superclassWithProperties(Class1,Class2,Properties,Class3) :-
%       Class3 is superclass of Class1 and Class2 and has
%       Properties.
%
superclassWithProperties(C1,C2,Ps,C3) :-
        inheritance(C1,C3),
        inheritance(C2,C3),
        classProperties(C3,Ps).


%****************************************************
% class(Class) :- Class is a class.
%
class(C) :-
        dbBaseClass(C).
class(C) :-
        dbDefinedDerivedClass(C).
class(C) :-
        dbGeneratedDerivedClass(C).


%****************************************************
% compareClasses(C1,C2,Comparation) :- Comparation
%       between two classes.
%
compareClasses(c(C1),c(C2),eq) :-
        C1 = C2,
        !.
compareClasses(c(C1),c(C2),ne) :-
        C1 <> C2,
        !.


%****************************************************
% classProperties(Class,Properties) :- Class has
%       Properties.
%
classProperties(C,Ps) :-
        class(C),
        not(bound(Ps)),
        dbAllClassProperties(C,Ps).
classProperties(C,Ps1) :-
        class(C),
        bound(Ps1),
        dbAllClassProperties(C,Ps2),
        equalElements(Ps2,Ps1).


%****************************************************
% classProperty(Class,Property) :- Class has Property.
%
classProperty(C,P) :-
        dbAllClassProperties(C,Ps),
        includedElement(P,Ps).


%****************************************************
% classPropertiesIntersection(Class1,Class2,Properties) :-
%       Class1 and Class2 have Properties in common.
%
classPropertiesIntersection(C1,C2,Ps) :-
        dbAllClassProperties(C1,Ps1),
        dbAllClassProperties(C2,Ps2),
        elementsIntersection(Ps1,Ps2,Ps).
```

```
%*******************************************************************
%      Aggregation.
%*******************************************************************

       %****************************************************
       % aggregationInconsistency(N) :- Aggregation
       %      inconsistencies in the repository.
       %
       aggregationInconsistency :-
              aggregationInconsistency(N),
              write("\nAggregation Inconsistency: ",N,"\n"),
              !.
       aggregationInconsistency(c01) :-
              dbDirectAggregation(C,_,_),
              not(class(C)).
       aggregationInconsistency(c02) :-
              dbDirectAggregation(_,C,_),
              not(class(C)).
       aggregationInconsistency(c03) :-
              dbDirectAggregation(C,_,P),
              not(classProperty(C,P)).
       aggregationInconsistency(c04) :-
              dbDirectClassProperty(C1,P),
              dbDirectAggregation(C2,_,P),
              compareClasses(C1,C2,ne),
              not(dbDirectAggregation(C1,_,P)).
       aggregationInconsistency(c05) :-
              dbDirectAggregation(C1,C2,P),
              dbDirectAggregation(C3,C4,P),
              compareClasses(C1,C3,ne),
              compareClasses(C2,C4,ne),
              inheritance(C3,C1),
              not(inheritance(C4,C2)).
       aggregationInconsistency(c06) :-
              dbDirectAggregation(C1,C2,P),
              dbDirectAggregation(C1,C3,P),
              compareClasses(C2,C3,ne).
       aggregationInconsistency(c07) :-
              findInheritedAggregation,
              aggregation(C1,C2,P),
              aggregation(C1,C3,P),
              compareClasses(C2,C3,ne).


       %****************************************************
       % findInheritedAggregation :- Find inherited
       %      aggregation relationships.
       %
       findInheritedAggregation :-
              dbBaseClass(C1),
              dbDirectAggregation(_,C2,P),
              not(dbDirectAggregation(C1,C2,P)),
              not(dbInheritedAggregation(C1,C2,P)),
              thereIsAggregation(C1,C2,P),
              not(refinedAggregation(C1,C2,P)),
              assertz(dbInheritedAggregation(C1,C2,P)),
              fail.
       findInheritedAggregation :-
              !.


       %****************************************************
       % thereIsAggregation(Class1,Class2,Property) :- There
       %      is an aggregation relationship between Class1 and
       %      Class2 in Property.
       %
       thereIsAggregation(C1,C2,P) :-
              dbDirectAggregation(C1,C2,P).
       thereIsAggregation(C1,C2,P) :-
              inheritance(C1,C3),
              dbDirectAggregation(C3,C2,P).


       %****************************************************
       % refinedAggregation(Class1,Class2,Property) :- The
       %      aggregation relationship between Class1 y Class2
       %      has been redefined.
       %
       refinedAggregation(C1,C2,P) :-
              inheritance(C3,C2),
              thereIsAggregation(C1,C3,P).
```

```
        %****************************************************
        % aggregation(Class1,Class2,Property) :- There is a
        %       property function defined for class Class1 with the
        %       property label Property and the domain class Class2.
        %
        aggregation(C1,C2,P) :-
               dbDirectAggregation(C1,C2,P).
        aggregation(C1,C2,P) :-
               dbInheritedAggregation(C1,C2,P).


%*******************************************************************
%       Objects.
%*******************************************************************

        %****************************************************
        % objectInconsistency(N) :- Object inconsistencies in
        %       the repository.
        %
        objectInconsistency :-
               objectInconsistency(N),
               write("\nObject Inconsistency: ",N,"\n"),
               !.
        objectInconsistency(d01) :-
               dbDirectClassObject(C,_),
               not(dbBaseClass(C)).
        objectInconsistency(d02) :-
               dbDirectClassObject(_,O),
               not(dbBaseObject(O)).
        objectInconsistency(d03) :-
               dbBaseObject(O),
               not(dbDirectClassObject(_,O)).
        objectInconsistency(d04) :-
               dbDirectClassObject(C1,O),
               dbDirectClassObject(C2,O),
               compareClasses(C1,C2,ne),
               inheritance(C1,C2).
        objectInconsistency(d05) :-
               dbDirectObjectProperty(O,_,_),
               not(dbBaseObject(O)).
        objectInconsistency(d06) :-
               dbDirectObjectProperty(_,P,_),
               not(dbBaseProperty(P)).
        objectInconsistency(d07) :-
               dbDirectObjectProperty(_,_,o(Os)),
               not(objects(Os)).
        objectInconsistency(d08) :-
               dbDirectObjectProperty(_,_,v(Vs)),
               not(values(Vs)).
        objectInconsistency(d09) :-
               dbDirectObjectProperty(O,P,v(Vs1)),
               dbDirectObjectProperty(O,P,v(Vs2)),
               not(equalElements(Vs1,Vs2)).
        objectInconsistency(d10) :-
               dbDirectObjectProperty(O,P,o(Os1)),
               dbDirectObjectProperty(O,P,o(Os2)),
               not(equalElements(Os1,Os2)).
        objectInconsistency(d11) :-
               dbDirectObjectProperty(O,P,o(_)),
               dbDirectObjectProperty(O,P,v(_)).
        objectInconsistency(d12) :-
               dbDirectObjectProperty(O,P,_),
               not(objectClassProperty(O,_,P)).
        objectInconsistency(d13) :-
               dbDirectObjectProperty(O,P,o(_)),
               dbDirectClassObject(C,O),
               classProperty(C,P),
               not(aggregation(C,_,P)).
        objectInconsistency(d14) :-
               dbDirectObjectProperty(O,P,v(_)),
               dbDirectClassObject(C,O),
               classProperty(C,P),
               aggregation(C,_,P).
        objectInconsistency(d15) :-
               findClassObjects,
               dbDirectObjectProperty(O,P,o(Os1)),
               dbDirectClassObject(C1,O),
               classProperty(C1,P),
               aggregation(C1,C2,P),
               classObjects(C2,Os2),
               not(includedElements(Os1,Os2)).


       %*****************************************************
       % findClassObjects :- Find all the objects of classes.
       %
       findClassObjects :-
               dbBaseClass(C),
               findall(O,containsClassObject(C,O),Os2),
               nonDuplicatedElements(Os2,Os1),
               assertz(dbAllClassObjects(C,Os1)),
               fail.
       findClassObjects :-
               !.


       %*****************************************************
       % containsClassObject(Class,Object) :- Object is member
       %       of Class.
       %
       containsClassObject(C,O) :-
               dbDirectClassObject(C,O).
       containsClassObject(C1,O) :-
               dbDirectClassObject(C2,O),
               inheritance(C2,C1).


       %*****************************************************
       % objectClassProperty(Object,Class,Property) :- Object
       %       is member of Class that has Property defined.
       %
       objectClassProperty(O,C,P) :-
               dbDirectClassObject(C,O),
               classProperty(C,P).


       %*****************************************************
       % objects(Objects) :- Objects is a set of objects.
       %
       objects([O|Os]) :-
               dbBaseObject(O),
               !,
               objects(Os).
       objects([]) :-
               !.


       %*****************************************************
       % values(Values) :- Values is a set of values.
       %
       values([v(_)|Vs]) :-
               !,
               values(Vs).
       values([]) :-
               !.


       %*****************************************************
       % classObjects(Class,Objects) :- Class has Objects as
       %       members.
       %
       classObjects(C,Os) :-
               class(C),
               not(bound(Os)),
               dbAllClassObjects(C,Os).
       classObjects(C,Os1) :-
               class(C),
               bound(Os1),
               dbAllClassObjects(C,Os2),
               equalElements(Os2,Os1).


       %*****************************************************
       % subsumesExtension(Class1,Class2) :- Class1 subsumes
       %       extesion of Class2. APROXIMACIÓN INCORRECTA.
       %
       subsumesExtension(C1,C2) :-
               classObjects(C1,Os1),
               classObjects(C2,Os2),
               includedElements(Os2,Os1),
        !.


%*******************************************************************
%       Derivation.
%*******************************************************************

        %****************************************************
        % derivationInconsistency(N) :- Derivation
        %       inconsistencies in the repository.
        %
        derivationInconsistency :-
                derivationInconsistency(N),
                write("\nDerivation Inconsistency: ",N,"\n"),
                !.
        derivationInconsistency(e01) :-
                dbDirectDerivation(C,_,_),
                not(dbDefinedDerivedClass(C)).
        derivationInconsistency(e02) :-
                dbDefinedDerivedClass(C),
                not(dbDirectDerivation(C,_,_)).
        derivationInconsistency(e03) :-
                dbDirectDerivation(_,[],_).
        derivationInconsistency(e04) :-
                dbDirectDerivation(_,Cs,_),
                not(classes(Cs)).
        derivationInconsistency(e05) :-
                dbDefinedDerivedClass(C),
                dbBaseClass(C).
        derivationInconsistency(e06) :-
                dbDirectDerivation(C,Cs,_),
                includedElement(C,Cs).
        derivationInconsistency(e07) :-
                dbDirectDerivation(C,Cs,_),
                loopInDerivation(C,Cs).
        derivationInconsistency(e08) :-
                findDerivedClassObjects,
                dbDefinedDerivedClass(C1),
                class(C2),
                compareClasses(C1,C2,ne),
                dbAllClassProperties(C1,Ps1),
                dbAllClassProperties(C2,Ps2),
                equalElements(Ps1,Ps2),
                subsumesExtension(C1,C2),
                subsumesExtension(C2,C1).


        %****************************************************
        % findDerivedClassObjects :- Definition predicates of
        %       derived classes.
        %
        findDerivedClassObjects :-
                dbDefinedDerivedClass(C),
                findall(O,characteristicFunction(C,O),Os),
                assertz(dbAllClassObjects(C,Os)),
                fail.
        findDerivedClassObjects :-
                !.


        %****************************************************
        % classes(Classes) :- Classes is a set of classes.
        %
        classes([C|Cs]) :-
                class(C),
                !,
                classes(Cs).
        classes([]) :-
                !.


        %****************************************************
        % loopInDerivation(Class,Classes) :- Class is derived
        %       from a class in Classes which also is derived from
        %       Class.
        %
        loopInDerivation(C1,Cs1) :-
                includedElement(C2,Cs1),
                dbDirectDerivation(C2,Cs2,_),
                includedElement(C1,Cs2),
                !.
```

```
       loopInDerivation(C1,Cs1) :-
              includedElement(C2,Cs1),
              dbDirectDerivation(C2,Cs2,_),
              loopInDerivation(C1,Cs2).


