The Development of Semantic Concepts in the BLOOM Model using an Object Metamodel

M. Guadalupe Castellanos
Félix Saltor
Manuel García-Solaco

Report LSI-91-22
INDEX

1. Introduction ................................................................. 1

2. The Object Metamodel
   2.1 Four levels of classification/instantiation ......................... 2
   2.2 Metaclasses ............................................................. 3
   2.3 Models and metamodel ................................................. 3
   2.4 Generalization/specialization ...................................... 5
   2.5 Inheritance: downwards and upwards ................................ 6
   2.6 Instantiation loops ................................................... 6
   2.7 Other dimensions ..................................................... 7

3. An overview of the BLOOM model: abstractions and inherent behaviour
   3.1 Existence dependencies
      3.1.1 Strict existence dependencies .................................. 8
      3.2.1 Relaxed Existence dependencies ................................ 10
   3.2 The Generalization/Specialization dimension
      3.2.1 Disjoint specialization .......................................... 13
      3.2.2 Complementary specialization .................................. 13
      3.2.1 Alternative specialization ...................................... 13
      3.2.1 General specialization .......................................... 14
      3.2.5 Example ............................................................ 14
   3.3 The Aggregation dimension
      3.3.1 Simple aggregation .............................................. 16
      3.3.2 Not simple aggregations
         3.3.2.1 Collection aggregation .................................... 18
         3.3.2.2 Composition aggregation .................................. 19
   3.4 The Classification dimension ....................................... 22

4. Applications of the BLOOM model .................................... 22

5. Related work .................................................................. 23

6. Conclusions and future work ........................................... 24
The Development of Semantic Concepts in the BLOOM Model using an Object Metamodel

M.G. Castellanos, F. Saltor & M. García-Solaco
Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
eamalu@ebrupc51.bitnet, saltor@lsi.upc.es

Abstract

An object metamodel, based on four levels in the classification/instantiation dimension, is presented. The metaclass level is what characterizes models, including their behaviour. Using this metamodel, an object oriented model, BLOOM, is described. This model incorporates a rich set of semantic abstractions including four types of specialization and three kinds of aggregation. The behaviour of these abstractions is analyzed by means of different kinds of existence dependencies.

1. Introduction

The object oriented (O-O) approach has proved successful in programming, and has been extended to other areas. In the case of databases, it has produced O-O DBMSs and O-O data models. It is claimed that O-O databases, or object bases, are better equipped than conventional databases for new, demanding applications, such as CAD, CAE, CIM, CASE or office automation, and for stronger support of traditional business applications [M89]. Furthermore, O-O models are good candidates as canonical models in the context of interoperable databases [CS91].

Several prototypes of O-O DBMSs have been developed, and a number of them are already commercialized in an expanding market [M86], [AH87], [HK89], [B88], [K88]. Each one of them has its own object oriented model, and other models with no implementation exist [M89]. Each model, among other things, has a set of semantic abstractions, that differ in semantic power to model our conceptions of the real world, or Universe of Discourse (UoD).

In this paper, a new O-O model, the BLOOM model is introduced. BLOOM has a richer set of semantic abstractions, including four kinds of specialization, three kinds of aggregation with several subclasses, and different kinds of existence dependencies. BLOOM stands for Barcelona Object Oriented Model. To describe BLOOM, a new object metamodel is used. This metamodel is based on four levels in the classification/instantiation dimension. The level of the metaclasses is what characterizes a model. In our metamodel, metaclasses allow the specification of the behaviour of the instances of their instances.

This paper is organized as follows. Section 2 presents the object metamodel and its features, in particular how metaclasses are used to specify the behaviour of the instances of their instances. In section 3 we describe the abstractions and associated behaviour of the BLOOM model. Section 4 mentions some applications we envision for BLOOM and lastly in section 5 we present our conclusions and reference current and future work.
2. The Object Metamodel

The language used to describe a language is called a metalanguage; the metalanguage and the language being described must be different, otherwise logical paradoxes appear. Analogously, to describe a model a metamodel is needed. In order to describe object oriented models, an object metamodel will be presented. Its usage will be exemplified in the next section, in which the BLOOM model will be described.

2.1 Four levels of classification/instantiation

Our metamodel is based on the classification/instantiation dimension, and has four levels in this dimension: objects, classes, metaclasses and the metametaclas. An object is an instance of one or more classes, a class is an instance of one or more metaclasses, and a metaclass is an instance of the metametaclas. (This dimension should not be confused with the subclass/superclass or generalization/specialization dimension).

This dimension corresponds to the intension/extension dimension of the Reference Model for DBMS Standardization [DAFTG85], based on previous work by Mark & Roussopoulos [MR83]. There are four levels in this dimension in the Reference Model, which are shown in Fig. 2.1, taken from [DAFTG85]. With respect to the other, orthogonal, dimension of the Reference Model, the point-of-view dimension, this paper deals only with conceptual schemas, not with external or internal schemas (unless otherwise noted), because we are concerned with models and metamodels, not with systems.

