Describing BLOOM99 with regard to UML Semantics

Alberto Abelló and M. Elena Rodríguez

U. Politècnica de Catalunya (UPC), Dept. de Llenguatges i Sistemes Informàtics
e-mail: {aabello,malena}@lsi.upc.es

Abstract. In this paper, we describe the BLOOM metaclasses with regard to the Unified Modeling Language (UML) semantics. We concentrate essentially on the Generalization/Specialization and Aggregation/Decomposition dimensions, because they are used to guide the integration process BLOOM was intended for. Here we focus on conceptual data modeling constructs that UML offers. In spite of UML provides much more abstractions than BLOOM, we will show that BLOOM still has some abstractions that UML does not. For some of these abstractions, we will sketch how UML can be extended to deal with this semantics that BLOOM adds.

Key words: Semantics, Conceptual data modeling, Schema integration

1 Introduction

The growing need to share information among several autonomous and heterogeneous DataBases (DBs) has became an active research area. A possible solution to satisfy this need of cooperation is providing integrated access through a Federated Information System (FIS). In order to provide this integrated access, as explained in [SR99], it is necessary to overcome semantic heterogeneities, and represent related concepts. This is accomplished through an integration process in which a Canonical Data Model (CDM) plays a central role.

Once argued the desirable characteristics of a suitable CDM in [SCGS91], and after stating that existing models did not satisfy these characteristics, the BLOOM model (BarceLona Object Oriented Model) was progressively defined in [CKSGS94], and [GSSC95] among others. It results in an extension of an object oriented model with a semantically rich set of abstractions.

The main objective of the BLOOM model is to facilitate the integration of the schemas of different DataBases (DBs) that have decided to cooperate through a FIS. Basically, the goal of the integration process is to identify similarities among classes of different DBs. Our integration approach uses the BLOOM abstractions to achieve this goal by acting at two levels. The first level uses the Generalization/Specialization dimension to decide which group of classes should be compared at each moment; while the second level is guided by Aggregation/Decomposition dimension, and it determines the order of pairwise comparisons between classes in each group.

The need of revising the BLOOM model outcropped during the design process of the directory of the FIS. It is essential to have such storage system because of the amount of needed information in building and operating a FIS. Thus, the model had to be fixed in order to store schemas and mappings (see [SL90]) in a structured manner. This revision gave rise to the BLOOM99 metaclass hierarchy presented in [AORS99b].

This paper is structured as follows: section 2 explains BLOOM99 metaclasses hierarchy; section 3 compares BLOOM99 semantics with UML, mainly attending to the Generalization/Specialization and Aggregation/Decomposition dimensions; section 4 presents some conclusions, and work in progress.

2 The BLOOM99 model

We take the position that the CDM needs a high degree of "representation ability", because the difficult process of "schema integration" requires as much semantics as possible. A previous process, called "semantic enrichment", discovers semantics in the schemas, extensions, application programs, ... of each component database, and expresses it in the CDM. This semantic enrichment is also of use to migrating from one Database Management System (DBMS) to another, and, in general, to better understand the contents of a DB. About its usefulness in the context of Data Mining/Knowledge Discovery you can read [SM98].

As we said in previous section, the schema integration process is driven through the Generalization/Specialization and the Aggregation/Decomposition dimensions. In [AORS99b], the BLOOM model itself has been used as metamodel to describe not only its own Generalization/Specialization dimension, as was previously done, but also part of its Aggregation/Decomposition dimension. Both dimensions include metaclasses that form a semi-lattice. In figure 1 (see [AORS99a] for the BLOOM syntax used in this paper, shorten in the rectangle at the right side of the figure), we can see the sub-hierarchy corresponding to the Generalization/Specialization dimension, while figure 2 shows the sub-hierarchy associated to the Aggregation/Decomposition dimension.