%*******************************************************************
%      External Schema Class Set Selection.
%*******************************************************************

       %****************************************************
       % classSetInconsistency(N) :- Class set selection
       %      inconsistencies in the repository.
       %
       classSetInconsistency :-
              classSetInconsistency(N),
              write("\nClass Set Inconsistency: ",N,"\n"),
              !.
       classSetInconsistency(f01) :-
              dbClassSetSelection(_,[]).
       classSetInconsistency(f02) :-
              dbClassSetSelection(_,Cs),
              not(classes(Cs)).
       classSetInconsistency(f03) :-
              dbClassSetSelection(_,Cs),
              repeatedElement(_,Cs).
       classSetInconsistency(f04) :-
              dbClassSetSelection(S,Cs1),
              dbClassSetSelection(S,Cs2),
              not(equalElements(Cs1,Cs2)).


%*******************************************************************
%      External Schema Qualified Class Set Selection.
%*******************************************************************

       %****************************************************
       % qualifiedClassSetInconsistency(N) :- Qualified class
       %      set selection inconsistencies in the repository.
       %
       qualifiedClassSetInconsistency :-
              qualifiedClassSetInconsistency(N),
              write("\nQualified Class Set Inconsistency: ",N,"\n"),
              !.
       qualifiedClassSetInconsistency(g01) :-
              dbQualifiedClassSetSelection(_,[]).
       qualifiedClassSetInconsistency(g02) :-
              dbQualifiedClassSetSelection(_,QCs),
              qualifiedClasses(QCs,any,Cs),
              not(classes(Cs)).
       qualifiedClassSetInconsistency(g03) :-
              dbQualifiedClassSetSelection(_,QCs),
              qualifiedClasses(QCs,any,Cs),
              repeatedElement(_,Cs).
       qualifiedClassSetInconsistency(g04) :-
              dbQualifiedClassSetSelection(S,QCs1),
              dbQualifiedClassSetSelection(S,QCs2),
              not(equalElements(QCs1,QCs2)).
       qualifiedClassSetInconsistency(g05) :-
              dbClassSetSelection(S,_),
              dbQualifiedClassSetSelection(S,_).
       qualifiedClassSetInconsistency(g06) :-
              dbQualifiedClassSetSelection(_,QCs),
              not(rightQualifiedClasses(QCs)).
       qualifiedClassSetInconsistency(g07) :-
              dbQualifiedClassSetSelection(_,QCs),
              includedElement(q(esc(c(objects),transformable)),QCs).


       %****************************************************
       % rightQualifiedClasses(QCs) :- Given a set of qualified
       %      classes, checks if they are righly qualified.
       %
       rightQualifiedClasses([q(esc(_,transformable))|QCs]) :-
              !,
              rightQualifiedClasses(QCs).
       rightQualifiedClasses([q(esc(_,nonTransformable))|QCs]) :-
              !,
              rightQualifiedClasses(QCs).
       rightQualifiedClasses([]) :-
              !.
```

```
        %*****************************************************
        % qualifiedClasses(QCs,Q,Cs) :- Given a set of qualified
        %       classes and a qualification, obtains the set of
        %       classes qualified this way.
        %
        qualifiedClasses([q(esc(C,_))|QCs],any,[C|Cs]) :-
                !,
                qualifiedClasses(QCs,any,Cs).
        qualifiedClasses([q(esc(C,Q))|QCs],Q,[C|Cs]) :-
                !,
                qualifiedClasses(QCs,Q,Cs).
        qualifiedClasses([_|QCs],Q,Cs) :-
                !,
                qualifiedClasses(QCs,Q,Cs).
        qualifiedClasses([],_,[]) :-
                !.


%*******************************************************************
%       Characteristic Functions.
%*******************************************************************

        %*****************************************************
        % characteristicFunction(Class,Object) :- Characteristic
        %       function of a class.
        %
        characteristicFunction(c(employees_),O) :-
                dbDefinedDerivedClass(c(employees_)),
                dbDirectDerivation(c(employees_),[c(employees)],preservation),
                dbDirectClassObject(c(employees),O),
                dbDirectObjectProperty(O,p(category),v([v(Category)])),
                Category <> "boss".
```

```
%*****************************************************************
%*****************************************************************
%                            ELEMLIST.PRO
%       List of Elements.
%*****************************************************************
%*****************************************************************

predicates
       nondeterm      includedElement(element,elements)
                      includedElements(elements,elements)
                      includeElement(element,elements,elements)
       nondeterm      repeatedElement(element,elements)
                      equalElements(elements,elements)
                      nonDuplicatedElements(elements,elements)
                      elementsIntersection(elements,elements,elements)
                      elementsUnion(elements,elements,elements)
                      elementsDifference(elements,elements,elements)

clauses

       %*****************************************************
       % includedElement(Element,Elements) :- Element is
       %      included in the list Elements.
       %
       includedElement(E,[E|_]).
       includedElement(E,[_|Es]) :-
              includedElement(E,Es).


       %*****************************************************
       % includedElements(Es1,Es2) :- The list of elements Es1
       %      is included in the list Es2.
       %
       includedElements([E|Es1],Es2) :-
              includedElement(E,Es2),
              !,
              includedElements(Es1,Es2).
       includedElements([],_) :-
              !.


       %*****************************************************
       % includeElement(Element,ElementsI,ElementsO) :-
       %      Include the given Element in the set of Elements,
       %      if not included yet.
       %
       includeElement(E,Es,Es) :-
              includedElement(E,Es),
              !.
       includeElement(E,Es,[E|Es]) :-
              !.


       %*****************************************************
       % repeatedElement(Element,Elements) :- Element is
       %      repeated in the list of Elements.
       %
       repeatedElement(E,[E|Es]) :-
              includedElement(E,Es).
       repeatedElement(E,[_|Es]) :-
              repeatedElement(E,Es).


       %*****************************************************
       % equalElements(Elements1,Elements2) :- Both sets have
       %      the same elements.
       %
       equalElements(Es1,Es2) :-
              includedElements(Es1,Es2),
              includedElements(Es2,Es1).


       %*****************************************************
       % nonDuplicatedElements(Es1,Es2) :- Given a list of
       %      elements obtains a new list without duplicates.
       %
       nonDuplicatedElements([E|Es1],Es2) :-
              includedElement(E,Es1),
              !,
              nonDuplicatedElements(Es1,Es2).
nonDuplicatedElements([E|Es1],[E|Es2]) :-
        !,
        nonDuplicatedElements(Es1,Es2).
nonDuplicatedElements([],[]) :-
        !.


%*******************************************************
% elementsIntersection(Es1,Es2,Es) :- Intersection
%       of two lists of elements.
%
elementsIntersection([E|Es1],Es2,[E|Es]) :-
        includedElement(E,Es2),
        !,
        elementsIntersection(Es1,Es2,Es).
elementsIntersection([_|Es1],Es2,Es) :-
        !,
        elementsIntersection(Es1,Es2,Es).
elementsIntersection([],_,[]) :-
        !.


%*******************************************************
% elementsUnion(Es1,Es2,Es) :- Union of two lists of
%       elements.
%
elementsUnion([E|Es1],Es2,Es3) :-
        includedElement(E,Es2),
        !,
        elementsUnion(Es1,Es2,Es3).
elementsUnion([E|Es1],Es2,[E|Es3]) :-
        !,
        elementsUnion(Es1,Es2,Es3).
elementsUnion([],Es,Es) :-
        !.


%*******************************************************
% elementsDifference(Es1,Es2,Es) :- Difference between
%       two lists of elements.
%
elementsDifference([E|Es1],Es2,Es3) :-
        includedElement(E,Es2),
        !,
        elementsDifference(Es1,Es2,Es3).
elementsDifference([E|Es1],Es2,[E|Es3]) :-
        !,
        elementsDifference(Es1,Es2,Es3).
elementsDifference([],_,[]) :-
        !.
```

Appendix C. External schema definition DCM

The following is the source code of the external schema definition DCM developed. Its input file is the output file of the DCM in appendix B.

```
%*****************************************************************
%*****************************************************************
%                            OODBEE.DAT
%       Output file
%*****************************************************************
%*****************************************************************

rbaseproperty(p("address"),1)
rbaseproperty(p("name"),2)
rbaseproperty(p("category"),3)
rbaseclass(c("objects"),4)
rbaseclass(c("people"),5)
rbaseclass(c("addresses"),6)
rbaseclass(c("clients"),7)
rbaseclass(c("employees"),8)
rdirectclassproperty(c("people"),p("address"),9)
rdirectclassproperty(c("employees"),p("category"),10)
rdirectclassproperty(c("clients"),p("name"),11)
rdirectclassproperty(c("employees_"),p("name"),12)
rdirectclassproperty(c("employees_"),p("address"),13)
rdirectinheritance(c("people"),c("objects"),14)
rdirectinheritance(c("addresses"),c("objects"),15)
rdirectinheritance(c("clients"),c("people"),16)
rdirectinheritance(c("employees"),c("people"),17)
rdirectinheritance(c("clients"),c("g0"),58)
rdirectinheritance(c("employees_"),c("g0"),58)
rdirectinheritance(c("g0"),c("objects"),58)
rdirectinheritance(c("employees_"),c("g1"),59)
rdirectinheritance(c("clients"),c("g1"),59)
rdirectinheritance(c("g1"),c("objects"),59)
rdirectinheritance(c("g0"),c("people"),60)
rdirectaggregation(c("people"),c("addresses"),p("address"),18)
rdirectaggregation(c("employees_"),c("addresses"),p("address"),19)
rbaseobject(o("o1"),20)
rbaseobject(o("o2"),21)
rbaseobject(o("o3"),22)
rbaseobject(o("o4"),23)
rbaseobject(o("o5"),24)
rdirectclassobject(c("people"),o("o1"),25)
rdirectclassobject(c("clients"),o("o2"),26)
rdirectclassobject(c("employees"),o("o2"),27)
rdirectclassobject(c("addresses"),o("o3"),28)
rdirectclassobject(c("employees"),o("o4"),29)
rdirectclassobject(c("addresses"),o("o5"),30)
rdirectobjectproperty(o("o1"),p("address"),o([o("o3")]),31)
rdirectobjectproperty(o("o2"),p("address"),o([o("o3"),o("o5")]),32)
rdirectobjectproperty(o("o4"),p("address"),o([o("o5")]),33)
rdirectobjectproperty(o("o2"),p("category"),v([v("boss")]),34)
rdirectobjectproperty(o("o4"),p("category"),v([v("technician")]),35)
rdefinedderivedclass(c("employees_"),36)
rdirectderivation(c("employees_"),[c("employees")],preservation,37)
rdirectderivation(c("g1"),[c("people")],preservation,59)
rgeneratedderivedclass(c("g0"),58)
rgeneratedderivedclass(c("g1"),59)
rclasssetselection("es1",[c("objects"),c("clients"),c("employees_")],38)
rqualifiedclasssetselection("es2",[q(esc(c("objects"),nontransformable)),q(esc(c("client
s"),transformable)),q(esc(c("employees_"),nontransformable)),q(esc(c("addresses"),transf
ormable)),q(esc(c("people"),transformable))],39)
rqualifiedclasssetselection("es3",[q(esc(c("objects"),nontransformable)),q(esc(c("client
s"),transformable)),q(esc(c("employees_"),transformable)),q(esc(c("addresses"),transform
able)),q(esc(c("people"),nontransformable))],40)
```

