The syntax of BLOOM99 schemas

Alberto Abelló[1], Marta Oliva[2], M. Elena Rodríguez[1] & Fèlix Saltor[1]
[1]Universitat Politècnica de Catalunya
[2]Universitat de Lleida

June 29, 2000

Abstract

The BLOOM (BarceLona Object Oriented Model) data model was developed to be the Canonical Data Model (CDM) of a Federated Database Management System prototype. Its design satisfies the features that a data model should have to be suitable as a CDM. The initial version of the model (BLOOM91) has evolved into the present version, BLOOM99.

This report specifies the syntax of the schema definition language of BLOOM99. In our model, a schema is a set of classes, related through two dimensions: the generalization/specialization dimension, and the aggregation/decomposition dimension. BLOOM supports several features in each of these dimensions, through their corresponding metaclasses.

Even if users are supposed to define and modify schemas in an interactive way, using a Graphical User Interface, a linear schema definition language is clearly needed. Syntax diagrams are used in this report to specify the language; an alternative using grammar productions appears as Appendix A. A possible graphical notation is given in Appendix B.

A comprehensive running example illustrates the model, the language and its syntax, and the graphical notation.

Resum

El model de dades BLOOM (BarceLona Object Oriented Model) va ser desenvolupat per tal de ser el model canònic de dades (CDM) d'un prototipus de Sistema Federat de Gestió de Bases de Dades. El seu disseny satisfà les característiques que hauria de tenir un model de dades per ser adequat com a CDM. La versió inicial del model (BLOOM91) ha evolucionat fins a la versió actual, BLOOM99.

Aquest report especifica la sintaxi del llenguatge de definició d'esquemes de BLOOM99. En el nostre model, un esquema és un conjunt de classes relacionades al llarg de dues dimensions: la de generalització/especialització, i la d'agregació/descomposició. BLOOM soporta diverses característiques en cadascuna d'aquestes dimensions, mitjançant les metaclasses corresponents.

Encara que se suposa que els usuaris defineixen i modifiquen els esquemes d'una manera interactiva, utilitzant una interfície gràfica d'usuari, és clarament necessari disposar d'un llenguatge lineal de definició d'esquemes. En aquest report es fan servir diagrames sintàctics per especificar el llenguatge. L'apèndix A conté les produccions de la gramàtica, que constitueixen una especificació alternativa. A l'apèndix B es troba una possible notació gràfica.

Al llarg del report es desenvolupa un exemple que il·lustra el model, el llenguatge i la seva sintaxi, i la notació gràfica.

Contents

List of Figures

1 Introduction

Nowadays Object Oriented Databases (OODB) are used in several contexts: CAD, CAE, CIM, etc. and also for inter-operable databases; so Object Oriented (OO) models are needed. This paper describes the syntax that we have to use to define an OODB with BLOOM99 [AORS99] which is the current version of the BLOOM91 (BarceLona Object Oriented Model) model presented in [CSGS91] and [CSGS94].

BLOOM is a semantic extension of an OO model which complies with the essential suitability features of canonical data models for inter-operable databases as defined in [SCGS91]. The main difference between BLOOM and other OO models is that BLOOM has a richer set of semantic abstractions. These BLOOM characteristics can be simulated by a software layer over any OO DBMS.

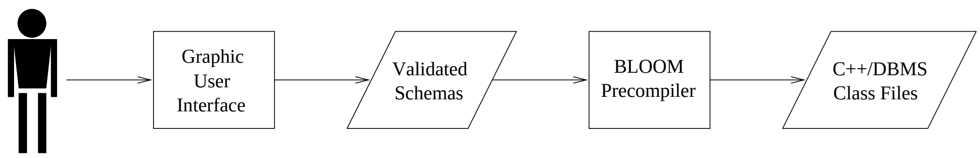

Figure 1: BLOOM code generation process

To define a BLOOM schema we can use a Graphic User Interface (see figure 1) or just a text editor, but the result, named "Validated Schemas", should be the same because both alternatives have to use the established syntax. Later on, using a BLOOM precompiler, the specific DBMS/C++ code will be automatically generated.

This paper is organized as follows: section 2 describes the general structure of a class showing the different BLOOM semantic abstractions. Section 3 and section 4 present the particular syntax of Generalization/Specialization dimension and Aggregation/Decomposition dimension respectively. Section 5 puts all together with a full example. Finally, we can see BLOOM grammar productions in appendix A and an alternative, graphical notation for BLOOM syntax in appendix B.

2 General class structure

The definition of a BLOOM database schema is a set of definitions of BLOOM classes. So, it is important to describe the general structure for each one of them. The syntax used to define a BLOOM class is showed with the diagram in figure 2. After the reserved word *class*, that indicates the beginning of a new class, it is essential to register the class name, and then, between bracets, we can define the class properties as well as its identifier.

In a class, we distinguish two kinds of properties (i.e. particular and inherited) depending on different BLOOM abstractions. Inherited properties are inferred across the generalization dimension, and particular properties are defined using the aggregation dimension. There are different components in the general class structure that allow to define these properties. *generalization* and *specialization* components belong to generalization dimension and we can see their syntax diagrams at section 3; *composition*, *part of*, *aggregates in* and *aggregation of* components

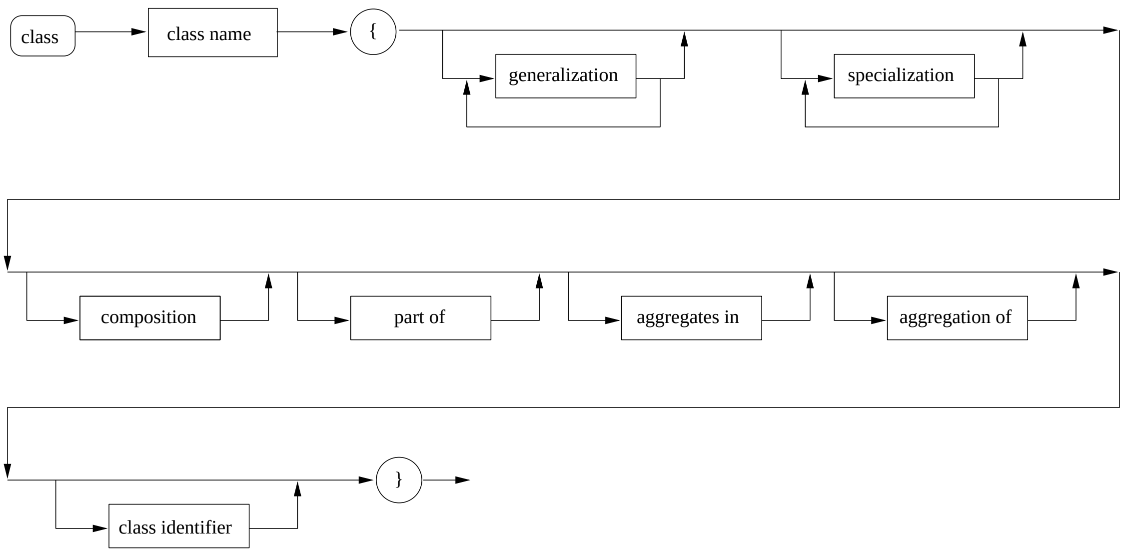

Figure 2: BLOOM class syntax

correspond to aggregation dimension, and we can see their syntax diagrams at section 4.

It is important to note that a class can have several *generalization*, exactly one, or none at all. The same happens to the *specialization*. However, a class can be, at most, a *composition* and *aggregation of*. Symmetrically, a class can be used or not to define properties in other classes (indicated by the existence of an *aggregates in* clause), or parts of other classes (indicated by the *part of* clause).