Figure 1 shows how the *Specialization* metaclass constitutes the top metaclass in the Generalization/Specialization dimension. Every *Specialization* is composed by a *superclass, subclass*, and *criterion* used in the specialization of the superclass into the subclass. The *Specialization* metaclass is specialized depending on its complementariness and disjointness. So, a specialization can be complementary or not whether every object in the superclass is at least in a subclass, and disjoint or not whether every object in the superclass is at most in a subclass. These specializations in the metaclass give rise to four different kinds of specialization corresponding to the Cartesian product of the subclasses obtained at both specializations of *Specialization*: complementary (i.e. *Complementary*×*Overlapping*), disjoint (i.e. *Non complementary*×*Disjoint*), alternative (i.e. *Complementary*×*Disjoint*), and general (i.e. *Non complementary*×*Overlapping*). Moreover, in the figure, we can see that complementary

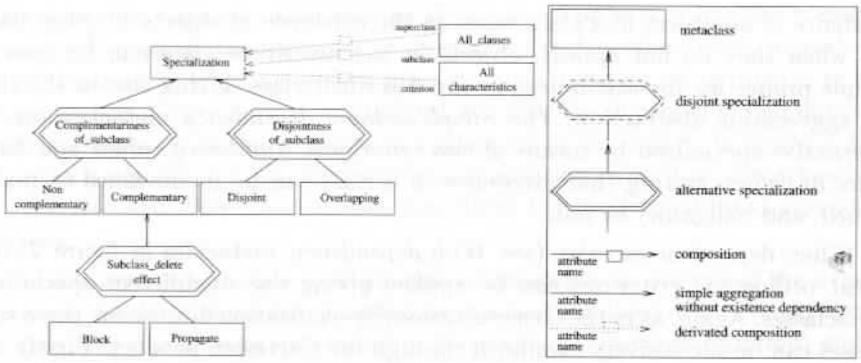

Fig. 1. Generalization metaclasses

and alternative specializations will have two different kinds of delete effect, since in both cases it is not allowed an instance to be member of the superclass if it is not member of any of its subclasses. If an instance of a complementary or alternative specialization is deleted, it can be propagated to the superclass, or blocked until the object is also removed from the superclass by other means.

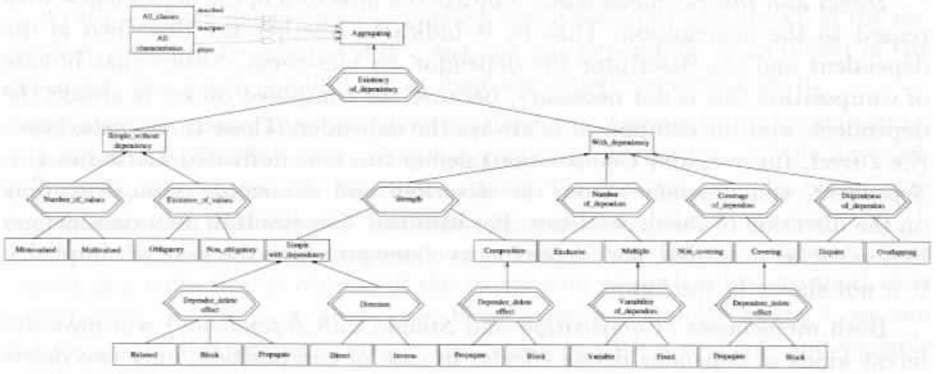

Fig. 2. Aggregation metaclasses

Figure 2 shows the metaclasses that appear along the Aggregation/Decomposition dimension; this dimension allows that several objects can be aggregated into a new and complex one. The *Aggregating* metaclass is the top metaclass in this dimension and it is composed by a *described* (i.e. the class that aggregates objects from other classes), a *descriptor* (i.e. the class which objects should be aggregated in the described class) and a *player* (i.e. the role that the descriptor class plays in the described one).

The *Aggregating* metaclass is specialized depending on the existence of dependencies between the described and descriptor classes into *Simple_without_dependency* and *With_Dependency* metaclasses. If existence dependencies exist, the existence of an object in a class relies on the existence of objects in other class. So, when they do not appear, objects in the descriptor class can be seen as simple properties (or attributes) in the described class, giving rise to the simple aggregation abstraction. The *Simple_without_dependency* metaclass can be alternative specialized by means of two criterions, *Number_of_values* and *Existence_of_values*, stating that attributes in a class can be monovalued or multivalued, and obligatory or not.