```
rexternalschema("es1",[c("g0"),c("addresses"),c("objects"),c("clients"),c("employees_")]
,[e(is_a(c("addresses"),c("objects"))),e(is_a(c("clients"),c("g0"))),e(is_a(c("employees
_"),c("g0"))),e(is_a(c("g0"),c("objects")))],58)
rexternalschema("es2",[c("clients"),c("addresses"),c("objects"),c("employees_"),c("g1")]
,[e(is_a(c("employees_"),c("g1"))),e(is_a(c("clients"),c("g1"))),e(is_a(c("g1"),c("objec
ts"))),e(is_a(c("addresses"),c("objects")))],59)
rexternalschema("es3",[c("g0"),c("employees_"),c("clients"),c("addresses"),c("objects"),
c("people")],[e(is_a(c("people"),c("objects"))),e(is_a(c("addresses"),c("objects"))),e(i
s_a(c("employees_"),c("g0"))),e(is_a(c("clients"),c("g0"))),e(is_a(c("g0"),c("people")))
],60)
rindirectinheritance(c("clients"),c("objects"),41)
rindirectinheritance(c("employees"),c("objects"),42)
rindirectinheritance(c("employees_"),c("objects"),58)
rinheritedaggregation(c("clients"),c("addresses"),p("address"),43)
rinheritedaggregation(c("employees"),c("addresses"),p("address"),44)
rallclassproperties(c("objects"),[],45)
rallclassproperties(c("people"),[p("address")],46)
rallclassproperties(c("addresses"),[],47)
rallclassproperties(c("clients"),[p("name"),p("address")],48)
rallclassproperties(c("employees"),[p("category"),p("address")],49)
rallclassproperties(c("employees_"),[p("name"),p("address")],50)
rallclassproperties(c("g0"),[p("name"),p("address")],58)
rallclassproperties(c("g1"),[p("name"),p("address")],59)
rallclassobjects(c("objects"),[o("o1"),o("o2"),o("o3"),o("o4"),o("o5")],51)
rallclassobjects(c("people"),[o("o1"),o("o2"),o("o4")],52)
rallclassobjects(c("addresses"),[o("o3"),o("o5")],53)
rallclassobjects(c("clients"),[o("o2")],54)
rallclassobjects(c("employees"),[o("o2"),o("o4")],55)
rallclassobjects(c("employees_"),[o("o4")],56)
rallclassobjects(c("g0"),[o("o2"),o("o4")],58)
rallclassobjects(c("g1"),[o("o1"),o("o2"),o("o4")],59)
rtime(60)
rid(1)
```

```
%*******************************************************************
%*******************************************************************
%                            COMMREPO.PRO
%       Definition Repository.
%*******************************************************************
%*******************************************************************

predicates
                      now(time)
       nondeterm      inheritance(element,element,time)
                      updateNewInheritanceRelationships(elements,time)
                      updateNewInheritanceRelationships(elements,difList,time)
                      updateNewIndirectInheritanceRelationships(elements,time)
                      findIndirectInheritance2(element,time)
       nondeterm      thereIsInheritance2(element,element,time)

       nondeterm      class(element,time)
       nondeterm      classProperties(element,elements,time)
       nondeterm      classPropertiesIntersection(element,element,elements,time)
                      compareClasses(element,element,comparation)

                      directDerivation(element,elements,derSemantics,time)
                      generateDerivedClass(element,time)
                      newClassName(element)
                      associateClassesByDerivation(element,elements,time)

       nondeterm      classObjects(element,elements,time)
                      classObjectsUnion(element,element,element,time)
                      subsumesExtension(element,element,time)

       nondeterm      aggregation(element,element,element,time)

       nondeterm      classSetSelection(schema,elements,time)
       nondeterm      qualifiedClassSetSelection(schema,elements,time)
       nondeterm      externalSchema(schema,elements,elements,time)
                      defineExternalSchema(schema,elements,elements,time)


database - coodb

       rBaseProperty(element,time)
       rBaseClass(element,time)
       rDirectClassProperty(element,element,time)
       rDirectInheritance(element,element,time)
       rDirectAggregation(element,element,element,time)
       rBaseObject(element,time)
       rDirectClassObject(element,element,time)
       rDirectObjectProperty(element,element,evaluation,time)
       rDefinedDerivedClass(element,time)
       rDirectDerivation(element,elements,derSemantics,time)
       rGeneratedDerivedClass(element,time)
       rClassSetSelection(schema,elements,time)
       rQualifiedClassSetSelection(schema,elements,time)
       rExternalSchema(schema,elements,elements,time)

       rIndirectInheritance(element,element,time)
       rInheritedAggregation(element,element,element,time)
       rAllClassProperties(element,elements,time)
       rAllClassObjects(element,elements,time)

       rTime(integer)
       rId(integer)


clauses

       %*****************************************************
       % now(T) :- Obtain the time.
       %
       now(T) :-
              retract(rTime(T1)),
              !,
              T = T1 + 1,
              assertz(rTime(T)).
       now(1) :-
              !,
              assertz(rTime(1)).
```

```
%*******************************************************************
%       Inheritance.
%*******************************************************************

        %*****************************************************
        % inheritance(Class1,Class2,T) :- Class1 is subclass of
        %       Class2.
        %
        inheritance(C1,C2,T) :-
                rDirectInheritance(C1,C2,T2),
                T2 <= T.
        inheritance(C1,C2,T) :-
                rIndirectInheritance(C1,C2,T2),
                T2 <= T.


        %*****************************************************
        % updateNewInheritanceRelationships(Edges,T) :- Given
        %       the Edges obtained in an External Schema, updates
        %       the repository with the new inheritance
        %       relationships obtained.
        %
        updateNewInheritanceRelationships(Es,T) :-
                updateNewInheritanceRelationships(Es,dl(Cs1,[]),T),
                nonDuplicatedElements(Cs1,Cs2),
                updateNewIndirectInheritanceRelationships(Cs2,T).

        updateNewInheritanceRelationships([e(is_a(C1,C2))|Es],
                        dl(Cs1,Cs2),T) :-
                not(inheritance(C1,C2,T)),
                !,
                assertz(rDirectInheritance(C1,C2,T)),
                updateNewInheritanceRelationships(Es,dl(Cs1,[C1|Cs2]),T).
        updateNewInheritanceRelationships([_|Es],Csdl,T) :-
                !,
                updateNewInheritanceRelationships(Es,Csdl,T).
        updateNewInheritanceRelationships([],dl(Cs,Cs),_) :-
                !.


        %*****************************************************
        % updateNewIndirectInheritanceRelationships(Classes,T) :-
        %       Given the set of new classes related directly by
        %       inheritance, obtains the indirect inheritance.
        %
        updateNewIndirectInheritanceRelationships([C|Cs],T) :-
                findIndirectInheritance2(C,T),
                !,
                updateNewIndirectInheritanceRelationships(Cs,T).
        updateNewIndirectInheritanceRelationships([],_) :-
                !.


        %*****************************************************
        % findIndirectInheritance2 :- Find indirect inheritance
        %       relationships.
        %
        findIndirectInheritance2(C1,T) :-
                class(C1,T),
                class(C2,T),
                compareClasses(C1,c(objects),ne),
                compareClasses(C1,C2,ne),
                not(rDirectInheritance(C2,C1,_)),
                not(rDirectInheritance(C1,C2,_)),
                not(rIndirectInheritance(C2,C1,_)),
                rDirectInheritance(C1,C3,_),
                thereIsInheritance2(C3,C2,T),
                not(rIndirectInheritance(C1,C2,_)),
                assertz(rIndirectInheritance(C1,C2,T)),
                fail.
        findIndirectInheritance2(_,_) :-
                !.


        %*****************************************************
        % thereIsInheritance2(Class1,Class2,T) :- Class1 and
        %       Class2 are related by inheritance.
        %
        thereIsInheritance2(C1,C2,T) :-
                rDirectInheritance(C1,C2,T2),
                T2 <= T.
```

```
       thereIsInheritance2(C1,C2,T) :-
              rDirectInheritance(C1,C3,T2),
              T2 <= T,
              thereIsInheritance2(C3,C2,T).


%********************************************************************
%      Classes.
%********************************************************************

       %*****************************************************
       % class(Class,T) :- Class is a class.
       %
       class(C,T) :-
              rBaseClass(C,T2),
              T2 <= T.
       class(C,T) :-
              rDefinedDerivedClass(C,T2),
              T2 <= T.
       class(C,T) :-
              rGeneratedDerivedClass(C,T2),
              T2 <= T.


       %*****************************************************
       % classProperties(Class,Properties,T) :- Class has
       %      Properties.
       %
       classProperties(C,Ps,T) :-
              class(C,T),
              not(bound(Ps)),
              rAllClassProperties(C,Ps,T2),
              T2 <= T.
       classProperties(C,Ps1,T) :-
              class(C,T),
              bound(Ps1),
              rAllClassProperties(C,Ps2,T2),
              T2 <= T,
              equalElements(Ps2,Ps1).
       classProperties(C,Ps,T) :-
              class(C,T),
              bound(Ps),
              not(rAllClassProperties(C,_,_)),
              assertz(rAllClassProperties(C,Ps,T)).


       %*****************************************************
       % classPropertiesIntersection(Class1,Class2,Properties,T) :-
       %      Class1 and Class2 have Properties in common.
       %
       classPropertiesIntersection(C1,C2,Ps,T) :-
              classProperties(C1,Ps1,T),
              classProperties(C2,Ps2,T),
              elementsIntersection(Ps1,Ps2,Ps).


       %*****************************************************
       % compareClasses(C1,C2,Comparation) :- Comparation
       %      between two classes.
       %
       compareClasses(c(C1),c(C2),eq) :-
              C1 = C2,
              !.
       compareClasses(c(C1),c(C2),ne) :-
              C1 <> C2,
              !.


%********************************************************************
%      Derivation.
%********************************************************************

       %*****************************************************
       % directDerivation(C,Cs,Way,T) :- C class is derived
       %      from Cs classes in the way specified.
       %
       directDerivation(C,Cs,Way,T) :-
              rDirectDerivation(C,Cs,Way,T2),
              T2 <= T,
              !.
```

```
        %*****************************************************
        % generateDerivedClass(Class) :- Defines a new generated
        %       derived class.
        %
        generateDerivedClass(C,T) :-
                newClassName(C),
                assertz(rGeneratedDerivedClass(C,T)).


        %*****************************************************
        % newClassName(Class) :- Generates a new class name.
        %
        newClassName(c(C)) :-
                retract(rId(Id1)),
                !,
                Id = Id1 + 1,
                assertz(rId(Id)),
                str_int(StrId,Id),
                concat("g",StrId,Nom),
                C = Nom.
        newClassName(c(g0)) :-
                assertz(rId(0)).


        %*****************************************************
        % associateClassesByDerivation(C,Cs,T) :- Associate C
        %       class with Cs classes by the derivation relationship.
        %
        associateClassesByDerivation(C,Cs,_) :-
                includedElement(C,Cs),
                !.
        associateClassesByDerivation(C,Cs1,T) :-
                rDirectDerivation(C,Cs2,preservation,T2),
                T2 <= T,
                retract(rDirectDerivation(C,_,preservation,_)),
                !,
                elementsUnion(Cs1,Cs2,Cs3),
                assertz(rDirectDerivation(C,Cs3,preservation,T)).
        associateClassesByDerivation(C,Cs,T) :-
                !,
                assertz(rDirectDerivation(C,Cs,preservation,T)).


%*****************************************************************
%       Objects.
%*****************************************************************

        %*****************************************************
        % classObjects(Class,Objects,T) :- Class has Objects as
        %       members.
        %
        classObjects(C,Os,T) :-
                class(C,T),
                not(bound(Os)),
                rAllClassObjects(C,Os,T2),
                T2 <= T.
        classObjects(C,Os1,T) :-
                class(C,T),
                bound(Os1),
                rAllClassObjects(C,Os2,T2),
                T2 <= T,
                equalElements(Os2,Os1).
        classObjects(C,Os,T) :-
                class(C,T),
                bound(Os),
                not(rAllClassObjects(C,_,_)),
                assertz(rAllClassObjects(C,Os,T)).


        %*****************************************************
        % classObjectsUnion(Class1,Class2,Class3,T) :- Class3
        %       contains the union of the set of objects of Class1
        %       and Class2.
        %
        classObjectsUnion(C1,C2,C3,T) :-
                not(rAllClassObjects(C3,_,_)),
                classObjects(C1,Os1,T),
                classObjects(C2,Os2,T),
                !,
                elementsUnion(Os1,Os2,Os3),
                assertz(rAllClassObjects(C3,Os3,T)).
        classObjectsUnion(C1,C2,C3,T) :-
                classObjects(C3,Os3,T),
                classObjects(C1,Os1,T),
                classObjects(C2,Os2,T),
                elementsUnion(Os1,Os2,Os4),
                equalElements(Os3,Os4),
                !.


        %****************************************************
        % subsumesExtension(Class1,Class2,T) :- Class1 subsumes
        %       extesion of Class2. APROXIMATION.
        %
        subsumesExtension(C1,C2,T) :-
                classObjects(C1,Os1,T),
                classObjects(C2,Os2,T),
                includedElements(Os2,Os1),
                !.


%************************************************************
%       Aggregation.
%************************************************************

        %****************************************************
        % aggregation(Class1,Class2,Property,T) :- There is a
        %       property function defined for class Class1 with the
        %       property label Property and the domain class Class2.
        %
        aggregation(C1,C2,P,T) :-
                rDirectAggregation(C1,C2,P,T2),
                T2 <= T.
        aggregation(C1,C2,P,T) :-
                rInheritedAggregation(C1,C2,P,T2),
                T2 <= T.