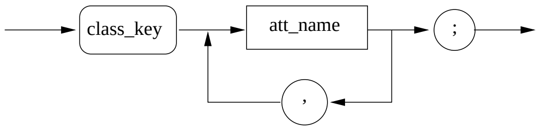

Figure 3: Class identifier syntax

When needed, the class identifier is defined with a list of attribute names (separated by commas) after the reserved word *class_key*, as we can see in figure 3.

3 Generalization dimension

Generalization is the abstraction that allows to extract common properties from similar classes. These common properties are extracted and defined in a generic class (called the superclass) whereas particular properties of each class remain at the specialized classes (known as the subclasses).

BLOOM distinguishes different types of specialization that reflect the relationship between the population of a superclass and the population of their respective subclasses. It has four types of specialization:

Disjoint specialization is used to denote that the population of the specialized subclasses is disjoint. That is, when each object of the superclass belongs, at most, to one subclass.

Complementary specialization is used to represent the fact that the union of population of a set of specialized subclasses is the whole population of the superclass. In this case, each object of the superclass belongs, at least, to one subclass.

Alternative specialization is the conjunction of the previous two kinds of specialization. Therefore, each object of the superclass belongs to one and only one subclass. Subclasses are complementary but they do not overlap.

General specialization is the last kind of specialization. It does not imply any restriction about which subclasses the instances belong to. An instance of the superclass can be instance or not of any of its subclasses.

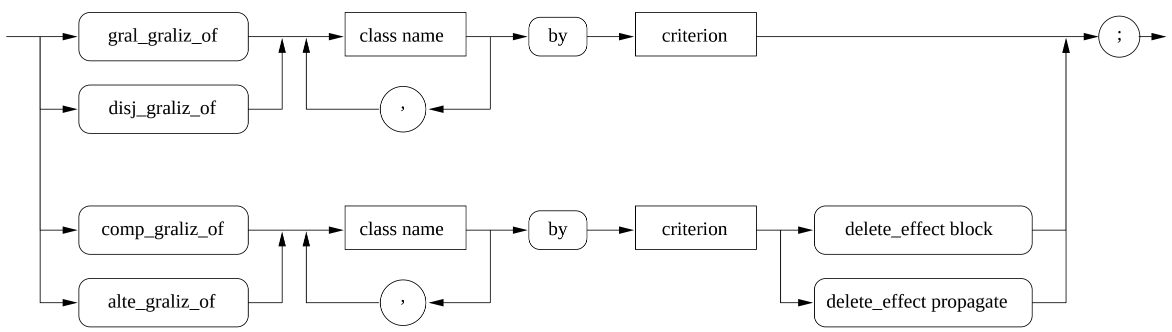

Figure 4: Generalization syntax

Given a superclass, it could be specialized in more than one group of subclasses accordingly to different criterions. These criterions allow to distinguish the specialization used. Conversely, each subclass could be generalized as one or more superclasses, that is, BLOOM supports multiple generalization, commonly known as multiple inheritance.

Figure 4 shows the syntax for generalization dimension. From the point of view of the superclass, it can be either general generalization (*gral_graliz_of*), disjoint generalization (*disj_graliz_of*), complementary generalization (*comp_graliz_of*) or alternative generalization (*alte_graliz_of*) of a group of subclasses accordingly to a *criterion*. It is mandatory to indicate the name of the subclasses (at least one) involved in a given type of generalization (separated by commas), as well as the criterion used.

In case of having complementary or alternative generalization, it is also necessary to indicate the impact of the deletion of an object from one of the subclasses to its superclass, because in both cases, we have to maintain the defined restrictions between the population of the superclass and those of the subclasses. This is called the delete effect and we have two possibilities:

Delete effect block , the object cannot be removed from a subclass if it does not exist in another sibling subclass. The reserved words to indicate this kind of delete effect is *delete_effect block*.

Delete effect propagate , the deletion of the object from a subclass, when it does not belong to another sibling subclass, implies also the deletion of the object from the superclass. The reserved words to indicate this type of delete effect is *delete_effect propagate*.

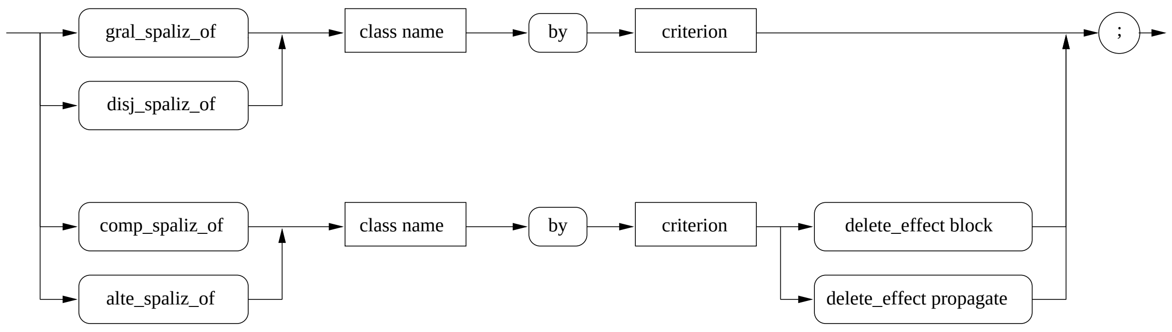

Figure 5: Specialization syntax

In the other hand, given a subclass, we have to indicate which are its superclasses, that is, we must indicate the hierarchies where the subclass is involved, as well as the kind of specialization, the criterions used and the delete effects if needed (see figure 5).

It is important to note that the design has to be consistent, if it defines that a class X is specialized accordingly to a certain criterion in several subclasses Y_1, Y_2,..., Y_n, when it defines each subclass, for example the subclass Y_j, it must indicate that is subclass of class X. Also, the kind of the generalization for Y_j used in the definition of class X should match the kind of specialization for X defined in subclass Y_j. In the same way, the criterion have to coincide.

A partial example that shows the generalization/specialization dimension is depicted in figure 6 and 7. For sake of simplicity, we only concentrate on this dimension. The specification of the classes leaves out components related to the aggregation dimension.

class Person {
disj_graliz_of Teenager, Elder *by* age;
comp_graliz_of American, Spaniard, Briton *by* citizenship *delete_effect block* ;
alte_graliz_of Man, Woman *by* gender *delete_effect propagate*;
gral_graliz_of Student, Employee *by* occupation;
}

Figure 6: Generalization example

In this example, the designer of a BLOOM database schema has defined a class *Person* (see figure 6) which is the generalization of several groups of related classes taking into account different criterions. In this way, class *Person* is the generalization of:

- *Teenager* and *Elder* according to age. The kind of generalization chosen is disjoint. This means that a person, according to her age, will be a teenager or an elder or none of them (e.g. she could be an adult) but not more than one at the same time.

- *Spaniard*, *American* and *Briton* according to citizenship. In this case, the generalization is complementary. This implies that given a person, her citizenship could be one or more of those defined (for instance, people that have double citizenship). It is important to note that all the persons of interest to this specific database schema will have one of the defined citizenships due to the fact that the complementary generalization requires that each object of the superclass belongs, at least, to one subclass.
  The delete effect chosen is *delete_effect block*. This means that, since a person needs a

citizenship, when trying to remove an object from a citizenship, the deletion is blocked if it is not in another sibling class.