When dependencies exist (see *With_dependency* metaclass in figure 2) different orthogonal criterions can be applied giving rise to different specialized metaclasses. Again, as in the Generalization/Specialization dimension, these subclasses can be successively combined through the Cartesian product. Firstly, according to the *Strength* criterion we can distinguish two alternative specialized metaclasses, i.e. *Composition* metaclass and *Simple_with_dependency* metaclass. The *Composition* metaclass corresponds to the composition aggregation abstraction. It allows to model classes of objects (the composed classes) as the aggregation of objects that belong to different classes (the component classes), so that the existence of the composed objects has no sense without the existence of the component objects. When the existence of complex objects can be conceived without the existence of some component object, the semantics embodied in the *Simple_with_dependency* metaclass is applied.

Direct and *Inverse* metaclasses indicate the direction of the dependency with regard to the aggregation. That is, it indicates whether the described is the dependent and the descriptor the dependor, or vice-versa. Notice that in case of composition this is not necessary, because the composed object is always the dependent, and the component is always the dependor. These three metaclasses (i.e *Direct*, *Inverse*, and *Composition*) define two new derivated attributes (i.e. *dependent*, and *dependor*) from the *described* and *descriptor* ones, depending on the direction of the dependency. For instance, *Composition* metaclass defines *dependent* as *described*, and *dependor* as *descriptor*. For the sake of simplicity, it is not shown in the figure.

Both metaclasses (*Composition* and *Simple_with_dependency*) will have different kinds of dependor delete effects. In case of composition, only two delete effects can be applied, i.e. propagate or block. The former propagates the deletion of an object (the dependor) to all those depending on it (the dependents). The other blocks the deletion of dependor objects when dependent objects exist. The *Simple_with_dependency* metaclass allows to specify a third kind of delete effect called relaxed delete effect; if it is applied, dependor objects will be deleted independently of the existence of dependent objects depending on them. Therefore, the relaxed delete effect just allows to break the dependency among dependor and dependent objects, which has no sense in a composition relationship.

As it was said before, other specialization criterions can be applied to the *With_dependency* metaclass. The *Number_of_dependors* criterion states whether objects of dependent class rely on the existence of only one object of the dependor class (i.e. *Exclusive* metaclass) or more (i.e. *Multiple* metaclass). Furthermore, when objects of the dependent class depend on the existence of a set of objects of the dependor class, we must express if this set of objects is fixed or variable. If variable is applied, the existence of objects of the dependent class depends on the existence of at least one object from among a variable set of objects of the dependor class. Conversely, if it is defined as fixed, the dependent class depends on the existence of all the elements in a given set of dependors. In figure 2, this semantics is embodied in the alternative specialization of *Multiple* metaclass into the *Variable* and *Fixed* metaclasses according to the *Variability_of_dependors* criterion.

Finally, attending to the participation of dependors in objects of the dependent class, the *With_dependency* metaclass can be specialized in the *Non_covering* and *Covering* metaclasses (according to *Coverage_of_dependors* criterion) and in the *Disjoint* and *Overlapping* metaclasses (according to the *Disjointness_of_dependors* criterion). So, the participation of objects of the dependor class in objects of the dependent class can be covering or not whether every object in the dependor class must be related to at least one object in the dependent class, and disjoint or not whether each object in the dependor class can be related to at most one object in the dependent class. Moreover, in the figure we can see that *Covering* metaclass can be specialized in the *Propagate* and *Block* metaclasses depending on the *Dependent_delete_effect* criterion. Dependent delete effect propagate implies the deletion of an object of the dependent class will entail the deletion of the object of the dependor class if it is the last one depending on it. When block delete effect is applied, the deletion of one object of the dependent class will be rejected if it is the last one depending on an object of the dependor class which should have at least one object depending on it.

In previous paragraphs we have described the Aggregation/Decomposition dimension in BLOOM, and the simple aggregation abstraction and the composition aggregation abstraction have been shown. The last one, includes both Cover aggregation (see [HM78]) and Cartesian aggregation abstractions (see [SS77]). The Cover aggregation abstraction means that objects of the composed class consist of a collection of objects of the component class. It allows to treat each collection of objects as a single one. If we examine carefully figure 2, we can deduce that Cover aggregations are compositions with a multiple and variable number of dependors.