%************************************************************
%       External Schemas.
%************************************************************

        %****************************************************
        % classSetSelection(S,Cs,T) :- Selection of classes
        %       to compose an External Schema.
        %
        classSetSelection(S,Cs,T) :-
                rClassSetSelection(S,Cs,T2),
                T2 <= T.


        %****************************************************
        % qualifiedClassSetSelection(S,QCs,T) :- Qualified
        %       selection of classes to compose an External Schema.
        %
        qualifiedClassSetSelection(S,QCs,T) :-
                rQualifiedClassSetSelection(S,QCs,T2),
                T2 <= T.


        %****************************************************
        % externalSchema(S,Cs,Es,T) :- Defined External Schema.
        %
        externalSchema(S,Cs,Es,T) :-
                rExternalSchema(S,Cs,Es,T2),
                T2 <= T.


        %****************************************************
        % defineExternalSchema(S,Cs,Es,T) :- Define an External
        %       Schema.
        %
        defineExternalSchema(S,Cs,Es,T) :-
                assertz(rExternalSchema(S,Cs,Es,T)),
                !.
```

```
%*****************************************************************
%*****************************************************************
%                          GESGEN.PRO
%       General External Schema Generation.
%*****************************************************************
%*****************************************************************

include "domains.pro"
include "elemlist.pro"
include "commrepo.pro"

predicates
       run
       generateExternalSchemas(time)

       generateExternalSchema(elements,elements,elements,time)
       propertyDecompositionHierarchyClosure(elements,elements,time)
       propertyDecompositionHierarchyClosureDL(elements,difList,time)
       classHierarchyClosure(elements,elements,elements,time)
       classHierarchyClosure(elements,elements,difList,difList,time)

       addLowestCommonSuperclass(element,element,elements,difList,
                      difList,difList,time)
       nondeterm superclassWithPropertiesInSet(element,element,
                      elements,elements,element,time)
       lowerSuperclassWithPropertiesInSet(element,element,
                      elements,elements,element,time)
       nondeterm superclassWithPropertiesInRepository(element,
                      element,elements,element,time)
       lowersuperclassWithPropertiesInRepository(element,element,
                      elements,element,time)

       eliminateReduntantEdges(elements,elements,time)
       eliminateReduntantEdges(elements,elements,difList,time)
       nondeterm indirectEdge(element,elements)
       nondeterm definedEdge(element,elements)

       generateQualifiedExternalSchema(elements,elements,elements,time)
       qualifiedClasses(elements,classQuality,elements)
       transfClassesRelatedByAggregation(elements,elements,elements,
                      elements,time)
       transfClassesRelatedByAggregationDL(elements,elements,difList,
                      difList,time)
       classesWithProperties(elements,elements,time)
       classesWithPropertiesDL(elements,difList,time)
       transfClassesPropDecompHierarchyClosure(elements,elements,
                      elements,elements,time)
       transfClassesPropDecompHierarchyClosure(elements,elements,
                      difList,time)
       classPropertyDecompositionClosure(element,elements,element,time)
       classPropertyDecompositionClosure(element,elements,elements,
                      difList,time)

       subsumtionIsomorficClasses(elements,elements,elements,time)
       subsumtionIsomorficClassesDL(elements,difList,difList,time)
       integrationOfTansformableClasses(elements,elements,elements,
                      elements,elements,time)
       integrationOfTansformableClasses(elements,difList,difList,time)
       transformableClassIntegration(element,elements,elements,
                      elements,elements,time)
       branchTransformableClassIntegration(element,elements,elements,
                      elements,elements,time)
       branchTransformableClassIntegrationDL(element,elements,difList,
                      difList,time)
       integrateTransformableClass(element,element,elements,elements,
                      elements,elements,time)
       leafTransformableClassIntegration(element,elements,elements,
                      elements,elements,time)
       subsumedBySomeClass(element,elements,time)
       subsumingSuperclassPropertyUnion(element,element,elements,
                      elements,elements,time)
       subsumingSuperclassPropertyUnionDL(element,element,elements,
                      elements,difList,time)
       subsumingClassPropertyUnion(element,elements,elements,time)
       subsumingClassPropertyUnionDL(element,elements,difList,time)
       defineTransformableClass(element,element,time)
       eliminateRedundantTransformableClasses(elements,elements,
                      elements,elements,elements,elements,time)
       eliminateRedundantTransformableClasses(elements,elements,difList,
                      difList,time)
```

```
        exclusiveNodesEdge(element,elements)
        unifyTransformableAndAddedClasses(element,element,element,time)
        unifyClassesInSchema(element,element,element,elements,elements,
                       elements,elements,time)
        unifyClassesInSchema(elements,element,element,difList,time)
        obtainedFromTransformableClasses(elements,elements,elements,time)
        obtainedFromTransformableClassesDL(elements,elements,difList,time)


goal
        run.


clauses
        run :-
                consult("oodb.dat",coodb),
                now(T),
                generateExternalSchemas(T),
                save("oodbee.dat",coodb),
                write("\n\nOK."),
                !.

%******************************************************************
%******************************************************************

        %*****************************************************
        % generateExternalSchemas(T) :- Generated all the
        %       defined external schemas.
        %
        generateExternalSchemas(T) :-
                classSetSelection(S,Cs1,T),
                not(externalSchema(S,_,_,T)),
                now(T1),
                generateExternalSchema(Cs1,Cs2,Es,T1),
                defineExternalSchema(S,Cs2,Es,T1),
                fail.
        generateExternalSchemas(T) :-
                qualifiedClassSetSelection(S,QCs,T),
                not(externalSchema(S,_,_,T)),
                now(T1),
                generateQualifiedExternalSchema(QCs,Cs,Es,T1),
                defineExternalSchema(S,Cs,Es,T1),
                fail.
        generateExternalSchemas(_) :-
                !.


%******************************************************************
%******************************************************************

        %*****************************************************
        % generateExternalSchema(Cs1,Cs2,Es,T) :- Given a set
        %       of classes Cs1, generates a correct External Schema
        %       with all the classes needed (Cs2) and the edges
        %       between them (Es).
        %
        generateExternalSchema(Cs1,Cs2,Es2,T) :-
                includeElement(c(objects),Cs1,Cs3),
                propertyDecompositionHierarchyClosure(Cs3,Cs4,T),
                classHierarchyClosure(Cs4,Cs2,Es1,T),
                eliminateReduntantEdges(Es1,Es2,T),

                updateNewInheritanceRelationships(Es2,T),
                !.


        %*****************************************************
        % propertyDecompositionHierarchyClosure(Cs1,Cs2,T) :-
        %       Given a set of classes Cs1 obtains the set of
        %       classes Cs2 closed according to the property
        %       decomposition hierarchy: "all classes that are being
        %       used in a external schema are also defined within
        %       the external schema"; adding the classes referenced
        %       and not included.
        %
        propertyDecompositionHierarchyClosure(Cs1,Cs2,T) :-
                propertyDecompositionHierarchyClosureDL(Cs1,dl(Cs2,Cs1),T).

        propertyDecompositionHierarchyClosureDL([C1|Cs1],dl(Cs2,Cs3),T) :-
                aggregation(C1,C2,_,T),
                not(includedElement(C2,Cs3)),
                !,
                propertyDecompositionHierarchyClosureDL([C1,C2|Cs1],
                                dl(Cs2,[C2|Cs3]),T).
        propertyDecompositionHierarchyClosureDL([_|Cs1],Cs2dl,T) :-
                !,
                propertyDecompositionHierarchyClosureDL(Cs1,Cs2dl,T).
        propertyDecompositionHierarchyClosureDL([],dl(Cs,Cs),_) :-
                !.


        %*******************************************************
        % classHierarchyClosure(Cs1,Cs2,Es,T) :- Given the set
        %       of classes Cs1, obtains the associated External
        %       Schema closed w.r.t. class hierarchy.
        %
        classHierarchyClosure(Cs1,Cs2,Es,T) :-
                classHierarchyClosure(Cs1,Cs1,dl(Cs2,Cs1),dl(Es,[]),T).

        classHierarchyClosure([C|Cs1],[C|Cs2],Cs3dl,Esdl,T) :-
                !,
                classHierarchyClosure([C|Cs1],Cs2,Cs3dl,Esdl,T).
        classHierarchyClosure([C1|Cs1],[C2|Cs2],dl(Cs3,Cs4),dl(Es1,Es2),T) :-
                classPropertiesIntersection(C1,C2,Ps,T),
                !,
                addLowestCommonSuperclass(C1,C2,Ps,dl(Cs6,Cs4),dl(Cs5,Cs1),
                                dl(Es3,Es2),T),
                classHierarchyClosure([C1|Cs5],Cs2,dl(Cs3,Cs6),dl(Es1,Es3),T).
                        % Added classes are considered again in order to update
                        % all its inheritance relationships (Cs1 -> Cs5).
        classHierarchyClosure([_|Cs1],[],dl(Cs2,Cs3),Esdl,T) :-
                !,
                classHierarchyClosure(Cs1,Cs3,dl(Cs2,Cs3),Esdl,T).
        classHierarchyClosure([],_,dl(Cs,Cs),dl(Es,Es),_) :-
                !.


        %*******************************************************
        % addLowestCommonSuperclass(C1,C2,Ps,Cs1dl,Cs2dl,Esdl,T) :-
        %       Given the classes C1, C2 and their common properties
        %       Ps, obtains the lowest common superclass (LCS) in
        %       order to update the sets of classes and edges adding
        %       the LCS class and its inheritance relationships with
        %       the classes given.
        %
        addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
                        dl(Es2,Es1),T) :-
                classProperties(C1,Ps,T),
                subsumesExtension(C1,C2,T),
                includeElement(e(is_a(C2,C1)),Es1,Es2),
                !.
        addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
                        dl(Es2,Es1),T) :-
                classProperties(C2,Ps,T),
                subsumesExtension(C2,C1,T),
                includeElement(e(is_a(C1,C2)),Es1,Es2),
                !.
        addLowestCommonSuperclass(C1,C2,Ps,dl(Cs1,Cs1),dl(Cs2,Cs2),
                        dl(Es3,Es1),T) :-
                superclassWithPropertiesInSet(C1,C2,Ps,Cs1,C3,T),
                not(lowerSuperclassWithPropertiesInSet(C1,C2,Ps,Cs1,C3,T)),
                includeElement(e(is_a(C1,C3)),Es1,Es2),
                includeElement(e(is_a(C2,C3)),Es2,Es3),
                !.
        addLowestCommonSuperclass(C1,C2,Ps,dl(Cs2,Cs1),dl(Cs4,Cs3),
                        dl(Es3,Es1),T) :-
                superclassWithPropertiesInRepository(C1,C2,Ps,C3,T),
                not(lowersuperclassWithPropertiesInRepository(C1,C2,Ps,C3,T)),
                includeElement(C3,Cs1,Cs2),
                includeElement(C3,Cs3,Cs4),
                includeElement(e(is_a(C1,C3)),Es1,Es2),
                includeElement(e(is_a(C2,C3)),Es2,Es3),
                !.
        addLowestCommonSuperclass(C1,C2,Ps,dl(Cs2,Cs1),dl(Cs4,Cs3),
                        dl(Es3,Es1),T) :-
                generateDerivedClass(C3,T),
                classProperties(C3,Ps,T),
                classObjectsUnion(C1,C2,C3,T),
                includeElement(C3,Cs1,Cs2),
                includeElement(C3,Cs3,Cs4),
                includeElement(e(is_a(C1,C3)),Es1,Es2),
                includeElement(e(is_a(C2,C3)),Es2,Es3),
                !.


         %*****************************************************
         % superclassWithPropertiesInSet(C1,C2,Ps,Cs,C3,T) :-
         %      C3 is superclass of C1 and C2, belongs to Cs and
         %      has properties Ps.
         %
         superclassWithPropertiesInSet(C1,C2,Ps,Cs,C3,T) :-
                includedElement(C3,Cs),
                classProperties(C3,Ps,T),
                subsumesExtension(C3,C1,T),
                subsumesExtension(C3,C2,T).