- *Man* and *Woman* according to gender. The designer has specified that the generalization is alternative, given that a person only can have a gender at the same time, and always has one. In this case, the delete effect is *delete_effect propagate*. This means that if an object of class *Man* or *Women* is removed, the deletion is propagated to the immediate superclass, that is, to class *Person*.
- *Student* and *Employee* according to occupation. In this case, the designer has chosen general generalization, trying to reflect the fact that a specific person could be a student and/or an employee or none of them.

class Student {
  gral_spaliz_of Person *by* occupation;
  }

class Teenager {
  disj_spaliz_of Person *by* age;
  }

class Spaniard {
  *comp_spaliz_of*Person *by* citizenship*delete_effect block* ;
  }

class Woman {
  alte_spaliz_of Person *by* gender *delete_effect propagate*;
  }

Figure 7: Specialization example

The design should include the same information in the subclasses (see figure 7 - just one subclass for every group). As stated before, the design must be consistent, that is, when it defines each subclass, it has to indicate that the class is specialization of a given class by the same criterion. Thus, it has to indicate that the class *Teenager* is a disjoint specialization of *Person*, the class *Spaniard* is a complementary specialization of *Person*, the class *Men* is an alternative specialization of *Person* and the class *Student* is a general specialization of *Person* according to criterions age, citizenship, gender and occupation respectively.

Finally, note that the delete effect also matches for the cases of complementary and alternative generalization.

4 Aggregation dimension

The aggregation dimension is the abstraction by which several objects are aggregated into a new and complex one. The BLOOM model was conceived to express as much semantics as possible. Thus, it defines two different kinds of aggregation in order to represent the real wold accurately. The kinds of aggregation found in the real world, attending to the strength of the relationship between the aggregate and its attributes, are:

- Simple aggregation, and
- Composition aggregation.

It is important to understand that this dimension shows an object as the aggregation of others. In the same way than the generalization/specialization dimension does, the aggregation has to be written twice: which are the objects aggregated in a complex one, and which are the complex objects an object is aggregated in. However, it must not be seen as the complex object being even a property of its attributes. It is just notation. If it is desired that way, the complex object should appear in the *aggregation_of* clause of its attribute classes.

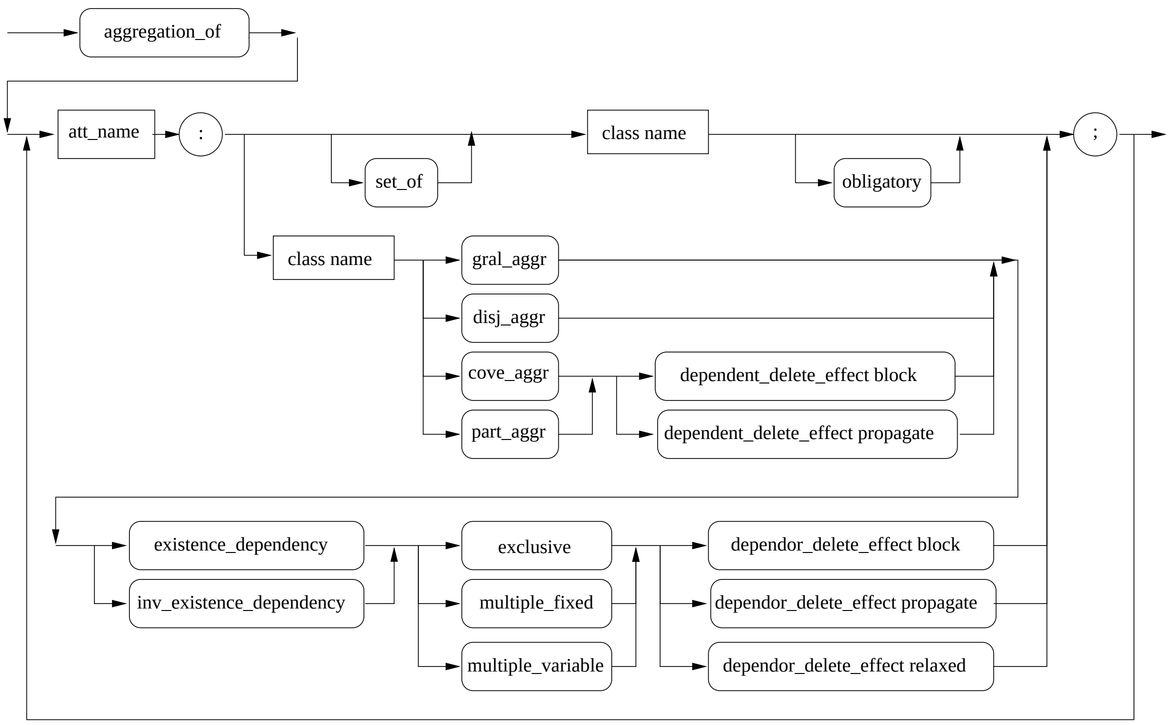

Figure 8: Simple aggregation syntax

4.1 Simple aggregation

This is the kind of aggregation commonly expressed by the OO data models. Using it, the attributes are seen as simple properties of the object being described. They represent the characteristics of interest in the UoD.

In order to make everything to fit well, it is not enough declaring the attributes a class has, by means of the syntax depicted in figure 8. It is also needed to declare where a class is used as attribute. It is done using the syntax described in figure 9. In the future both sides of the declaration will be automatically generated by a GUI[1] avoiding redundant work and making easy the live of the designers.

As seen in figures 8 and 9, these attributes can be mono-valued or multi-valued (*set_of*), and mandatory or not (*obligatory*). Being mandatory means it does not accept null values. At the same time, they could have an existence dependency with the described object.

About the existence dependency, it can be the case the object depending on the attributes or vice versa. Most of the times, the attribute will depend on the aggregate object, because it was created just to represent a given fact of the object (*inv_existence_dependency*). However, it could be conceived the existence of an object depending on that of its attributes, as well

[1]Graphic User Interface

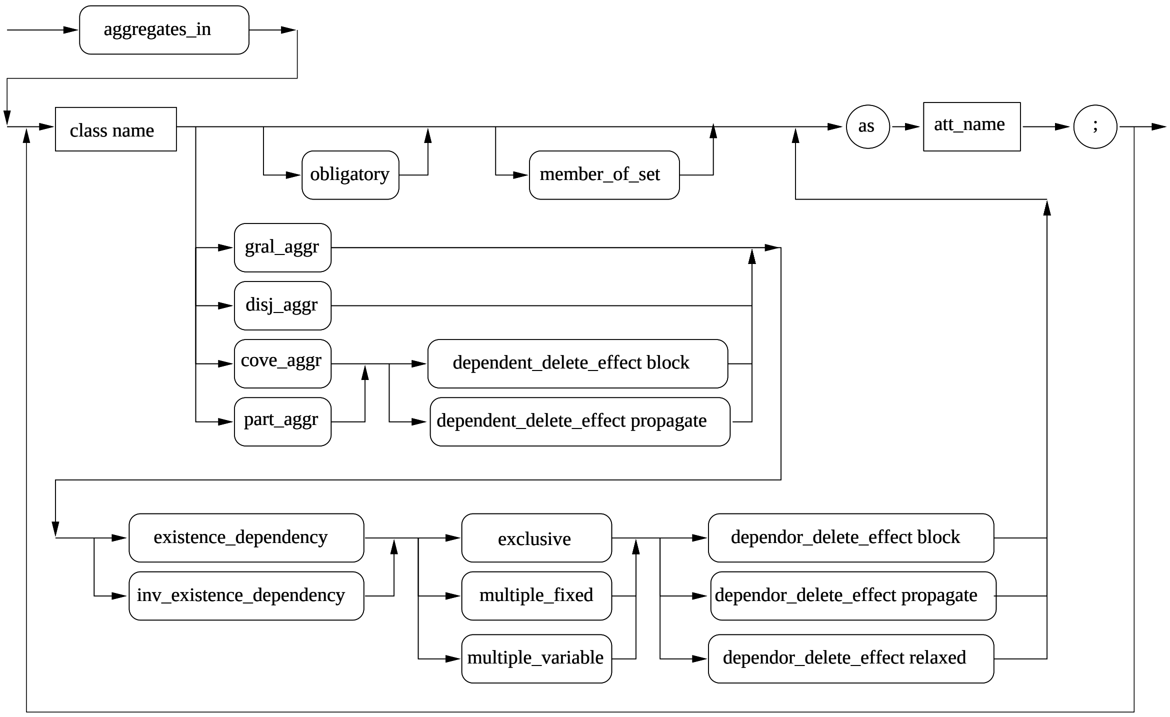

Figure 9: Aggregates syntax

(*existence_dependency*). No matter the direction of the aggregate, the existence dependency can be in the same direction of the aggregation or inverse.