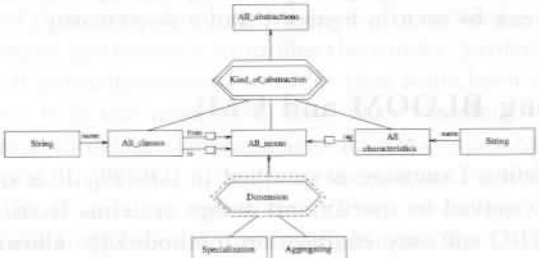

Fig. 3. Top metaclasses and BLOOM99 syntax

Finally, the BLOOM99 metaclasses, along the Generalization/Specialization dimension, form a semi-lattice. The Generalization/Specialization dimension and the Aggregation/Decomposition dimension should be generalized into a common superclass. In figure 3, we can see the top metaclasses of that semi-lattice. *All_abstractions* represents every element in a BLOOM schema, and is specialized by means of an alternative specialization depending on the *kind_of_abstraction* in *All_classes*, *All_nexus*, and *All_characteristics*. *All_classes* is the metaclass containing all the classes in a schema. *All_characteristics* represents the concepts used to link two different classes. Finally, *All_nexus* represents any relationship (called nexus from now on) between two classes.

Every nexus is composed by two related classes and a characteristic showing the reason why they are related. All three monovalued attributes form the key of the *All_nexus* class, univocally identifying each nexus. Besides, *All_classes* metaclass has a general participation in the *from* and *to* components of the nexus, meaning they have neither a covering nor a disjoint participation. However, the deletion of a class is always propagated to all the nexus it is involved in, so they are automatically removed. Moreover, with regards to the participation of characteristics in the nexus, its relationship is covering, meaning that a characteristic must be tagging a nexus to be of interest in the schema. This means we have to define a delete effect to be used whether the last nexus where a characteristic is used in is deleted from the schema. If that is the case, the characteristic will be removed as well (propagate delete effect). In the other way, if somebody would like to remove a characteristic still being used to tag a nexus, it would be blocked.

If we specialize the *All_nexus* metaclass depending on the dimension the nexus appears in (i.e. Generalization/Specialization, or Aggregation/Decomposition), it gives rise to a disjoint specialization into *Specialization*, and *Aggregating* metaclasses. It is a disjoint specialization because we could find nexus along other dimensions in the Object-Oriented (O-O) paradigm (i.e Classification/Instantiation, Behavioural, Dynamicity, or Derivability), as explained in [Sal96]. Therefore, in this case, every instance of the superclass belongs to, at most, one of both subclasses.

Since both, *Specialization* and *Aggregating*, are a specialization of *All_nexus*, they are composed by two classes and one characteristic. However, the role played by those components changes. While a *Specialization* is composed by a *superclass*, *subclass*, and *criterion*; an *aggregating* is composed by a *described*, *descriptor*, and *player* (as it can be seen in figures 1 and 2 respectively).

3 Comparing BLOOM and UML

The Unified Modeling Language is specified in [Obj99]. It is not a data model but a language conceived to specify and design systems. It tries to embrace all the steps in any O-O software engineering methodology, allowing to document the whole process in a standard manner by means of its constructs. Therefore, since data modeling is an important part of the software engineering process, data modeling notation and semantics is an important part of the UML, as well (see [KER99]). In this section, we try to compare BLOOM with regard to UML conceptual modeling semantics, outlying what we miss at each one.

From here on, we are going to show how UML should be extended by adding new metaclasses or stereotypes to be able to capture BLOOM99 semantics. An important limitation of UML to perform this extension is it allows neither multiple classification, nor the usage of more than one stereotype in a given element. Therefore, it forces to define all possible combinations of BLOOM99 metaclasses as specific stereotypes. This is not done in this paper because it seems a mechanical work which would not bring any new information. On the contrary, it would confuse the readers. On defining BLOOM99 metaclasses, orthogonal specializations were found and moved up in the metaclass hierarchy to simplify it and avoid generating unnecessary mixing classes. Thus, generating the Cartesian product of these specializations would undo the work, although there is no problem on doing it. Therefore, it is assumed that the differences found between UML and BLOOM99 will generate a new set of stereotypes to represent any combination of BLOOM99 semantics missing.

3.1 In general

The first thing that catches someone's eye is that UML is much more general than BLOOM. As it was said before, UML is conceived to be used along the whole software engineering process of any system. On the other hand, BLOOM was defined only to represent data schemas to be integrated in the construction process of a FIS. Therefore, due to this reason, UML provides much more abstractions than BLOOM. However, surprisingly, BLOOM still provides some abstractions that UML does not, as we will see below.