         %*****************************************************
         % lowerSuperclassWithPropertiesInSet(C1,C2,Ps,Cs,C3,T) :-
         %      C3 is superclass of C1 and C2, belongs to Cs and
         %      has properties Ps, but another class exists with
         %      the same properties that C3 and is subclass of C3.
         %
         lowerSuperclassWithPropertiesInSet(C1,C2,Ps,Cs,C3,T) :-
                superclassWithPropertiesInSet(C1,C2,Ps,Cs,C4,T),
                compareClasses(C4,C3,ne),
                subsumesExtension(C3,C4,T),
                !.


         %*****************************************************
         % superclassWithPropertiesInRepository(C1,C2,Ps,C3,T) :-
         %      C3 is superclass of C1 and C2 in the repository and
         %      has properties Ps.
         %
         superclassWithPropertiesInRepository(C1,C2,Ps,C3,T) :-
                classProperties(C3,Ps,T),
                subsumesExtension(C3,C1,T),
                subsumesExtension(C3,C2,T).


         %*****************************************************
         % lowerSuperclassWithPropertiesInRepository(C1,C2,Ps,C3,T) :-
         %      C3 is superclass of C1 and C2 in the repository and
         %      has properties Ps, but another class exists with
         %      the same properties that C3 and subclass of C3.
         %
         lowersuperclassWithPropertiesInRepository(C1,C2,Ps,C3,T) :-
                superclassWithPropertiesInRepository(C1,C2,Ps,C4,T),
                compareClasses(C4,C3,ne),
                subsumesExtension(C3,C4,T),
                !.


         %*****************************************************
         % eliminateReduntantEdges(Es1,Es2,T) :- Given a set of
         %      edges Es1, eliminates the redundant edges to obtain
         %      the set Es2.
         %
         eliminateReduntantEdges(Es1,Es2,T) :-
                nonDuplicatedElements(Es1,Es3),
                eliminateReduntantEdges(Es3,Es3,dl(Es2,[]),T).

         eliminateReduntantEdges([E|Es1],Es2,Es3dl,T) :-
                indirectEdge(E,Es2),
                !,
                eliminateReduntantEdges(Es1,Es2,Es3dl,T).
         eliminateReduntantEdges([E|Es1],Es2,dl(Es3,Es4),T) :-
                !,
                eliminateReduntantEdges(Es1,Es2,dl(Es3,[E|Es4]),T).
         eliminateReduntantEdges([],_,dl(Es,Es),_) :-
                !.


         %*****************************************************
         % indirectEdge(E,Es) :- E is an edge that is indirectly
         %      defined in the set of edges Es.
         %
         indirectEdge(e(is_a(C1,C2)),Es) :-
                includedElement(e(is_a(C1,C3)),Es),
                definedEdge(e(is_a(C3,C2)),Es).
```

```
        %******************************************************
        % definedEdge(E,Es) :- The given edge E is defined
        %       directly or indirectly in the list of edges Es.
        %
        definedEdge(E,Es) :-
                includedElement(E,Es).
        definedEdge(E,Es) :-
                indirectEdge(E,Es).


%*******************************************************************
%*******************************************************************

        %******************************************************
        % generateQualifiedExternalSchema(QCs,Cs,Es,T) :- Given
        %       a set of qualified classes QCs, generates a correct
        %       External Schema with all the classes Cs needed and
        %       the edges Es between them.
        %
        generateQualifiedExternalSchema(QCs1,Cs,Es,T) :-
                includeElement(q(esc(c(objects),nonTransformable)),QCs1,QCs2),

                % NonTransformable and Transformable Classes.
                qualifiedClasses(QCs2,nonTransformable,CsNT1),
                qualifiedClasses(QCs2,transformable,CsT1),

                % Property Decomposition Hierarchy Closure.
                % ----------------------------------------
                        % Classes refered by NT classes become NT.
                        propertyDecompositionHierarchyClosure(CsNT1,CsNT2,T),

                        % T classes refered by NT classes become NT.
                        elementsDifference(CsT1,CsNT2,CsT2),

                        % T classes refered by T classes become NT.
                        transfClassesRelatedByAggregation(CsT2,CsNT2,CsT3,CsNT3,T),

                        % References of T classes to classes not included
                        % have to be supressed.
                        classesWithProperties(CsT3,CsTWPs1,T),
                        transfClassesPropDecompHierarchyClosure(CsTWPs1,CsT3,CsNT3,
                                        CsTWPs2,T),

                % Class Hierarchy Closure.
                % ------------------------
                        % NT classes hierarchy closure.
                        classHierarchyClosure(CsNT3,CsNT4,EsNT1,T),
                        eliminateReduntantEdges(EsNT1,EsNT2,T),

                        % Unification of T classes that contain the same objects.
                        subsumtionIsomorficClasses(CsTWPs2,ICsTWPs,CsT4,T),

                        % Integration of T classes.
                        integrationOfTansformableClasses(ICsTWPs,CsNT4,EsNT2,
                                        Cs3,Es2,T),
                        eliminateReduntantEdges(Es2,Es3,T),

                        % T classes that can be replaced by auxiliary added clases.
                        eliminateRedundantTransformableClasses(CsNT3,CsT4,Cs3,Es3,
                                        Cs,Es,T),

                updateNewInheritanceRelationships(Es,T),
                !.


        %******************************************************
        % qualifiedClasses(QCs,Q,Cs) :- Given a set of qualified
        %       classes and a qualification, obtains the set of
        %       classes qualified this way.
        %
        qualifiedClasses([q(esc(C,_))|QCs],any,[C|Cs]) :-
                !,
                qualifiedClasses(QCs,any,Cs).
        qualifiedClasses([q(esc(C,Q))|QCs],Q,[C|Cs]) :-
                !,
                qualifiedClasses(QCs,Q,Cs).
        qualifiedClasses([_|QCs],Q,Cs) :-
                !,
                qualifiedClasses(QCs,Q,Cs).
        qualifiedClasses([],_,[]) :-
                !.
```

```
%*****************************************************
% transfClassesRelatedByAggregation(CsT1,CsNT1,CsT2,
%       CsNT2,T) :- T classes refered by T classes become NT.
%       Given the original T and NT sets of classes, obtains
%       the new ones.
%
transfClassesRelatedByAggregation(CsT1,CsNT1,CsT2,CsNT2,T) :-
        transfClassesRelatedByAggregationDL(CsT1,CsT1,dl(CsT2,[]),
                        dl(CsNT2,CsNT1),T).

transfClassesRelatedByAggregationDL([C1|CsT1],CsT0,CsT2dl,
                dl(CsNT2,CsNT1),T) :-
        aggregation(C2,C1,_,T),
        includedElement(C2,CsT0),
        !,
        transfClassesRelatedByAggregationDL(CsT1,CsT0,CsT2dl,
                        dl(CsNT2,[C1|CsNT1]),T).
transfClassesRelatedByAggregationDL([C|CsT1],CsT0,
                dl(CsT2,CsT3),CsNTdl,T) :-
        !,
        transfClassesRelatedByAggregationDL(CsT1,CsT0,
                        dl(CsT2,[C|CsT3]),CsNTdl,T).
transfClassesRelatedByAggregationDL([],_,dl(CsT,CsT),
                dl(CsNT,CsNT),_) :-
        !.


%*****************************************************
% classesWithProperties(Cs,CsWPs,T) :- Given a set of
%       classes Cs returns a set of classes with properties.
%
classesWithProperties(Cs,CsWPs,T) :-
        classesWithPropertiesDL(Cs,dl(CsWPs,[]),T).

classesWithPropertiesDL([C|Cs1],dl(CsWPs,Cs2),T) :-
        classProperties(C,Ps,T),
        !,
        classesWithPropertiesDL(Cs1,dl(CsWPs,[cp(C,Ps)|Cs2]),T).
classesWithPropertiesDL([],dl(CsWPs,CsWPs),_) :-
        !.


%*****************************************************
% transfClassesPropDecompHierarchyClosure(CsTWPs1,CsT,
%       CsNT,CsTWPs2,T) :- Given a set of classes with their
%       properties, deletes the properties that reference
%       classes not included in the sets of NT and T classes.
%
transfClassesPropDecompHierarchyClosure(CsPs1,CsT,CsNT,
                CsPs2,T) :-
        elementsUnion(CsT,CsNT,AllCs),
        transfClassesPropDecompHierarchyClosure(CsPs1,AllCs,
                        dl(CsPs2,[]),T).

transfClassesPropDecompHierarchyClosure([CP1|CsPs1],AllCs,
                dl(CsTWPs,CsPs2),T) :-
        !,
        classPropertyDecompositionClosure(CP1,AllCs,CP2,T),
        transfClassesPropDecompHierarchyClosure(CsPs1,AllCs,
                dl(CsTWPs,[CP2|CsPs2]),T).
transfClassesPropDecompHierarchyClosure([],_,
                dl(CsTWPs,CsTWPs),_) :-
        !.


%*****************************************************
% classPropertyDecompositionClosure(CP1,Cs,CP2,T) :-
%       Given a class with its properties, deletes the
%       properties that don't reference the set of classes
%       also given; returns the class with the properties
%       that remain.
%
classPropertyDecompositionClosure(cp(C,Ps1),Cs,cp(C,Ps2),T) :-
        classPropertyDecompositionClosure(C,Ps1,Cs,dl(Ps2,[]),T).

classPropertyDecompositionClosure(C1,[P|Ps1],Cs,Ps2dl,T) :-
        aggregation(C1,C2,P,T),
        not(includedElement(C2,Cs)),
        !,
        classPropertyDecompositionClosure(C1,Ps1,Cs,Ps2dl,T).
classPropertyDecompositionClosure(C,[P|Ps1],Cs,dl(Ps2,Ps3),T) :-
        !,
        classPropertyDecompositionClosure(C,Ps1,Cs,dl(Ps2,[P|Ps3]),T).
classPropertyDecompositionClosure(_,[],_,dl(Ps,Ps),_) :-
        !.


%*******************************************************
% subsumtionIsomorficClasses(CsPs,ICsPs,Cs,T) :- Given
%       a set of classes with properties, obtains isomorfic
%       groups of classes using the subsumtion relationship.
%
subsumtionIsomorficClasses(CsPs,ICsPs,Cs,T) :-
        subsumtionIsomorficClassesDL(CsPs,dl(ICsPs,[]),
                        dl(Cs,[]),T).

subsumtionIsomorficClassesDL([cp(C1,Ps1)|CsPs1],
                dl(ICsPs1,ICsPs2),Csdl,T) :-
        includedElement(si(C2,Ps2,Cs2),ICsPs2),
        subsumesExtension(C1,C2,T),
        subsumesExtension(C2,C1,T),
        !,
        elementsDifference(ICsPs2,[si(C2,Ps2,Cs2)],ICsPs3),
        elementsUnion(Ps2,Ps1,Ps3),
        subsumtionIsomorficClassesDL(CsPs1,
                        dl(ICsPs1,[si(C2,Ps3,[C1|Cs2])|ICsPs3]),Csdl,T).
subsumtionIsomorficClassesDL([cp(C,Ps)|CsPs],dl(ICsPs1,ICsPs2),
                dl(Cs1,Cs2),T) :-
        !,
        subsumtionIsomorficClassesDL(CsPs,dl(ICsPs1,
                        [si(C,Ps,[C])|ICsPs2]),dl(Cs1,[C|Cs2]),T).
subsumtionIsomorficClassesDL([],dl(ICsPs,ICsPs),dl(Cs,Cs),_) :-
        !.


%*******************************************************
% integrationOfTansformableClasses(ICsTWPs,CsNT,EsNT,Cs,
%               Es,T) :- Given the set of T classes and the
%       schema of NT classes, integrates the T classes
%       obtaining the new schema.
%
integrationOfTansformableClasses(ICsTWPs,CsNT,EsNT,Cs,Es,T) :-
        integrationOfTansformableClasses(ICsTWPs,
                        dl(Cs,CsNT),dl(Es,EsNT),T).

integrationOfTansformableClasses(ICsTWPs1,dl(Cs1,Cs2),
                dl(Es1,Es2),T) :-
        includedElement(si(C,Ps,Cs),ICsTWPs1),
        not(subsumedBySomeClass(C,ICsTWPs1,T)),
        !,
        elementsDifference(ICsTWPs1,[si(C,Ps,Cs)],ICsTWPs2),
        transformableClassIntegration(si(C,Ps,Cs),Cs2,Es2,Cs3,Es3,T),
        integrationOfTansformableClasses(ICsTWPs2,dl(Cs1,Cs3),
                        dl(Es1,Es3),T).
integrationOfTansformableClasses([],dl(Cs,Cs),dl(Es,Es),_) :-
        !.