From the point of view of the dependent class, we have the following possibilities about multiplicity and variability of the dependors:

Existence dependency exclusive means that the existence of an object of the dependent class relies exclusively on the existence of one object of the dependor class.

Existence dependency multiple fixed shows that the existence of an object of the dependent class depends on the existence of a fixed set of objects of the dependor class.

Existence dependency multiple variable implies the existence of an object of the dependent class depends on the existence of at least one object from among a variable set of objects of the component class.

When declaring an existence dependency between objects it is mandatory to show the effect the deletion of an object in the dependor side of the relationship has on its dependents. That is, what happens when an object is deleted and any other object depends on its existence. Three different behaviors are considered:

Dependor delete effect propagate propagates the deletion of an object to all those depending on it.

Dependor delete effect block blocks the deletion of the dependor if exists any object depending on it.

Dependor delete effect relaxed , which means it does not matter if there are objects depending on the one being deleted. It just breaks the dependency and deletes the desired object.

Moreover, our model distinguishes four sub-kinds of dependencies, attending to the participation of the dependors in the aggregation:

Disjoint aggregation is the case were each object in the dependor class can be related at most with one dependent. This means that the intersection of any pair of dependencies is empty.

Covering aggregation is the sub-kind of dependency used to model situations where each object of the dependor class must be related with at least one dependent. This implies that the set of all dependor objects must be equal to the extension of the dependor class.

Partitioning aggregation implies each object of the dependor class must have exactly one object depending on it. This means that the list of all dependors must be equal to the extension of the dependor class and any dependor appears only once.

General aggregation has no restrictions about the participation of objects of the dependor class in dependencies.

Taking into account this relationship between the dependor objects and their dependents, the designer has to define how the deletion of dependents affect the dependors. This problem arises when there exists a covering or alternative dependency, and the last object depending on another one is removed from the database. Two solutions are offered:

Dependent delete effect block models the case were an object of the dependent class can not be removed if it is the last one depending on an object of the dependor class which should have at least one depending on it.

Dependent delete effect propagate means that the deletion of an object of the dependent class will entail the deletion of the dependor one if it is the last one depending on it.

To illustrate the simple aggregation, an example is shown in figure 10. It describes a person with properties: name, age, being a female, telephone numbers and car owned. The *name* attribute is a string of characters representing the name of the person which cannot be null; *age* is an integer value and *female* is a boolean value indicating whether the person is a female or not. A set of telephone numbers is kept in the *phones* attribute. The last attribute is *car* and has an existence dependency.

The existence dependency between the classes *Person* and *Car* means the existence of a given car depends on being owned by a person. A *Car* instance must be referenced in exactly one *Person* object (*inv_existence_dependency exclusive*). Thus, a car has to be created being attribute of a person and when the person is deleted, it is propagated (*dependor_delete_effect propagate*) to his car (the car is not of interest anymore). Besides, this dependency is covering (*cove_aggr*) which means every person must have at least a car depending on her. This implies that a delete behaviour have to be defined to prevent the last car of a person being removed. In this case, block (*dependent_delete_effect block*) means that the last car of a person cannot be deleted.

```
class Person {
  aggregation_of
    name : String obligatory ;
    age : Int ;
    female : Boolean ;
    phones : set_of String ;
    car : Car Cove_aggr dependent_delete_effect block
      inv_existence_dependency exclusive dependor_delete_effect propagate ;
  }

class Car {
  aggregates_in
    Person cove_aggr dependent_delete_effect block
      inv_existence_dependency exclusive dependor_delete_effect propagate as car ;
  }
```

Figure 10: Simple aggregation example

4.2 Composition aggregation

The composition aggregation abstraction allows the designer to model classes of objects (called aggregate classes) as the aggregation of a fixed set of objects that belong to different classes (known as component classes). This means that the aggregation of objects which belong to the component classes (known as the component objects) give rise to a new complex object, simply called the aggregate object. It could be seen as a stronger simple aggregation which requires an existence dependency between aggregating and aggregate, where the aggregate depends on the existence of its aggregatings.

Component objects are not simple properties of the aggregate object, but they are necessary for the existence of the aggregate object. In other words, the existence of the aggregate object has no sense without one of its parts. Obviously, the aggregate object can have its own properties which can be specified in the definition of the aggregate class by the simple aggregation abstraction explained before.

The behavior of this abstraction is given by the dependencies that the designer establishes between the objects of the component classes and the objects of the aggregate class. As can be seen in figure 11, a composed object can have a dependency relationship with a single object (*exclusive*) or with a set of them (*multiple*). If it is the case of a *multiple* dependency, it can rely on the existence of every element in the set (*fixed*) or in the existence of at least one object in the given set, which can change (*variable*).

From the point of view of the objects of the component classes it is mandatory to specify the impact of the deletion of these objects in the objects of aggregate class where they participate. Different situations can arise:

Dependor delete effect propagate implies propagating the deletion of an object of the component class to the objects of the aggregate class where the component object to be removed participates.

Dependor delete effect block preserves an object of the component class against being removed if it is part of some object of the aggregate class.

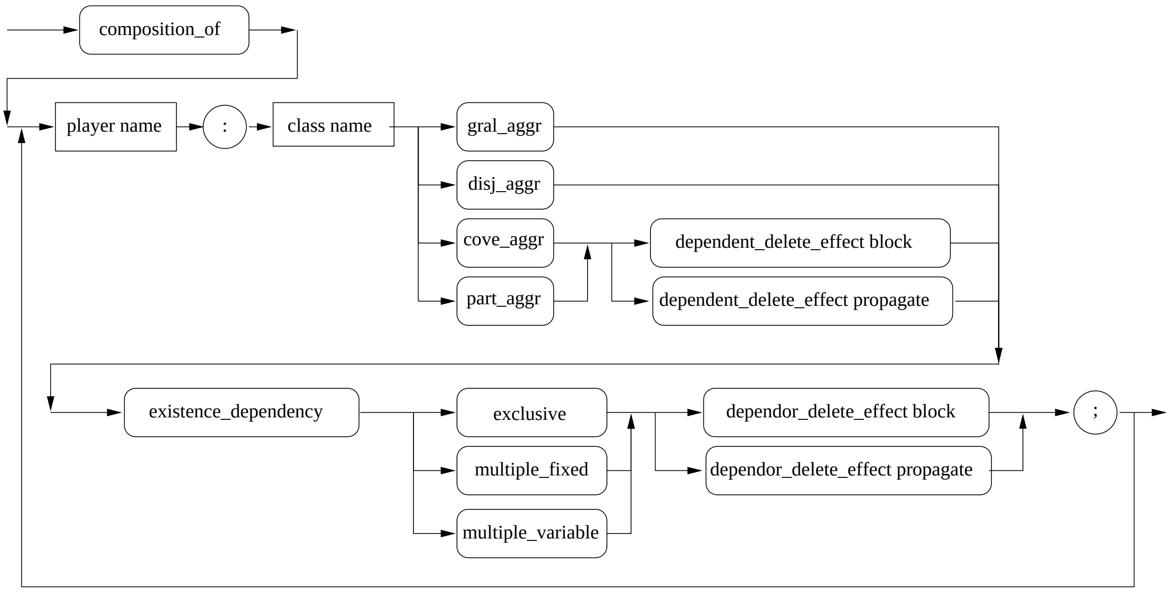

Figure 11: Composition syntax

Moreover, the relations between the population of the component classes and the composed one has to be defined. It means that, as well as in the simple aggregation, for every existence dependency, it should be specified whether it is covering or partitioning (*gral_aggr*, *disj_aggr*, *cove_aggr*, or *part_aggr*). This also entails the designation of a dependent (the composed object) delete effect for coverage and partitioning aggregations. This dependent delete effect (i.e. *propagate* or *block*) describes what happens when trying to delete a composed object, and it is the last composition in which one of its parts participates. As always, block means the deletion of the composed object is blocked, and propagates shows that the deletion will be propagated to the parts.