One of the main differences found at first glance, when comparing BLOOM and UML, is the existence of *DataTypes*. While UML distinguishes between object types (i.e. classes), and data types whose instances do not have Object Identifiers (OIDs), BLOOM considers that everything is an object and, thus, has an OID. We consider that each thing or concept should have an OID, and the numbers, strings of characters, boolean values, ... are nothing else than concepts. Avoiding unnecessary distinctions simplifies the model. Probably, at the implementation phase, it is much easier to consider that some basic types are already provided. However, it is also possible to have basic standard classes defined. It is absolutely correct to define the operations on data types by means of methods in the corresponding classes. Therefore, there is no reason to have different metaclasses for classes and data types. Having one metaclass (i.e. *All_classes* in BLOOM, or *Class* in UML) should be enough. Maybe, the OID of a number will spend unnecessary extra space, but that is an implementation issue we are not going to discuss here.

3.2 Generalization/Specialization dimension

The nexus in this dimension relate two classes (or metaclasses). One of those classes has a more specific meaning than the other. The more general class is called "superclass" with regard to the specific one, referred as "subclass". As a consequence of this kind of nexus, we obtain inheritance. That is, the subclass inherits the properties and methods of its superclass (or superclasses). Besides, replaceability is also obtained (i.e. where an instance of the a superclass is required, any instance of a subclass could be used). BLOOM as well as UML provide similar abstractions to represent this kind of relationships.

Both provide a discriminated specialization. The UML *Generalization* metaclass has a *discriminator* attribute, and the equivalent BLOOM *Specialization* metaclass has a *criterion* attribute. These attributes allow to classify the instances of a superclass into its subclasses, being used as discriminator or decision criterion. Moreover, both consider the nexus in this dimension as binary relationships, while group them by the value of that attribute (all nexus sharing superclass and criterion must be instance of the same metaclasses). For instance, in the BLOOM metaclasses hierarchy (figure 3), we can see two disjoint specialization nexus: *All_nexus* is specialized into *Specialization*, and *All_nexus* is specialized into *Aggregating*. Therefore, we can say that *All_nexus* is disjointly specialized into either *Specialization* or *Aggregating*, because they share the same criterion. Thus, depending on the *dimension* a nexus is in, it will be instance of one of both metaclasses.

context *Disjoint* **inv**:
self.superclass.allInstances→forAll(x|*Specialization*.allInstances→
→select(n|n.superclass=self.superclass and n.criterion=self.criterion
and n.subclass.allInstances.exists(x)).size≤1)

context *Complementary* **inv**:
self.superclass.allInstances→forAll(x|*Specialization*.allInstances→
→select(n|n.superclass=self.superclass and n.criterion=self.criterion
and n.subclass.allInstances.exists(x)).size≥1)

context *AllNexus::InstanceOf::delete*(c:*AllClasses*, o:*AllObjects*)
pre specializationBlockDeleteEffect: *Specialization::Complementary::Block*.allInstances→forAll(n_1 | n_1.subclass=c implies (deleting(n_1.superclass,o) or *AllClasses*.allInstances→exists(c_2 | $c_2 \neq$ c and c_2.allInstances→includes(o) and *Specialization::Complementary::Block*.allInstances→exists(n_2 | n_2.subclass=c_2 and n_2.superclass=n_1.superclass and n_2.criterion=n_1.criterion))))
post specializationPropagateDeleteEffect: *Specialization::Complementary::Propagate*.allInstances→forAll(n_1 | n_1.subclass=c implies (toBeDeleted(n_1.superclass, o) or *AllClasses*.allInstances→exists(c_2 | $c_2 \neq$c and c_2.allInstances→includes(o) and *Specialization::Complementary::Propagate*.allInstances→exists(n_2 | n_2.subclass=c_2 and n_2.superclass=n_1.superclass and n_2.criterion=n_1.criterion))))

Table 1. Specialization/Generalization rules

However, there are also some differences in this dimension between BLOOM and UML about metaclass representation, nomenclature, etc. On the first hand, as it was explained above, BLOOM has a metaclass (i.e. *Specialization*) which is specialized by two different criterions (two pairs of two metaclasses) giving rise to the four kinds of specialization. Meanwhile, UML also has a metaclass (i.e. *Generalization*), but it uses four "standard" constraints to define the appropriate behaviour (i.e. *complete*, *disjoint*, *incomplete*, and *overlapping*). In UML specification, probably, the applicability and relationship between constraints is not clear enough, but it seems obvious that they exactly match the four corresponding BLOOM metaclasses (i.e. respectively *Complementary*, *Disjoint*, *Non complementary*, and *Overlapping*). In table 1, the constraints associated to every BLOOM specialization metaclass are expressed by means of OCL (Object Constraint Language). Notice that *Non_complementary* and *Overlapping* do not impose any constraint.