%*******************************************************
% transformableClassIntegration(si(C,Ps,Cs),Cs1,Es1,Cs2,
%               Es2,T) :- Given a T class, the set of classes
%       integrated and the edges between them, integrates
%       the class obtaining a new set of classes and edges.
%
transformableClassIntegration(si(C1,Ps1,Cs1),Cs2,Es2,
                Cs3,Es3,T) :-
        includedElement(C2,Cs2),
        subsumesExtension(C1,C2,T),
        !,
        branchTransformableClassIntegration(si(C1,Ps1,Cs1),
                        Cs2,Es2,Cs3,Es3,T).
transformableClassIntegration(si(C1,Ps1,Cs1),Cs2,Es2,
                Cs3,Es3,T) :-
        !,
        leafTransformableClassIntegration(si(C1,Ps1,Cs1),
                        Cs2,Es2,Cs3,Es3,T).


%*******************************************************
% branchTransformableClassIntegration(si(C1,Ps1,Cs1),
%       ,Cs2,Es2,Cs3,Es3,T) :- Transform and integrate the
%       given class in a branch of the class hierarchy.
%
branchTransformableClassIntegration(si(C1,Ps1,Cs1),
               Cs2,Es2,Cs3,Es3,T) :-
       branchTransformableClassIntegrationDL(si(C1,Ps1,Cs1),Cs2,
                      dl(Cs3,Cs2),dl(Es3,Es2),T).
branchTransformableClassIntegrationDL(si(C1,Ps1,Cs1),[C2|Cs2],
               dl(Cs3,Cs4),dl(Es1,Es2),T) :-
       subsumesExtension(C1,C2,T),
       !,
       integrateTransformableClass(si(C1,Ps1,Cs1),C2,Cs4,Es2,
                      Cs5,Es3,T),
       branchTransformableClassIntegrationDL(si(C1,Ps1,Cs1),Cs2,
                      dl(Cs3,Cs5),dl(Es1,Es3),T).
branchTransformableClassIntegrationDL(C,[_|Cs],Csdl,Esdl,T) :-
       !,
       branchTransformableClassIntegrationDL(C,Cs,Csdl,Esdl,T).
branchTransformableClassIntegrationDL(_,[],dl(Cs,Cs),dl(Es,Es),_) :-
       !.


%*****************************************************
% integrateTransformableClass(si(C1,Ps1,Cs1),C2,Cs2,Es2,
%       Cs3,Es3,T) :- Given a transformable class C1 that
%       subsumes C2 class, and a schema, transforms and
%       integrates the transformable class in the schema.
%
integrateTransformableClass(si(C1,Ps1,Cs1),C2,Cs2,Es2,
               Cs3,Es3,T) :-
       subsumingSuperclassPropertyUnion(C1,C2,Cs2,Es2,Ps3,T),
       elementsUnion(Ps1,Ps3,Ps4),

       classProperties(C2,Ps2,T),
       elementsIntersection(Ps2,Ps4,Ps5),

       defineTransformableClass(si(C1,Ps5,Cs1),C3,T),
       includeElement(C3,Cs2,Cs4),
       classHierarchyClosure([C3],Cs2,dl(Cs3,Cs4),
                      dl(Es3,Es2),T),
       !.
       % Add to the T class C1 the properties of the superclasses of
       % C2 class that subsume C1; the new class will have all these
       % properties intesectioned with C2 properties (it is a C2
       % superclass), this is the class to integrate by inheritance
       % in a closed schema.


%*****************************************************
% leafTransformableClassIntegration(si(C1,Ps1,Cs1),
%       Cs2,Es2,Cs3,Es3,T) :- The given transformable class,
%       that doesn't subsume any of the existing classes, is
%       integrated in the class hierarchy.
%
leafTransformableClassIntegration(si(C1,Ps1,Cs1),
               Cs2,Es2,Cs3,Es3,T) :-
       subsumingClassPropertyUnion(C1,Cs2,Ps2,T),
       elementsUnion(Ps1,Ps2,Ps3),

       defineTransformableClass(si(C1,Ps3,Cs1),C2,T),
       includeElement(C2,Cs2,Cs4),
       classHierarchyClosure([C2],Cs2,dl(Cs3,Cs4),
                      dl(Es3,Es2),T),
       !.


%*****************************************************
% subsumedBySomeClass(C,ICsTWPs,T) :- The given a class
%       is subsumed by some class from the given list.
%
subsumedBySomeClass(C1,[si(C2,_,_)|_],T) :-
       compareClasses(C1,C2,ne),
       subsumesExtension(C2,C1,T),
       !.
subsumedBySomeClass(C,[_|ICsTWPs],T) :-
       !,
       subsumedBySomeClass(C,ICsTWPs,T).


%*****************************************************
% subsumingSuperclassPropertyUnion(C1,C2,Cs,Es,Ps,T) :-
%       Ps is the union of the properties of the superclasses
%       of C2 class that subsume C1 class.
%
subsumingSuperclassPropertyUnion(C1,C2,Cs,Es,Ps,T) :-
        subsumingSuperclassPropertyUnionDL(C1,C2,Cs,Es,
                      dl(Ps,[]),T).

subsumingSuperclassPropertyUnionDL(C1,C2,[C3|Cs],Es,
               dl(Ps1,Ps2),T) :-
        definedEdge(e(is_a(C2,C3)),Es),
        subsumesExtension(C3,C1,T),
        classProperties(C3,Ps3,T),
        elementsUnion(Ps2,Ps3,Ps4),
        !,
        subsumingSuperclassPropertyUnionDL(C1,C2,Cs,Es,
                      dl(Ps1,Ps4),T).
subsumingSuperclassPropertyUnionDL(C1,C2,[_|Cs],Es,Psdl,T) :-
        !,
        subsumingSuperclassPropertyUnionDL(C1,C2,Cs,Es,Psdl,T).
subsumingSuperclassPropertyUnionDL(_,_,[],_,dl(Ps,Ps),_) :-
        !.


%*******************************************************
% subsumingClassPropertyUnion(C1,Cs,Ps,T) :- Ps is the
%       union of the properties of the classes of Cs that
%       subsume C1 class.
%
subsumingClassPropertyUnion(C1,Cs,Ps,T) :-
        subsumingClassPropertyUnionDL(C1,Cs,dl(Ps,[]),T).

subsumingClassPropertyUnionDL(C1,[C2|Cs],dl(Ps1,Ps2),T) :-
        subsumesExtension(C2,C1,T),
        classProperties(C2,Ps3,T),
        elementsUnion(Ps2,Ps3,Ps4),
        !,
        subsumingClassPropertyUnionDL(C1,Cs,dl(Ps1,Ps4),T).
subsumingClassPropertyUnionDL(C,[_|Cs],Psdl,T) :-
        !,
        subsumingClassPropertyUnionDL(C,Cs,Psdl,T).
subsumingClassPropertyUnionDL(_,_,dl(Ps,Ps),_) :-
        !.


%*******************************************************
% defineTransformableClass(si(C1,Ps1,Cs1),C2,T) :- Given
%       a transformed transformable class C1, defines this
%       class in the repository (if not defined yet) and
%       returns it in C2.
%
defineTransformableClass(si(C1,Ps1,Cs1),C2,T) :-
        classProperties(C2,Ps1,T),
        subsumesExtension(C2,C1,T),
        subsumesExtension(C1,C2,T),
        !,
        associateClassesByDerivation(C2,Cs1,T).
defineTransformableClass(si(C1,Ps1,Cs1),C2,T) :-
        generateDerivedClass(C2,T),
        classProperties(C2,Ps1,T),
        classObjects(C1,Os1,T),
        classObjects(C2,Os1,T),
        !,
        associateClassesByDerivation(C2,Cs1,T).


%*******************************************************
% eliminateRedundantTransformableClasses(CsNT,CsT,
%       Cs1,Es1,Cs2,Es2,T) :- Given the sets of NT and T
%       classes and the schema obtained, eliminate the
%       redundant classes of the schema, obtaining a new one.
%
eliminateRedundantTransformableClasses(CsNT,CsT,Cs1,Es1,
               Cs2,Es2,T) :-
        % Classes added to the schema.
        elementsDifference(Cs1,CsNT,Cs3),
        elementsDifference(Cs3,CsT,Cs4),
        obtainedFromTransformableClasses(Cs4,CsT,CsT2,T),
        elementsUnion(CsT,CsT2,CsTnew),
        elementsDifference(Cs4,CsT2,CsAdd),
        eliminateRedundantTransformableClasses(CsAdd,CsTnew,
                      dl(Cs2,Cs1),dl(Es2,Es1),T).
```

```
eliminateRedundantTransformableClasses([C1|Cs],CsT,
               dl(Cs2,Cs1),dl(Es2,Es1),T) :-
       exclusiveNodesEdge(e(is_a(C1,C2)),Es1),
       includedElement(C2,CsT),
       !,
       unifyTransformableAndAddedClasses(C2,C1,C3,T),
       unifyClassesInSchema(C2,C1,C3,Cs1,Es1,Cs3,Es3,T),
       eliminateRedundantTransformableClasses(Cs,CsT,
               dl(Cs2,Cs3),dl(Es2,Es3),T).
eliminateRedundantTransformableClasses([_|Cs],CsT,Csdl,Esdl,T) :-
       !,
       eliminateRedundantTransformableClasses(Cs,CsT,Csdl,Esdl,T).
eliminateRedundantTransformableClasses([],_,dl(Cs,Cs),
               dl(Es,Es),_) :-
       !.


%*****************************************************
% obtainedFromTransformableClasses(Cs1,CsT,Cs2,T) :-
%      Cs2 are the classes from Cs1 that have been obtained
%      from the set CsT of T classes given.
%
obtainedFromTransformableClasses(Cs1,CsT,Cs2,T) :-
       obtainedFromTransformableClassesDL(Cs1,CsT,dl(Cs2,[]),T).

obtainedFromTransformableClassesDL([C|Cs1],CsT,dl(Cs2,Cs3),T) :-
       directDerivation(C,Cs4,preservation,T),
       not(elementsIntersection(CsT,Cs4,[])),
       !,
       obtainedFromTransformableClassesDL(Cs1,CsT,dl(Cs2,[C|Cs3]),T).
obtainedFromTransformableClassesDL([_|Cs1],CsT,Csdl,T) :-
       !,
       obtainedFromTransformableClassesDL(Cs1,CsT,Csdl,T).
obtainedFromTransformableClassesDL([],_,dl(Cs,Cs),_) :-
       !.


%*****************************************************
% exclusiveNodesEdge(e(is_a(C1,C2)),Es) :- The given
%      edge is the only one that has class C1 as the
%      starting class,       and also is the only one that has
%      C2 as the ending class.
%
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
       includedElement(e(is_a(C1,C2)),Es),
       includedElement(e(is_a(C1,C3)),Es),
       compareClasses(C3,C2,ne),
       !,
       fail.
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
       includedElement(e(is_a(C1,C2)),Es),
       includedElement(e(is_a(C3,C2)),Es),
       compareClasses(C3,C1,ne),
       !,
       fail.
exclusiveNodesEdge(e(is_a(C1,C2)),Es) :-
       includedElement(e(is_a(C1,C2)),Es),
       !.


%*****************************************************
% unifyTransformableAndAddedClasses(CT,CA,C,T) :- Given
%      a T class CT, and an added class CA, unifies them
%      in a new class C which has the properties of CA and
%      the objects of CT.
%
unifyTransformableAndAddedClasses(CT,CA,C,T) :-
       classProperties(CA,Ps,T),
       !,
       defineTransformableClass(si(CT,Ps,[CT]),C,T).