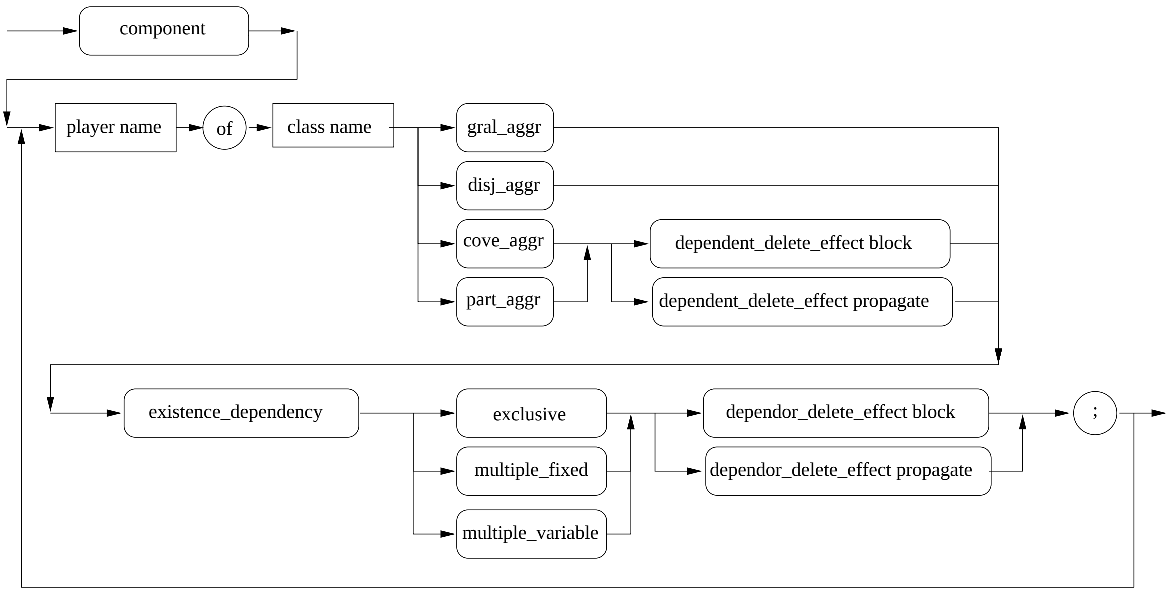

Figure 12: Part syntax

While figure 11 depicts the syntax of the composition aggregation abstraction from the point of view of the aggregate class, figure 12 points it out from the point of view of the component class. It is important to note that the designer will specify only one of them and the tool associated to the GUI will fill in the corresponding counterpart taking into account the designer's specifications in order to avoid redundant work.

If we examine the syntax of composition aggregation, from the point of view of aggregate class (figure 11) it is mandatory to define the role that each component class plays in the aggregate class. Meanwhile, as showed in figure 12, every part contains the role it plays in every class.

```
class Car {
  composition_of
    chassis : Chassis disj_aggr existence_dependency exclusive dependor_delete_effect propagate ;
    engine : Engine disj_aggr existence_dependency exclusive dependor_delete_effect block ;
    wheels : Wheel disj_aggr existence_dependency multiple_fixed dependor_delete_effect block ;
  aggregation_of
    color : String ;
  }

class Chassis {
  component
    chassis of Car disj_aggr existence_dependency exclusive dependor_delete_effect propagate;
  aggregation_of
    identifier : Integer ;
  }

class Wheel {
  component
    wheels of Car disj_aggr existence_dependency multiple_fixed dependor_delete_effect block;
  }
```

Figure 13: Composition aggregation example

Figure 13 depicts an example of composition aggregation abstraction. Our example defines the class *Car* as an aggregate class of component classes *Chassis*, *Engine* and *Wheels*. The existence dependencies established by the designer between the component class *Car* and its parts have been *existence_dependency exclusive* for *Chassis* and *Engine* component classes and *existence_dependency multiple_fixed* for component class *Wheels*. That means that for the existence of a specific car X, it is mandatory the existence of exactly one object (named Y) of component class *Chassis*, the existence of exactly one object (named Z) of component class *Engine* and the existence of a set of objects (W_1, W_2, W_3 and W_4 in this case, although our syntax does not allow specify cardinalities) of the component class *Wheels*.

The delete effect defined by the designer has been *dependor_delete_effect propagate* for component class *Chassis*. This means that if a specific object of class *Chassis*, for instance, chassis Y is deleted, then the aggregate objects where chassis Y participates (car X) should also be deleted. On the other hand, the delete effects specified for component classes *Engine* and *Wheels* have been *dependor_delete_effect block*. This implies that a specific *Engine* (engine Z) can not be deleted if it participates in a specific aggregate object (car X). Therefore, the deletion of engine Z or any of the wheels W_i would be rejected while car X exist.

Moreover, all dependencies are declared disjoint (*disj_aggr*) meaning two cars cannot share neither the engine, nor the chassis, nor any of their wheels. That is, a component can, at most, be a part of a car.

5 Putting all together

The objective of this section is to illustrate through an example all BLOOM dimensions explained in previous sections. Figures from 14 to 17 show the definition of the classes according to the BLOOM syntax fixed in this paper. To better understand it, the same example is depicted in a graphical format in appendix B.

```
class Person {
  comp_graliz_of Employee, Customer by enterpriseRelation delete_effect block ;

  aggregation_of
    name : String obligatory ;
    address : String ;
    telephone : String ;
  }

class Employee {
  gral_graliz_of Driver by post ;
  comp_spaliz_of Person by enterpriseRelation delete_effect block ;

  aggregation_of
    salary : Int ;
    employeeNumber : Int ;
  class_key employeeNumber ;
  }

class Customer {
  comp_spaliz_of Person by enterpriseRelation delete_effect block ;

  aggregates_in
    Pack cove_aggr dependent_delete_effect propagate existence_dependency exclusive
      dependor_delete_effect block as owner ;
  aggregation_of
    customerNumber : Int ;
  class_key customerNumber ;
  }

class Driver {
  gral_spaliz_of Employee by post ;

  aggregation_of
    yearsOfExperience : Int ;
    vehicles : Vehicle gral_aggr inv_existence_dependency multiple_fixed dependor_delete_effect relaxed ;
  }
```

Figure 14: Person, Employee, Customer and Driver classes

The example shows a (partial) database schema for a courier company. In order to explore it in a exhaustive manner, firstly we will proceed along generalization/specialization dimension, and after we will examine the aggregation dimension.