![Fig. 2.1 Four levels of data description in the Reference Model](image)

In our case, we are dealing with objects. We take the view that everything is an object: classes of objects are objects ("first class citizens"), and so are metaclasses and the metametaclas. In the same way, metaclasses are classes, and the metametaclas is a metaclass. Therefore, the four levels are not disjoint, but overlap; they are not confused into just one, but nested, as shown in Fig. 2.2. We call them levels, even if they appear to be at the same "level". We will use the term individual at any level to mean not in a further level: individual objects are objects which are not classes, individual classes are not metaclasses, individual metaclasses are not the metametaclas.
2.2 Metaclasses

What are metaclasses for? A class of individual objects not only is their storage or collection, it also abstracts all characteristics common to all objects in the class: which attributes and which methods they have, so that they are defined only once, but apply to all instances of the class (type in another terminology).

The same happens with metaclasses; just replace object by class, and class by metaclass in the above sentence: a metaclass of classes not only is their storage or collection, it also abstracts all characteristics common to all classes in the metaclass: which attributes and which methods they have, so that they are defined only once, but apply to all instances of the metaclass.

In particular, this means that the behaviour specified in a metaclass applies to all its classes, and, through them, governs the behaviour of the objects of these classes. As classes abstract the behaviour of its instances, metaclasses abstract the behaviour of the instances of their instances.

The concept and usage of metaclasses is not new. They appear implicitly in Smalltalk, and explicitly in CLOS [BDG88] and ObjVlisp [C87]. In these systems, however, a common behaviour for instances of its class instances cannot be specified. In VODAK [KNS90], metaclasses have been augmented in that direction, and we follow this approach.

Let us use an example to illustrate how metaclasses specify these behaviours. Assume a model that has an abstraction called cartesian aggregation (composition aggregation in BLOOM), and therefore a Cartesian Aggregation metaclass. Assume an object base has an Assignment class as a cartesian aggregation of the Employee class and the Project class. This cartesian aggregation is an instance of the metaclass of all such aggregations, the Cartesian Aggregation metaclass. To create a new assignment, the corresponding employee and project must exist. This is not defined at the object level, nor at the class level, because it is common to all cartesian aggregations. It is in the definition of the Cartesian Aggregation metaclass where it is specified, once and for all, that a new aggregate cannot be created unless its constituents exist. The Assignment class, as all instances of the Cartesian Aggregation metaclass, will implicitly enforce this behaviour. Further examples will be seen when describing BLOOM in Section 3.

2.3 Models and metamodel

Metaclasses are precisely what constitute a model. Different models differ in which metaclasses they have, which aggregation metaclasses, which generalization/specialization metaclasses, in whether or not new metaclasses may be created, and so on.

For a given object base, the state it has at a given point in time is characterized by the objects it has. As time goes by, objects are created and destroyed: its state changes. But, as long as the set of classes of the object base remains constant, all those states have something in common: the classes (the schema in another terminology). Classes, in turn, may be created and destroyed
Fig. 2.2 Levels of the Object Metamodel

Fig. 2.3 The Object Metamodel
(schema evolution), but something remains stable: the model, the metaclasses. And this model, these metaclasses, are common to other object bases as well: to all bases modeled after these metaclasses, to all bases defined following the model. Each model has a fixed set of metaclasses; if the model is extensible, it has the capacity to create new metaclasses - but not to destroy those fixed. Moreover, a model will have a fixed set of individual classes and of individual objects (constants). We say that the fixed set of all objects (metaclasses, classes and individual objects) is inherent to the model; they are its inherent objects, sometimes called primitive. They appear in every object base modeled after it. In contrast, all objects not inherent to a model, but particular to a specific object base, are called defined.

A model is common to all bases modeled after it; in the same way, a metamodel is common to all models definable by it. There is not just one O-O model, there are many, and this situation is not likely to change [M89]. These O-O models form a family, that [A89] tries to characterize. Our metamodel tries to be able to describe a large part of this object family. All models definable by our metamodel have something in common: a set of objects which we call intrinsic to the metamodel. They include the class of All Objects, the metaclass of All Classes, and the metametaclass of All Abstractions. This is represented in Fig 2.3, which is an elaboration of Fig 2.2.

2.4 Generalization/specialization

Generalization [SS77] is the abstraction by means of which differences among similar classes are ignored to form a (higher order) class in which only similarities are retained. Stating it in another way, by extracting common properties (static and dynamic) from similar classes, a more general one, the generic class, is defined. The similar classes become specializations of the generic one. The generic class is simply called superclass, while the specialized classes are called subclasses. A given class may be generalized into more than one superclass.

Generalization/specialization is intrinsic in our metamodel. All classes are subclasses of the class of All Objects. In the same way, all metaclasses are subclasses of the metaclass of All Classes. This means that there is an intrinsic Specialization metaclass.

The transitivity of generalization/specialization is also intrinsic in our metamodel. In particular, the (metameta)class of All Abstractions is transitively a subclass of the class of All Objects. Consider Fig. 2.3, which has been completed by a simple example to show how the metamodel is used to describe a model (metaclasses), how the model so described is used to define an object base (defined classes), and how this base is populated with individual objects: class C1, being a subclass of class C, is transitively a subclass of the class of All Objects (in fact, it is so also inherently in this case).