%*****************************************************
% unifyClassesInSchema(C1,C2,C3,Cs1,Es1,Cs2,Es2,T) :-
%      Given two classes C1 and C2 that are unified in C3,
%      and a schema, reflects this fact in the schema.
%
unifyClassesInSchema(C1,C2,C3,Cs1,Es1,Cs2,Es2,T) :-
       elementsDifference(Cs1,[C1,C2],Cs3),
       elementsUnion(Cs3,[C3],Cs2),
        unifyClassesInSchema(Es1,e(is_a(C2,C1)),C3,
                        dl(Es2,[]),T).

unifyClassesInSchema([E|Es],E,C,Esdl,T) :-
        !,
        unifyClassesInSchema(Es,E,C,Esdl,T).
unifyClassesInSchema([e(is_a(C1,C4))|Es],
                e(is_a(C2,C1)),C3,dl(Es2,Es1),T) :-
        !,
        unifyClassesInSchema(Es,e(is_a(C2,C1)),C3,
                        dl(Es2,[e(is_a(C3,C4))|Es1]),T).
unifyClassesInSchema([e(is_a(C4,C2))|Es],
                e(is_a(C2,C1)),C3,dl(Es2,Es1),T) :-
        !,
        unifyClassesInSchema(Es,e(is_a(C2,C1)),C3,
                        dl(Es2,[e(is_a(C4,C3))|Es1]),T).
unifyClassesInSchema([E1|Es],E2,C,dl(Es2,Es1),T) :-
        !,
        unifyClassesInSchema(Es,E2,C,dl(Es2,[E1|Es1]),T).
unifyClassesInSchema([],_,_,dl(Es,Es),_) :-
        !.
```

Appendix D. List of publications

The following is a list of the publications where work reported in this thesis is presented:

- J. Samos, “Definición de Vistas en Bases de Datos Orientadas a Objectos,” Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, Report LSI-93-19-T, May 1993.

  This report introduces the concept of “view” with the different meanings used by other authors (derived class and external schema), and presents the main problems and uses of the definition of derived classes and external schemas. It is related to chapters 1 and 2.

- J. Samos, “Esquemas Externos en Bases de Datos Orientadas a Objectos,” Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, Report LSI-95-26-R, May 1995.

  In this report the first proposal of the new external schema definition methodology; the concepts of transformable and non-transformable classes are first defined in it. It is related to chapters 2, 5 and 6.

- J. Samos, “Definition of External Schemas in Object Oriented Databases,” Proc. Int’l Conf. on Object Oriented Information Systems, Springer, pp. 154-166, Dublin, December 1995.

  This is a shortened version of the previous report, focused on the new methodology of definition of external schemas. It is mainly related to chapter 5.

- J. Samos, J. Sistac, “Definition of Deductive Conceptual Models of OODBs,” Proc. Int’l Workshop on Database and Expert Systems Applications, IEEE Computer Society Press, pp. 313-318, Zurich, September 1996.

  This paper proposes the definition of deductive conceptual models as a prototyping tool, specially suitable for the specification of different components of OODBs. It is covered by chapter 4.

- J. Samos, F. Saltor, “External Schema Generation Algorithms for Object Oriented Databases,” Proc. Int’l Conf. on Object Oriented Information Systems, Springer, pp. 317-332, London, December 1996.

  In this paper two external schema generation algorithms are proposed, they are defined as part of a deductive conceptual model, in the form of derived predicates. It is related to chapter 5, section 4.

- J. Samos, F. Saltor, “Definition of Derived Classes in OODBs Using both Object Preserving and Object Generating Semantics,” submitted for publication.

  In this paper the problems involved in the definition of derived classes are studied; the concept of partially derived class is defined; a proposal is made for transmitting the modifications between derived and base classes. It is related mainly to chapter 6, and also to chapter 7.

Glossary of terms

Aplication administrator	In order to define an external schema, the user of the information contained in the data dictionary is the *application administrator* -through the external schema definition system.
Attribute-identifiable class	We define a class as *attribute-identifiable* if its objects can be identified using a set of its attributes (without type restriction). Derived classes are attribute-identifiable: their objects can also be identified by their core attributes.
Base class	The classes from which a derived class is directly defined are its *base classes*; they can be derived or non-derived classes.
Base object	The objects in base classes that participate in the definition of a derived object are its *base objects*.
Classification	*Classification* is the process of taking a new class description and putting it where it belongs in the class hierarchy.
Conceptual schema	The *conceptual schema* is a logical representation of the reality modeled by the database; it describes the relevant aspects of the universe of discourse.
Core attributes	It can be considered that the object identifier of a derived object is generated from a set of its attributes, these attributes are called *core attributes*.
Data dictionary	The systems of conceptual and external schema definition are based on a *data dictionary*. The universe of discourse of the data dictionary is all information in the management and use of the database system -including the management and use of schemas.
Data relativism	*Data relativism* is the general activity of structuring the same data in different ways. In object schemas, the concept of data relativism is implemented defining external schemas and derived classes.
DCM	Deductive conceptual model.
Derivation relationship	A *derivation relationship* is defined between a derived class and the set of its base classes. The derivation relationship defines how to obtain a derived class from its base classes; it establishes the correspondence between the base objects and the derived objects. The derivation relationship is used to integrate the derived classes into the data dictionary.
Derivation relationship of identity	A base class is related through a *derivation relationship of identity* to a derived class if the objects of the base class participate in the definition of the identity of the objects of the derived class.

Derivation relationship of value	A *derivation relationship of value* existing between a base class and a derived class only if the objects of the base class do not participate in the definition of the identity of the derived objects.
Derived class	*Derived classes* are defined from previously existing classes (derived or non-derived); derived classes offer views of the information contained in the classes from which they are defined. Derived classes are defined during the lifetime of the database in order to be included in some external schema or in the conceptual schema.
Derived object	The objects contained in a derived class are *derived objects*.
Dynamic derivation relationship	A *dynamic derivation relationship* is made up of a static derivation relationship and a translator or update policy that determines how to transmit the modifications that are made to the objects of the derived class into modifications to the objects in the base class.
Dynamic derived class	The derived class defined by a dynamic derivation relationship is a *dynamic derived class*.
Enterprise administrator	In order to define the conceptual schema the user of the information contained in the data dictionary is the *enterprise administrator* -by means of the conceptual schema definition system
Equivalence preservation property	In the transmission of modifications from objects of derived classes to the corresponding objects in base classes, the *equivalence preservation property* has to be fulfilled: correct changes in the base objects have to be produced in order to provide the desired updates in derived objects.
Extension (of a class)	The *extension* of a class is its set of occurrences, the set of objects included in it.
External schema	*External schemas* offer views of the information contained in the conceptual schema; they allow the end-user to concentrate on a logical representation of data adapted to their particular requirements.
External schema definition system	The definition of external schemas is carried out by the *external schema definition system*.
Inheritance closure	The object schema requires that for each pair of classes of it that have some property in common, a superclass of them which only has all the properties common to both classes has to be also included in the object schema (this property is called *inheritance closure* of the object schema).
Integration of derived classes	*Integration of derived classes* refers to two different scopes: integration of derived classes and previously existing classes in the data dictionary (or in the conceptual schema playing the role of data dictionary) and integration of a set of classes (derived and/or non-derived) to form an external schema.
Intension (of a class)	The *intension* of a class is made up of the set of properties of that class.

Internal schema	The *internal schema* is a physical representation of the data stored into the database.
Local extension (of a partially derived class)	A partially derived class's *local extension* contains the non-derived elements that are defined both in the class's intension as well as its extension.
Logical association	The relationship expressed by way of the conditions defined in order to associate objects of the base classes to define a derived object is termed *logical association*.
Logical data independence	External schemas provide *logical data independence* (many aspects of the conceptual schema may be changed without having to modify the views of the conceptual schema offered by external schemas).
Non-derived class	*Non-derived classes* are defined during the initial definition of the conceptual schema (it can contain non-derived and also derived classes).
Non-required property	The properties of transformable classes can be required if they are referenced by a non-transformable class, or *non-required* in other case.
Non-side effect external schema	A *non-side effect external schema* is an external schema which is re-computed dynamically so that conceptual schema modifications are (whenever possible) "filtered out" from applications using the external schema.
Non-transformable class	The classes selected to compose the external schema can be qualified as *non-transformable* to indicate that they cannot be modified automatically, in the sense of adding or removing properties in the external schema generation process.
Object generating semantics	A derived class defined by *object generating semantics* contains new objects generated from the objects of its base classes.
Object preserving semantics	A derived class defined by *object preserving semantics* can only contain objects of its base classes.
OID	Object identifier.
OODB	Object-oriented database.
Operation consistency relation	For each method of modification of the base classes or the derived class, an associated operation is defined in the derivation relationship which is run each time the corresponding method is used, each modification method having a defined *operation consistency relation* in the derivation relationship, which is responsible for maintaining the consistency between the base classes and the derived class.
Partially derived class	*Partially derived classes* are derived classes that can contain non-derived information in their intension as well as in their extension.

Physical data independence	The distinction between the the conceptual schema and the internal schema provides *physical data independence* (many aspects fo the physical implementation may be changed without having to modify the abstract vision of the database).
Property	The set of *properties* or the intension of a class is defined as the union of its set of attributes and its set of methods.
Required property	The properties of transformable classes can be *required* if they are referenced by a non-transformable class, or non-required in other case.
Schema closure	A requirement that external schemas have to fulfil is *schema closure*: every class referenced by some class included in an external schema has to be also included in the same external schema.
Static derivation relationship	A *static derivation relationship* is defined between the set of base classes and a derived class, and establishes the correspondence between the base objects and the derived objects.
Static derived class	The derived class defined by a static derivation relationship is a *static derived class*.
Subsumption relationship	Class c_1 is said to subsume class c_2, denoted *subsumes*(c_1, c_2), if and only if c_1 can be defined as a superclass of c_2 in a class hierarchy correctly defined. This means that the type associated to c_1 is a supertype of the type of c_2; and, the set of objects of c_1 <u>always</u> contains the set of objects of c_2.
Temporal external schema	*Temporal external schemas* are external schemas that include non-derived information without having the conceptual schema affected; they can be defined in the test environment.