5.1 Generalization/Specialization dimension

The example depicts four classes which are specialized in different groups of subclasses. Each one of those specializations belongs to one of the four different kinds of specializations explained in section 3.

```
class TransportUnit {
  alte_graliz_of Vehicle, Convoy by numberOfMembers delete_effect propagate ;

  component
    container of Shipment disj_aggr existence_dependency exclusive dependor_delete_effect block ;
  aggregation_of
    unitNumber : Int ;
  class_key unitNumber ;
  }

class Vehicle {
  alte_spaliz_of TransportUnit by numberOfMembers delete_effect propagate ;
  disj_graliz_of Van, Truck by kind ;

  aggregates_in
    Driver gral_aggr inv_existence_dependency multiple_variable dependor_delete_effect relaxed as vehicles;
  aggregation_of
    plate : String ;
  class_key plate ;
  }

class Convoy {
  alte_spaliz_of TransportUnit by numberOfMembers delete_effect propagate ;

  composition_of
    line : Truck gral_aggr existence_dependency multiple_variable dependor_delete_effect block ;
  }

class Van {
  disj_spaliz_of Vehicle by kind ;

  aggregation_of
    capacity : Int ;
  }

class Truck {
  disj_spaliz_of Vehicle by kind ;

  component
    line of Convoy gral_aggr existence_dependency multiple_variable dependor_delete_effect block ;
  aggregation_of
    capacity : Int ;
    maxLoad : Int ;
  }
```

Figure 15: TransportUnit, Vehicle, Convoy, Van and Truck classes

Class *Person* is specialized in subclasses *Employee* and *Customer* through the criterion *enterpriseRelation*. The kind of specialization chosen is a complementary specialization. This means that we only are interested in those persons that work in our company or ask for our services. We do not admit instances of *Person* that are neither instances of *Employee* nor *Customer* classes. Given that we have chosen a complementary specialization, we allow the case that one of our workers ask for the services of the company, i.e, we can have persons that are at the same time employees and customers. In this case, the delete effect that we have defined is *delete_effect block*. Therefore the deletion of an object of class *Employee* will be accepted if it exists in the sibling class *Customer*. If this was not the case the deletion will be rejected. The same reasoning can be applied for deletion of objects of class *Customer*.

Class *Employee* is specialized in class *Driver* by *post* criterion. This means that a part of our employees have car license and act as drivers of some vehicle of our company. Given that a general specialization has been chosen, it does not imply any additional relationship between instances of *Employee* and *Driver*, i.e. a specific employee could be a driver or not.

Class *TransportUnit* is specialized into class *Vehicle* and class *Convoy* depending on the number of elements of transport needed for a shipment (i.e. the criterion is *numberOfMembers*). It is an alternative specialization, so each object of class *TransportUnit* should be either a *Vehicle* or a *Convoy*, but only one of them. In this case, the delete effect we have defined is *delete_effect propagate*. Thus, the deletion of an object of class *Vehicle* or class *Convoy* will be always accepted and it will cause the deletion of the corresponding object at the superclass *TransportUnit*.

Finally, class *Vehicle* is specialized by criterion *Kind* in *Van* and *Truck* classes. In this case we have chosen a disjoint specialization, thus a specific vehicle could be either a van or a truck (but only one of them) or perhaps none of them (for instance a specific vehicle could be a car or a motorcycle).

5.2 Aggregation dimension

As we explained in section 4, aggregation dimension allows to aggregate several objects into a new complex one. Our example shows all kinds of aggregation that can be defined in BLOOM. Following the strategy used in section 4, we will examine every kind independently.

5.2.1 Simple aggregation

This abstraction allows to describe relevant properties of the objects that belong to a specific class. We do not intend to explain all properties defined for all classes of our example. Instead of this, we will concentrate on the more interesting ones, i.e. where existence dependencies appear; this is the case of class *Pack* and class *Driver*.

```
class Pack {
  component
    content of Shipment part_aggr dependent_delete_effect block existence_dependency multiple_fixed
      dependor_delete_effect propagate ;

  aggregation_of
    volume : Int ;
    weight : Int ;
    deliveryAddress : String obligatory ;
    items : set of String ;
    owner : Customer cove_aggr dependent_delete_effect propagate existence_dependency exclusive
      dependor_delete_effect block ;
  }
```

Figure 16: Pack class

The relevant properties to objects of class *Pack* (see figure 16) are their volume (represented by an *Integer*), their weight (also represented by an *Integer*), their delivery address (represented by a *String* and being obligatory), a set of descriptions of the items that contains (represented by a set of *String*), and their owner; the owner of a specific pack is an object of class *Customer*, representing the specific customer who wants to send the pack. This last property expresses an existence dependency between objects of class *Pack* and objects of class *Customer*. In other words, the existence of a pack depends on the existence of the customer who sends it. Given that one pack will belong only to a customer, we have defined an *existence_dependency exclusive*. About the dependor delete effect (just recall that it is mandatory to define it as we stated in section 4), we have chosen a *dependor_delete_effect block*. This means that a customer could not be erased from class *Customer* if she has pending packs. Note that does not imply anything about the deletion of the packs, because they are the dependents of the relationship. However, since we are only interested in clients if they are sending packs, this dependency covers the *Customer* class (*cove_aggr*), which means each customer is sending, at least, a pack; and thus, when the last pack of a customer has been sent, the customer is not of interest anymore and is removed from the database (*dependent_delete_effect propagate*).

Conversely, in class *Customer* it is needed to declare that it is being used as attribute in class *Pack*. This is done through the statement *aggregates_in* (see figure 14) which indicates that class *Customer* appears in class *Pack* as *owner* attribute. It is important to be aware of the packs of a customer are not an attribute of class *Customer*. To have the set of packs as an attribute, the class *Customer* should have an attribute of class *Pack* in its *aggregation_of* clause. Having it only in its *aggregates_in* clause is merely informative.

On the other side, class *Driver* has the following attributes: the years of experience and his assigned vehicles (see figure 14). These are its own attributes, the specific ones to class *Driver*. Besides, from the point of view of simple aggregation, we have already mentioned one of the main implications of the generalization/specialization dimension: the inheritance of properties from superclasses. Just a comment on this matter; class *Driver* has its own attributes (*yearsOfExperience*) and inherits attributes from its superclasses (*Employee* and *Person*). The inherited attributes are: *name*, *address*, *telephone*, *salary* and *employeeNumber*.

Paying attention just to the *vehicles* attribute, it represents the specific vehicles (objects that belong to class *Vehicle*) assigned to a given driver. Conversely, in class *Vehicle*, it is needed to declare that it is being used as attribute in class *Driver*. This is done through the statement *aggregates_in* (see figure 15) which indicates that class *Vehicle* appears in class *Driver* as *vehicles*. Besides, there is an existence dependency in the inverse direction of the aggregation (*inv_existence_dependency*). The vehicles are aggregated to characterize drivers, but the existence of the vehicles depends on the drivers.

In this case, the kind of aggregation defined is general (*gral_aggr*), which means that being assigned to a car or not does not affect the drivers at all, in spite of the existence of drivers do affect the vehicles. The *inv_existence_dependency multiple_variable* implies that the existence of a vehicle depends on the existence of at least one driver for it. The *dependor_delete_effect relaxed* means that the deletion of drivers is allowed independently of whether they have or have not some assigned vehicles. If the driver to be removed is the last driver of some vehicle, the dependency between these objects is just broken.

The above explanation can seem contradictory, on the one hand we have defined an existence dependency between objects of classes *Vehicle* and *Driver* (specific vehicles depend on the existence of their drivers), on the other hand we allow later breaking this dependency. In fact, we have defined the initial conditions for the existence of new vehicles (*existence_dependency multiple_variable*) and how their situation can evolve (*dependor_delete_effect relaxed*). Translating this idea to our example, the courier company only will buy new vehicles in case that there are available drivers for them. After the purchase, vehicles could be reassigned to different drivers, and it is also possible that during a period of time a vehicle is not assigned to any driver.