The real difference in the kinds of specialization between UML and BLOOM is in that while BLOOM has two subclasses of *Complementary* depending on the associated delete effect, UML does not have the corresponding "subconstraints" of *complete*. In a complementary specialization, every instance of the superclass must be classified in at least one of the subclasses. Therefore, the effect of the "unclassification" of an object from one of the subclasses should be defined. BLOOM considers two different possibilities. The first one, corresponding to the UML behaviour, implies the object is automatically removed from the superclass at the same time it is removed from the subclass (i.e. *Propagate* metaclass). The second possibility is to forbid the deletion of an object from the subclass until it is deleted from the superclass (i.e. *Block* metaclass). Both rules are also shown in table 1. It seems that UML assumes a propagate delete effect since a generalization being complete means the superclass is "abstract" (i.e. it is not allowed to have instances). However, this point is meaningless in UML because it does not consider dynamic classification.

3.3 Aggregation/Decomposition dimension

It is possible to aggregate different objects to obtain a new complex one. This gives rise to aggregation nexus between classes. BLOOM distinguishes different kinds of aggregation. One of them is used to define the simple properties or attributes of a class. In this case, the object is not generally perceived as being the result of the aggregation of these attributes, they are just its properties. An attribute takes values in a given domain, and since BLOOM does not have data types, the domain has to correspond to an objects class. Therefore, every attribute defines a relationship between two classes. In this sense, the *Aggregating* metaclass provides a means for specifying the attributes of a class besides other kinds of relationships. This is one of the main differences with UML, which provides two different metaclasses (i.e. *Attribute*, and *Association*). However, it still considers the named associations as "pseudo-attributes". We think that the information represented in the data schema should be accessible to the user, and thus kept in the database (in software engineering this might not be true). Besides, the association ends should have names in order to be used by users in the navigation paths. Under this assumptions, there are no differences between attributes and associations but in their representation in the schema. The relationships with other classes are also properties of a class. Furthermore, if we identify both representations as a unique concept, there is nothing that avoids choosing the drawing (i.e. an arrow, or a text line in the class rectangle) at user will.

If the simple aggregation does not have any existence dependency (explained in next section), it offers two different characteristics to the attributes whether they are multivalued or not, obligatory or not. Both are represented at once by UML. It is done by the *multiplicity* attribute at every *AssociationEnd* or *Attribute*. The lower level in the multiplicity being zero implies optionality, while the upper level being one implies that the attribute is monovalued. BLOOM does not offer the possibility of fixing an specific multiplicity in the aggregation nexus, because it was not of interest to the integration process.

By means of an attribute (i.e. *aggregation*) in the *AssociationEnd* metaclass, UML considers three different subtypes of association (i.e. *ordinary association*, *composite aggregate*, and *shared aggregate*). The first one, as well as the *attribute* metaclass, would be identified with the BLOOM *Simple aggregation* explained above. The others correspond to a stronger kind of aggregation represented by the *Composition* metaclass in BLOOM. This stronger aggregation does represent a true "Whole/Part" relationship, where the aggregate is conceived as the sum of the parts, and cannot exist without one of them (they are not simple properties). As it will be explained below, there is an existence dependency implicit in a composition.

Existence dependencies Probably, one of the most important (at the same time that confusing) points in BLOOM is its existence dependencies. UML also defines the *Dependency* metaclass. However, its meaning is absolutely different. The UML dependency allows to show an existence relationship between two model elements (at any model in the software engineering process), outlying the importance of one (i.e. supplier) for another (i.e. client). It is not subclass of *Association*, but of the more general metaclass *Relationship*, and its subclasses are *Binding*, *Usage*, *Abstraction*, and *Permission*. On the other hand, BLOOM existence dependencies reflect the necessity of the existence of some class instances (i.e. dependors) for the existence of others (i.e. dependents). Its metaclass (i.e. *With_dependency*) is subclass of *Aggregating*, so that an existence dependency is a kind of aggregation nexus.