Test environment	In order to avoid the continual modification of the conceptual schema, the availability of a *test environment* is very useful. In this environment, temporal external schemas can be defined that include non-derived information without having the conceptual schema affected.
Three-level architecture	The ANSI/SPARC *three-level architecture* classified database fuctionalities into physical, logical, and external levels; information at these levels is represented by the internal, conceptual, and external schemas respectively.
Transformable class	The classes selected to compose the external schema can be qualified as *transformable* to indicate that they can be modified automatically, in the sense of adding or removing properties in the external schema generation process.
Translator	A dynamic derivation relationship is made up of a static derivation relationship and a *translator* or *update policy* that determines how to transmit the modifications that are made to the objects of the derived class into modifications to the objects in the base class.
Update policy	See translator.

Value-identifiable class	A class is defined to be *value-identifiable* if its objects can be identified using a set of its attributes that only can be values (not object identifiers).
View	A *view* is a simplifying abstraction of a complex structure. In OODBs, some authors identify the term “view” with the concept of schema; others consider it just a class.

Bibliography

[Abiteboul & Bonner, 1991]	S. Abiteboul, A. Bonner, "Objects and Views," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pp. 238-247, Denver, 1991.
[Abiteboul & Hull, 1988]	S. Abiteboul, R. Hull, "Restructuring Hierarchical Database Objects," *Theoretical Computer Science*, 62, pp. 3-38, 1988.
[Abiteboul et *al*., 1995]	S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley, 1995.
[Alhajj & Arkun, 1993]	R. Alhajj, M. Arkun, "An Object Algebra for Object-Oriented Database Systems," *DATABASE*, vol. 24, no. 3, pp. 13-22, August 1993.
[Andany et *al*., 1991]	J. Andany, M. Léonard, C. Palisser, "Management of Schema Evolution In Databases," *Proc. Int'l. Conf. on Very Large Databases*, pp. 161-170, Barcelona, September 1991.
[Andersen & Reenskaug, 1993]	J. Andersen, T. Reenskaug, "Operations on Sets in an OODB," *OOPS Messenger*, vol. 4, no. 1, pp. 12-25 January 1993.
[ANSI/X3/SPARC, 1975]	ANSI/X3/SPARC Study Group on Database Management Systems, "Interim report," *ACM SIGMOD*, bulletin 7, no. 2, 1975.
[ANSI/X3/SPARC, 1986]	ANSI/X3/SPARC Database System Study Group, "Reference Model for DBMS Standardisation," *SIGMOD Record*, vol. 15, no. 1, pp. 19-58, March 1986.
[Barclay & Kennedy, 1993]	P. Barclay, J. Kennedy, "Viewing Objects," *11th British National Conf. on Databases*, Springer, pp. 93-109, Keele, July, 1993.

[Bertino, 1992] E. Bertino, “A View Mechanism for Object-Oriented Databases,” *Proc. Int’l Conf. on Extending Database Technology,* Springer, pp. 136-151, Vienna, March 1992.

[Bertino et *al.*, 1996] E. Bertino, B. Catania, J. García-Molina, G. Gerrini, “A Formal Model of Views for Object-Oriented Database Systems,” submitted for publication.

[Bratsberg, 1992] S. Bratsberg, “Unified Class Evolution by Object Oriented Views,” *Proc. Int’l Conf. on the Entity-Relationship Approach*, pp. 423-439 , Karlsruhe, October 1992.

[Brèche et *al.*, 1995] P. Brèche, F. Ferrandina, M. Kuklok, “Simulation of Schema Change using Views,” *Proc. Int’l Conf. on Database and Expert Systems Applications*, pp. 247-258, London, September 1995.

[Buchheit et *al.*, 1994] M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt, “Subsumption between Queries to Object-Oriented Databases,” *Information Systems*, vol. 19, no. 1, pp. 33-54, 1984.

[Castellanos et *al.*, 1992] M. Castellanos, F. Saltor, M. García, “A Canonical Model for the Interoperatibility among Object Oriented and Relational Models,” *Proc. Int’l. Workshop on Distributed Object Management*, pp. 309-314, Edmonton, Aug. 1992.

[Costal et *al.*, 1989] D. Costal, J. Pastor, M. Sancho, “Deductive Conceptual Modelling of Systems Using Prolog,” *Proc. IFIP Working Conf.*, pp. 41-57, Barcelona, May 1989.

[Dayal, 1989] U. Dayal, “Queries and Views in an Object-Oriented Data Model,” *Proc. 2nd Int’l. Workshop on Database Programming Languages*, 1989.

[Díaz et *al.*, 1991] O. Díaz, N. Paton, P. Gray, “Rule Management in Object Oriented Databases: A Uniform Approach,” *Proc. Int’l. Conf. Very Large Databases*, pp. 317-326, Barcelona, Sep. 1991.

[Ferrandina et *al.*, 1995] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, J. Madec, “Schema and Database Evolution in the O_2 Object Database System,” *Proc. Int’l. Conf. on Very Large Databases*, pp. 170-181, Zürich, Sep. 1995.

[Gardarin & Yoon, 1996] G. Gardarin, S. Yoon, "On the Power of Views in Hypermedia Databases," *Proc. Engineering Systems Design and Analysis Conf.*, vol. 2, pp. 31-40, Montpellier, July 1996.

[Gentile & Zicari, 1994] M. Gentile, R. Zicari, " Updating Views in Object Oriented Database Systems," *Proc. Int'l. Symposium on Adavanced Database Technologies and their Integration*, Nara, Japan, October, 1994.

[Geppert et *al*., 1993] A. Geppert, S. Scherrer, K.R. Dittrich, "Derived Types and Subschemas: Towards Better Support for Logical Data Independence in Object-Oriented Data Models," Univ. Zürich, Institut für Informatik, Tech. Rep. 93.27, June, 1993.

[Gottlob et *al*., 1988] G. Gottlob, P. Paolini, R. Zicari, "Properties and Update Semantics of Consistent Views," *ACM Transactions on Database Systems*, vol. 13, no. 4, pp. 486-524, December 1988.

[Gray et *al*., 1992] P. Gray, K. Kulkarni, N. Paton, *Object-Oriented Databases. A Semantic Data Model Approach*, Prentice Hall, 1992.

[Gustafsson et *al*., 1982] M.R. Gustafsson, T. Karlsson, J. Bubenko, "A Declarative Approach to Conceptual Information Modelling," *Information Systems Design Methodologies: A Comparative Review*, pp. 93-143, T. Olle, H. Sol, A. Verrijn-Stuart (Eds.), North-Holland, Amsterdam, 1982.

[Heiler & Zdonik, 1988] S. Heiler, S. Zdonik, "Views, Data Abstraction, and Inheritance in the FUGUE Data Model," *Proc. 2nd Int'l. Workshop on OODBS*, Springer, FRG, Sep., 1988.

[Heuer & Sander, 1991] A. Heuer, P. Sander, "Preserving and Generating Objects in the LIVING IN A LATTICE Rule Language," *Proc. Int'l IEEE Conf. on Data Engineering*, pp. 562-569, Kobe, April 1991.

[Heuer & Scholl, 1991] A. Heuer, M. Scholl, "Principles of Object-Oriented Query Languages," *Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft"*, pp. 178-197, Springer, Kaiserslautern, March 1991.

[Hull, 1986]	R. Hull, “Relative Information Capacity of Simple Relational Database Schemata,” *SIAM Journal of Computing*, vol. 15, no. 3, pp. 856-886, August 1986.
[Hull & Yap, 1984]	R. Hull, C. Yap, “The Format Model: A Theory of Database Organization,” *Journal of the ACM*, vol. 31, no. 3, pp. 518-537, July 1984.
[Hull et *al*., 1991]	R. Hull, S. Widjojo, D. Wile, M. Yoshikawa, “On Data Restructuring and Merging with Object Identity,” *IEEE Data Engineering*, vol. 14, no. 2, pp. 18-22, June 1991.
[Kifer et *al*., 1992]	M. Kifer, W. Kim, Y. Sagiv, “Querying Object-Oriented Databases” *Proc. ACM SIGMOD Conference on Management of Data*, pp. 393-402, San Diego, 1992.
[Kim, 1989]	W. Kim, “A model of Queries for Object-Oriented Databases,” *Proc. Int’l Conf. on Very Large Databases*, pp. 423-432, Amsterdam, August 1989.
[Kim & Kelley, 1995]	W. Kim, W. Kelley, “On View Support in Object-Oriented Database Systems,” *Modern Database Systems: the Object Model, Interoperability, and beyond*, W. Kim (Ed.), pp. 108-129, ACM Press, 1995.
[Kimura & Tsuruoka, 1991]	Y. Kimura, K. Tsuruoka, “A View Class Mechanism for Object-Oriented Database Systems,” *Int’l Symp. on Database Systems for Advanced Applications*, pp. 269-273, Tokyo, April 1991.
[Lemke, 1995]	T. Lemke, “DDL = DML ? An Exercise in Reflective Schema Management for Chimera,” IDEA.WP.22.O.003, http://www.ecrc.de/IDEA/, March 1995.
[Miller et *al*., 1994]	R. Miller, Y. Ioannidis, R. Ramakrishnan, “Schema equivalence in Heterogeneous Systems: Bridging Theory and Practice,” *Information Systems*, vol. 19, no. 1, pp. 3-11, 1994.
[Monk, 1994]	S. Monk, “View Definition in an Object-Oriented Database,” *Information and Software Technology*, vol. 36, no. 9, pp. 549-554, 1994.

[Monk & Sommerville, 1993]	S. Monk, Y. Sommerville, “Schema Evolution in OODBs Using Class Versioning,” *SIGMOD Record*, vol. 22, no. 3, pp. 16-22, September 1993.
[Naja & Mouaddib, 1995]	H. Naja, N. Mouaddib, “The Multiple Representation in an Architectural Application,” *Proc. Int’l. Conf. on Database and Expert Systems Applications*, pp. 237-246, London, September 1995.
[Olivé, 1989]	A. Olivé, “On the Design and Implementation of Information Systems from Deductive Conceptual Models,” *Proc. Int’l. Conf. Very Large Databases*, pp. 3-11, Amsterdam, August 1989.
[Peters & Özsu, 1995]	R. Peters, M. Özsu, “Axiomatization of Dynamic Schema Evolution in Objectbases,” *Proc. Int’l IEEE Conf. Data Engineering*, pp. 156-164, Tapei, March 1995.
[Quer & Olivé, 1994]	C. Quer, A. Olivé, “Determining Object Interaction in Object-Oriented Deductive Conceptual Models,” *Information Systems*, vol. 19, no. 3, pp. 211-227, 1994.
[Ra & Rundensteiner, 1995]	Y. Ra, E. Rundensteiner, “A Transparent Object-Oriented Schema Change Approach Using View Evolution,” *Proc. Int’l IEEE Conf. on Data Engineering*, pp. 165-172, Taipei, March 1995.
[Rundensteiner, 1992a]	E. Rundensteiner, “MultiView: A Methodology for Supporting Views in Object-Oriented Databases,” Univ. of Cal., Irvine, Tech. Rep. #92-07, Jan. 1992.
[Rundensteiner, 1992b]	E. Rundensteiner, “A Class Integration Algorithm and its Application for Supporting Consistent Object Views,” Univ. of Cal., Irvine, Tech. Rep. #92-50, May 1992.
[Rundensteiner, 1992c]	E. Rundensteiner, “MultiView: A Methodology for Supporting Views in Object-Oriented Databases,” *Proc. Int’l Conf. on Very Large Databases*, pp. 187-198, Vancouver, Aug. 1992.
[Rundensteiner & Bic, 1992]	E. Rundensteiner, L. Bic, “Automatic View Schema Generation in Object-Oriented Databases,” Univ. of Cal., Irvine, Tech. Rep. #92-15, Feb. 1992.

[Samos, 1995] J. Samos, “Definition of External Schemas in Object Oriented Databases,” *Proc. Int’l Conf. on Object Oriented Information Systems,* pp. 154-166, Dublin, Dec. 1995.

[Santos, 1995] C. Santos, “Design and Implementation of Object-Oriented Views,” *Proc. Int’l Conf. on Database and Expert Systems Applications,* pp. 91-102, London, Sep. 1995.

[Santos et *al*., 1994] C. Santos, S. Abiteboul, C. Delobel, “Virtual Schemas and Bases,” *Proc. Int’l Conf. on Extending Database Technology,* pp. 81-94, Cambridge, March 1994.

[Schewe et *al*., 1992] K. Schewe, J. Schmidt, I. Wetzel, “Identification, Genericity and Consistency in Object-Oriented Databases,” *Int’l Conf. on the Entity-Relationship Approach*, pp. 341-356, Karlsruhe, Oct. 1992.

[Schmolze & Lipkis, 1983] J. Schmolze, T. Lipkis, “Classification in the KL-ONE Knowledge Representation System,” *The Eigth Int’l. Joint Conf. on Artificial Inteligence,* vol. 1, pp. 330-332, Aug. 1983.

[Scholl et *al*., 1992] M.H. Scholl, C. Laasch, C. Rich, H. Schek, M. Tresch, “The COCOON Object Model,” Univ. Ulm, Faculty of Computer Science, Rep. #193, Dec. 1992.

[Scholl & Schek, 1991] M. Scholl, H. Schek, “Supporting Views in Object-Oriented Databases,” *IEEE Data Engineering,* vol. 14, no. 2, pp. 43-47, June 1991.

[Shaw & Zdonik, 1990] G. Shaw, S. Zdonik, “A Query Algebra for Object-Oriented Databases,” *Proc. Int’l IEEE Conf. on Data Engineering,* pp. 154-162, Los Angeles, 1990.

[Skarra & Zdonik, 1986] A. Skarra, S. Zdonik, “The Management of Changing Types in an Object-Oriented Database,” *Proc. Int’l. Conf. on Object-Oriented Programming Systems and Languages Applications,* pp. 383-495, Portland, Sep. 1986.

[Sterling & Shapiro, 1986] L. Sterling, E. Shapiro, *The Art of Prolog. Advanced Programming Techniques*, The MIT Press, 1986.

[Tanaka et *al*., 1988] K. Tanaka, M. Yoshikawa, K. Ishihara, “Schema Virtualization in Object-Oriented Databases,” *Proc. of the 4th Int’l. Conf. on Data Engineering*, IEEE Computer Society Press, pp. 23-30, Feb., 1988.

[Tresch, 1991] M. Tresch, “A Framework for Schema Evolution by Meta Object Manipulation,” *Proc. Int’l Workshop on Foundations of Models and Languages for Data and Objects*, pp. 1-13, Aigen, Sep. 1991.

[Tresch & Scholl, 1992] M. Tresch, M. Scholl, “Meta Object Management and its Application to Database Evolution,” *Proc. Int’l. Conf. on Entity-Relationship Approach*, pp. 299-321, Karlsruhe, Oct. 1992.

[Tresch & Scholl, 1993] M. Tresch, M. Scholl, “Schema Transformation without Database Reorganization,” *SIGMOD Record*, vol. 22, no. 1, pp. 21-27, March 1993.