5.2.2 Composition aggregation

As a case of a composition abstraction, we have chosen class *Convoy* (see figure 15). A specific object of class *Convoy* is a set of objects of class *Truck*. The kind of aggregation defined is *gral_aggr*; this means that there is no restriction about the participation of objects of the component class in objects of the composed one. If we translate this to our example, it means that a given truck can participate in different convoys, and does not have to participate in any if it is not needed.

Regarding to the relationship between the aggregate object and the underlying objects that constitute it, we have to define how the deletion of objects of class *Truck* affect complex objects of class *Convoy*. We have chosen *dependor_delete_effect block*, so the deletion of a specific truck will be rejected if it is the last object member of some aggregate object convoy.

Finally, from the point of view of the underlying class *Truck*, its definition shows that it is component (or it is part) of class *Convoy*. Notice again that this is just notation. A truck object does not keep any information about the convoys it is in, if the information is not kept in one of its attributes.

```
class Shipment {
  composition_of
    content : Pack part_aggr dependent_delete_effect block existence_dependency multiple_fixed
      dependor_delete_effect propagate ;
    container : TransportUnit disj_aggr existence_dependency exclusive dependor_delete_effect block ;

  aggregation_of
    date : String ;
    origin : String ;
    destination : String ;
  }
```

Figure 17: Shipment class

Our example shows class *Shipment* as another case of compound class (see figure 17). The existence of a specific shipment depends on the existence of its parts, i.e. objects of classes *Pack* (as the content of a shipment) and *TransportUnit* (as the container of a shipment). From the point of view of the aggregate class *Shipment*, we have specified that the existence of a specific shipment relies on:

- The existence of a fixed set of objects of the component class Pack (*existence_dependency multiple_fixed*), and
- The existence of one object of component class TransportUnit (*existence_dependency exclusive*).

In other words, a shipment is a complex object formed by the packages that are sent for different customers and by the media of transport to be used in their delivery.

In a similar way, from the point of view of component classes, we have defined the impact of deletions of these objects in the objects of the complex class *Shipments* where they can potentially participate:

- The deletion of an object of class *Pack* implies propagating the deletion of the object (if there is any) of class *Shipment* where it participates (*dependor_delete_effect propagate*).
- The deletion of a specific transport unit is rejected (*dependor_delete_effect block*) if it is part of some object of class *Shipment*.

Moreover, both components have been defined as different kinds of aggregation (i.e. *part_aggr*, and *disj_aggr*). The container being a disjoint aggregation of *TransportUnit* means every transport unit is allowed to be assigned only to a shipment at the same time. On the other hand, the content partioning the packs implies a pack is always assigned to exactly one shipment, which forces to define a delete effect when a shipment is removed from the database. In this case, *dependent_delete_effect block* means a shipment cannot be deleted if its packs are not previously assigned to any other shipment.

Finally, note that the definition of class *Pack* and class *TransportUnit* reflects the fact that both classes are components of class *Shipment* (just notation again).

Acknowledgements

This work has been partially supported by the Spanish Research Program PRONTIC under project TIC96-0903, and a grant of the Comissionat per Universitats i Recerca of the Generalitat de Catalunya (1998FI 00228).

References

[AORS99] Alberto Abelló, Marta Oliva, Elena Rodríguez, and Fèlix Saltor. The BLOOM model revised: An evolution proposal. in ECOOP'99 Workshop Reader. Springer Verlag (to appear), Juny 1999.

[CSGS91] M Castellanos, F. Saltor, and M. García-Solaco. The Development of Semantic Concepts in the BLOOM Model using and Object Metamodel. Technical Report LSI-91-22, Departament Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, 1991.

[CSGS94] Malú Castellanos, Fèlix Saltor, and Manuel García-Solaco. A canonical model for the interoperability among object-oriented and relational databases. In Ozsu, Dayal, and Valduriez, editors, *Distributed Object Management*, pages 309–314. Morgan Kaufmann, 1994. Proceedings, Int. Workshop on Distributed Object Management, IWDOM, Edmonton, Canada, 1992.

[SCGS91] F. Saltor, M. Castellanos, and M. García-Solaco. Suitability of Data Models as Canonical Models for Federated DBs. *ACM SIGMOD Record*, 20(4):44–48, 1991.

A BLOOM grammar productions

bloom_class $\Rightarrow$ *class* class_name { graliz_spaliz aggregation class_identifier }

graliz_spaliz $\Rightarrow$ generalization specialization

generalization $\Rightarrow$
λ
| *gral_graliz_of* class_name_lst *by* criterion; generalization
| *disj_graliz_of* class_name_lst *by* criterion; generalization
| *comp_graliz_of* class_name_lst *by* criterion delete_strict_effect; generalization
| *alte_graliz_of* class_name_lst *by* criterion delete_strict_effect; generalization

specialization $\Rightarrow$
λ
| *gral_spaliz_of* class_name *by* criterion; specialization
| *disj_spaliz_of* class_name *by* criterion; specialization
| *comp_spaliz_of* class_name *by* criterion delete_strict_effect; specialization
| *alte_spaliz_of* class_name *by* criterion delete_strict_effect; specialization

aggregation $\Rightarrow$ composition part_of simple_of simple_in

composition $\Rightarrow$
λ
| *composition_of* component_lst

component_lst $\Rightarrow$
player_name: class_name kind_of_aggr dependency_type depr_delete_strict_effect;
| player_name: class_name kind_of_aggr dependency_type depr_delete_strict_effect; component_lst

part_of $\Rightarrow$
λ
| *component* part_lst

part_lst $\Rightarrow$
player_name *of* class_name kind_of_aggr dependency_type depr_delete_strict_effect;
| player_name *of* class_name kind_of_aggr dependency_type depr_delete_strict_effect; part_lst

simple_of $\Rightarrow$
λ
| *aggregation_of* simple

simple $\Rightarrow$
sim_without_dep
| sim_with_dep
| sim_without_dep simple
| sim_with_dep simple

sim_without_dep $\Rightarrow$ att_name: attribute_set class_name mandatory;

sim_with_dep $\Rightarrow$
att_name: class_name kind_of_aggr dependency_type dependor_delete_effect;
| att_name: class_name kind_of_aggr inv_dependency_type dependor_delete_effect;

simple_in $\Rightarrow$
λ
| *aggregates_in* ref_simple

ref_simple $\Rightarrow$
ref_sim_without_dep
| ref_sim_with_dep
| ref_sim_without_dep ref_simple
| ref_sim_with_dep ref_simple

ref_sim_without_dep $\Rightarrow$ class_name mandatory attribute_member *as* att_name;

ref_sim_with_dep $\Rightarrow$
class_name kind_of_aggr dependency_type dependor_delete_effect *as* att_name;
| class_name kind_of_aggr inv_dependency_type dependor_delete_effect *as* att_name;

kind_of_aggr ⇒	*gral_aggr*
	| *disj_aggr*
	| *cove_aggr* dependent_delete_effect
	| *part_aggr* dependent_delete_effect
class_identifier ⇒	λ
	| *class_key* att_name_lst;
att_name_lst ⇒	att_name
	| att_name, att_name_lst
class_name_lst ⇒	class_name
	| class_name, class_name_lst
dependent_delete_effect ⇒	*dependent_delete_effect block*
	| *dependent_delete_effect propagate*
depr_delete_strict_effect ⇒	*dependor_delete_effect block*
	| *dependor_delete_effect propagate*
dependor_delete_effect ⇒	*dependor_delete_effect block*
	| *dependor_delete_effect propagate*
	| *dependor_delete_effect relaxed*
delete_strict_effect ⇒	*delete_effect block*
	| *delete_effect propagate*
dependency_type ⇒	*existence_dependency exclusive*
	| *existence_dependency multiple_fixed*
	| *existence_dependency multiple_variable*
inv_dependency_type ⇒	*inv_existence_dependency exclusive*
	| *inv_existence_dependency multiple_fixed*
	| *inv_existence_dependency multiple_variable*
attribute_set ⇒	λ
	| *set_of*
attribute_member ⇒	λ
	| *member_of_set*
mandatory ⇒	λ
	| *obligatory*

B Graphical BLOOM syntax

Due to the complexity of the BLOOM syntax, it is proposed an alternative, more understandable, graphical syntax to design or just show BLOOM schemas. This, together with a GUI, will help future BLOOM designers.