If we are talking about a BLOOM composition nexus (corresponding to *composition aggregate* as well as *shared aggregate* in UML), it has an implicit existence dependency associated. That is, the described (whole, or composed object) depends on the existence of the descriptors (parts, or components). A simple aggregation could also have an associated existence dependency. However, in this case, the direction of the dependency is not forced, either the described, or descriptor can play the dependor role. That comes from the nexus being instance of *Direct* (as it is in the composition, the described is the dependent, and the descriptor is the dependor), or *Inverse* (the described is the dependor, and the descriptor is the dependent) metaclass. UML composite or shared aggregate also have implicit existence dependencies. However, UML does not allow to associate an existence dependency to a simple aggregation (i.e. *ordinary association* in UML), so that there is no way to explicit this kind of dependencies.

context *Exclusive* **inv**:
  self.dependent.allInstances→forAll(x|self.dependor.allInstances→
  →select(y|self.allInstances→exists(n|n.dependent=x and n.dependor=y)).size=1)
context *Multiple* **inv**:
  self.dependent.allInstances→forAll(x|self.dependor.allInstances→
  →select(y|self.allInstances→exists(n|n.dependent=x and n.dependor=y)).size≥1)
context *AllNexus::InstanceOf::delete*(c:*AllClasses*, o:*AllObjects*)
  pre dependorDeleteEffectBlock: *Aggregating::With_dependency::Block*.allInstances→forAll(n_1 | n_1.dependor=c implies not n_1.allInstances→exists(ni|ni.dependor=o and tied(ni)))
  post dependorDeleteEffectPropagate: *Aggregating::With_dependency::Propagate*. allInstances→forAll(n_1 | n_1.dependor=c implies (n_1.allInstances→forAll(ni|ni. dependor≠o))) and all dependents have been removed from their respective dependent class
  post dependorDeleteEffectRelaxed: *Aggregating::With_dependency::Propagate*.allInstances→forAll(n_1 | n_1.dependor=c implies (n_1.allInstances→forAll(ni|ni.dependor≠o)))

Table 2. Dependency rules

Table 2 shows rules about BLOOM existence dependencies. Firstly, they can be exclusive (an object depends on exactly one object) or multiple (if the existence of an object depends on more than one object). This could also be shown in UML by the *multiplicity* attribute in the *Association* metaclass. Besides this attribute, UML offers the *changeability* one, which allows three different values (i.e. *changeable*, *frozen*, or *addOnly*). An *addOnly* attribute or association is not considered in BLOOM. Moreover, exclusive existence dependencies are always "frozen" (the dependor cannot change). Nevertheless, the BLOOM multiple dependency (i.e. *Multiple* metaclass) is specialized into *Fixed* or *Variable* attending to the variability of its dependors (that is, it can be respectively frozen or changeable). In BLOOM, Different kinds of changeability are not taken into account out of the scope of existence dependencies.

Although exclusive and multiple fixed dependencies forbid changes in values, a relaxed delete effect allows to remove the dependor so that a new one can be assigned. Meanwhile, the other delete effects (i.e. block, and propagate) provide a similar behaviour to that described for the Generalization/Specialization dimension. The propagate delete effect means that the deletion of the dependor entails that of the dependent, while the block one means that the deletion of a dependor is blocked until it does not have any object depending on it. UML does not consider different kinds of delete effect (propagate delete effect seems always implicit).

Coverage and disjointness As it is in the Generalization/Specialization dimension, the coverage and disjointness participation of objects in existence dependencies should also be taken into account. In this sense, UML distinguishes between composite and shared aggregate. The former implies a component can only take part in an aggregation (which means having a disjoint participation), while the other means that the same object can be aggregated in more than one composition (i.e. overlapping).

On its part, BLOOM99 (as it does for specializations) offers four different kinds of existence dependency attending to the participation of the dependors (which rules are similar to those stated in tables 2 and 1). Thus, aggregating nexus with dependency can be specialized into *Covering* or *Non_covering*, and *Disjoint* or *Overlapping* giving rise to the four mentioned possibilities. As it happened for the complementary specialization, a covering existence dependency offers two different delete effects (i.e. block, and propagate).

4 Conclusions and future work

Along this paper we have described the BLOOM model and discussed about its differences and similarities with regard to UML. On the first hand, it serves to fix BLOOM semantics with regard to a standard language. On the other hand, this paper can help to better understand UML semantics. As noted in [KER99], the paper "Semantics of UML" in [Obj99] actually does not cope with semantics. Thus, further interpretation and explanation is needed.

Surprisingly, leaving out names and nomenclature issues, similarities between BLOOM and UML prevail over differences. General differences include the absence of predefined data types in BLOOM (present in UML) as well as the possibility of multiple classification (not allowed in UML). In the Generalization/Specialization dimension minor differences appear. Basically, differences are related to the delete effects that BLOOM defines and UML does not; in fact, discussion about this matter can be also applied to the Aggregation/Decomposition dimension.

Besides that of the delete effects, related to the Aggregation/Decomposition dimension, major differences appear between BLOOM and UML. A significant difference is related to the fact that BLOOM allows to explicitly specify existence dependencies, while in UML they are inherent to the composite and shared aggregate associations.

It is important to note that all the previous concepts that BLOOM explicitly takes into account, could be expressed in UML for every specific class in an ad-hoc manner, if desired. We have shown how this could be done by means of OCL. In this sense, we found difficulties expressing conditions that involve transitions between different states of the DB.

Our work in progress is intended to make BLOOM99 operative on a centralized DBMS as well as a FIS. With this objective in mind we are working on a *Graphical User Interface Tool*, which allows graphical design, definition and display of valid BLOOM99 database schemas; and a *precompiler for BLOOM99*, which will allow the automatic generation of the DBMS/C++ code that has to be associated to the different user classes to simulate BLOOM99 on an OODBMS as well as on a RDBMS.

Acknowledgements

This work has been partially supported by the Spanish Research Program PRONTIC (TIC99-1078-C02-01), and the Generalitat de Catalunya (1998FI-00228).

References

[AORS99a] A. Abelló, M. Oliva, E. Rodríguez, and F. Saltor. The syntax of BLOOM99 schemas. Technical Report LSI-99-34-R, Dept. LSI (UPC), 1999.

[AORS99b] A. Abelló, M. Oliva, E. Rodríguez, and F. Saltor. The BLOOM Model Revised: An Evolution Proposal. In *ECOOP'99 Workshops Reader (Poster Session)*, pages 376–378, Lisbon (Portugal), June 1999. Springer.

[CKSGS94] M. Castellanos, Th. Kudrass, F. Saltor, and M. García-Solaco. Interdatabase Existence Dependencies: A Metaclass Approach. In *Proc. of the Third International Conference on Parallel and Distributed Information Systems 1994*, pages 213–216. IEEE-CS Press, 1994.

[GSSC95] M. García-Solaco, F. Saltor, and M. Castellanos. Semantic Heterogeneity in Multidatabases. In Bukhres and Elmagarmid, editors, *Object Oriented Multidatabases*. Prentice-Hall, 1995. Invited chapter.

[HM78] Michael Hammer and Dennis McLeod. The semantic data model: A modelling mechanism for data base applications. In Eugene I. Lowenthal and Nell B. Dale, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–36, Austin (TX), June 1978.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In *ECOOP'99 Workshops Reader (LNCS 1743)*, pages 33–56. Springer, Lisbon (Portugal), June 1999.

[Obj99] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.

[Sal96] F. Saltor. Semántica de datos. In *Panorama Informático*, pages 39–64. Federación Española de Sociedades de Informática (FESI), 1996.

[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.

[SL90] A.P. Sheth and J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SM98] Spaccapietra and Maryanski. *Searching for Semantics: Data Mining and Reverse Engineering*. Kluwer, 1998.

[SR99] F. Saltor and E. Rodríguez. On Semantic Issues in Federated Information Systems (Extended abstract). In S. Conrad, W. Hasselbring, and G. Saake, editors, *Proc. of the 2nd Workshop on Engineering Federated Information Systems*, pages 1–4, Khlungsborn, Germany, May 1999. Infix-Verlag.

[SS77] J. Smith and D. Smith. Database abstractions: Aggregation. *Communications of the ACM*, 20(6):405–413, June 1977.