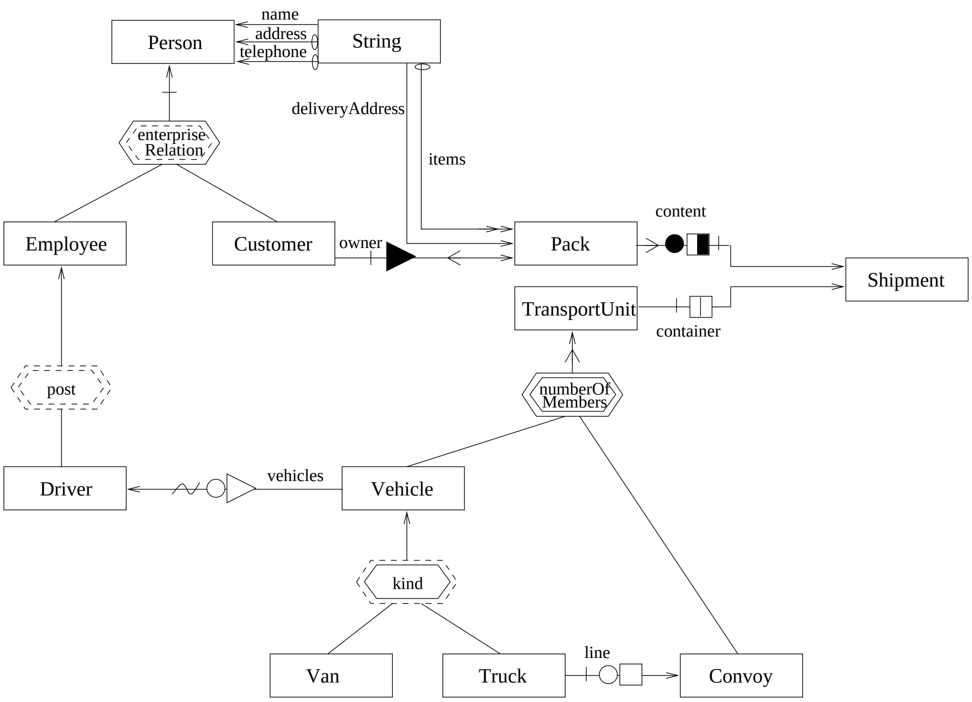

Figure 18: Courier company graphical schema

In figure 18 an example is depicted. It is the same example explained in section 5, which shows a (partial) database schema for a courier company. The generalization/specialization dimension is drawn with vertical lines while aggregation dimension is depicted with horizontal ones. Some attributes are omited. Only those specially illustrative and interesting are drawn.

Along the generalization/specialization dimension we show the different types of specialization the criterion used and the delete effect in case of having complementary or alternative specialization. In the aggregation dimension we have symbols to represent the composition aggregation abstraction, the existence dependencies (both direct and inverse), the delete effects and the participation of the dependors in the simple aggregation (when existence dependencies are defined) or in the composition aggregation (when compound objects exist). A complete list of the symbols (althought some combinations among them are not presented) and their meaning is shown in figure 19.

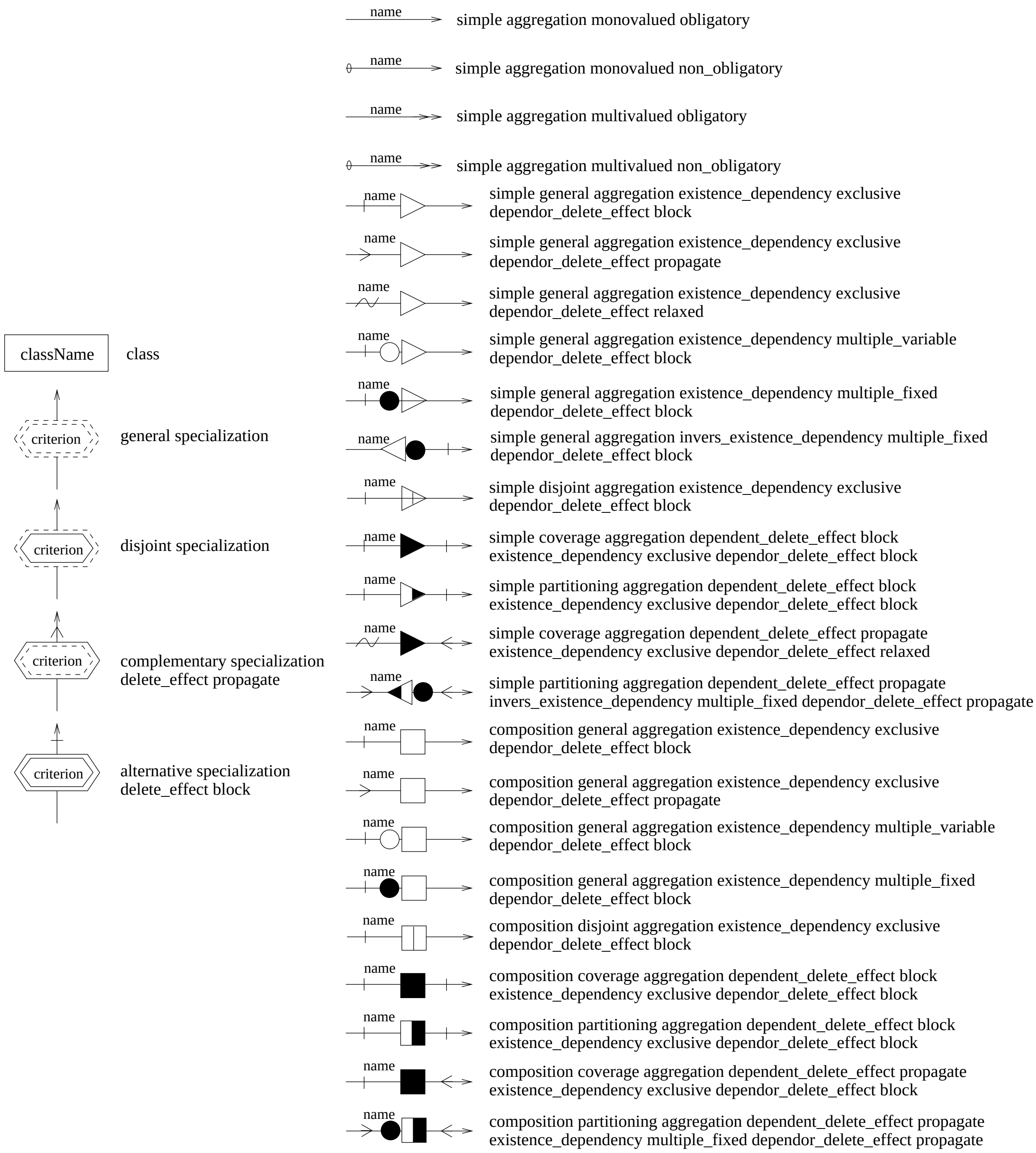

Figure 19: Graphical symbols