These transitive generalizations are not explicitly specified in our metamodel, and are not shown in Fig. 2.3, because we are at the conceptual point of view, as we said before. At the internal point of view, implementors are free to put in place the corresponding connexions, if they so choose. At the external point of view, an external schema may include any of the
generalizations, direct or transitive, and a user, when displaying the generalization and/or specialization of a class, may select between only those that are direct, those of length 2, those of length less or equal to n, or the whole transitive closure. The power to derive external schemas is then greater than in the metamodel and models analyzed in [S86].

All classes are interrelated in the form of a semi-lattice. This is so because any two classes have some superclass in common: at least the class of All Objects, which is the top of the semi-lattice. This is not a complete lattice, because any two classes have not, in general, a subclass in common, in our metamodel. A specific model may choose to adopt such a lattice, by having the class of No Objects, which would be a subclass of any class, the bottom of the lattice; this class, inherently empty, would be an object inherent to the model, not intrinsic to the metamodel as the top class is. This allows our metamodel to describe models with different forms of generalization/specialization: trees (sometimes called inheritance hierarchies), semi-lattices, and lattices.

2.5 Inheritance: downwards and upwards

A class inherits all the structure and behaviour, i.e. attributes and methods, of all its (direct and transitive) superclasses: downward inheritance. Since all classes are subclasses of the class of All Objects, the attributes and methods of this class are inherited by all classes. For example, all classes have the method new by inheritance from the class of All Objects; this is an intrinsic method.

A class with several direct superclasses may have name conflicts when inheriting from them (multiple inheritance). Our metamodel allows the description of models supporting multiple inheritance, but does not require this support.

Classes may inherit attributes and methods from their subclasses; this is called upward inheritance [SN88]. This is a requirement for interoperable databases, as explained in Section 4. Our metamodel allows upward inheritance, but does not force it.

2.6 Instantiation loops

Combining specialization and instantiation, it may be seen that an instance of a class is an instance of its superclasses. For example, object Ob in Fig. 2.3, being an instance of class C1, is also an instance of C. Classes C and C1 are instances of metaclass M, and consequently instances of the metaclass of All Classes. In particular, all objects are instances of some class, and all classes are subclasses of the top class of All Objects, therefore all objects are instances of this class, as it should be. All these not direct instantiations are not explicitly specified in our metamodel, and are not shown on Fig. 2.3. Again, we are at the conceptual point of view.

All objects are thence interrelated along the instantiation dimension: do they form another semi-lattice? No, because there are loops. The class of All Objects is an instance of the (meta)class
of All Classes, which is a subclass of the class of All Objects; therefore, this class is an instance of itself. In the same way, the (meta)class of All Classes, being an instance of the (metameta)class of All Abstractions, which is a subclass of it, is an instance of itself. The (metameta)class of All Abstractions is an instance of itself, by a direct loop shown in Fig. 2.3. In fact, the class of All Objects is not only an instance of itself, it is also an instance of an instance of itself, and an instance of an instance of an instance of itself!

These loops are expressable at the conceptual point of view in which we are, but they are not acceptable at the internal point of view. In an implementation, the representation of all intrinsic objects is built into the system, and loops are avoided. In practice, the level of the metametaclass, and intrinsic metaclasses such as the class of All Classes, would not be implemented as such; Fig. 2.3 suggests this by using dashed lines and dashed circles.

When defining a specific model, instantiation loops may appear in its metaclasses. This would mean that a system supporting this model would have these metaclasses implemented by some built in feature, and would therefore be bound to that model.

In conclusion, instantiation loops are allowable in (intrinsic and) inherent objects; they are forbidden in defined objects.

2.7 Other dimensions

The aggregation dimension is not intrinsic in our metamodel. This allows it to describe models with different forms of aggregation, as will be seen for BLOOM in the next section. The metamodel is also open to other kinds of abstractions.

3. An Overview of the BLOOM model: abstractions and inherent behaviour

BLOOM is an object oriented data model where object classes are defined according to the abstract data type (ADT) concept. An ADT encapsulates a data structure and its associated operations whose effects are public, while their implementation is private. Therefore, in BLOOM we have the corresponding structural and operational funcionality specifications.

In BLOOM, the description of the structure entails the description of the properties and the relationships to other object classes. The class abstraction is the primary conceptual modeling vehicle. Relationships are embodied in the interclass connections that are specified as part of the class definition. The intent is to permit the database designer to express as much as possible of the semantics of the object class in a clear way with mechanisms that map directly onto the designer's concepts, while at the same time permit flexible and multiple views of the data keeping with the semantic modeling philosophy. With these objectives in mind we have designed a model that offers a full set of modeling facilities instead of providing a few primitives from which the designer can construct more complex conceptual objects like in other approaches.
Modeling with BLOOM permits to capture a rich set of semantic relationships among real world entities. Relationships can be characterized by the abstractions they are capable of representing and the means by which they do so. In BLOOM we have the following ones:

- Classification/Instantiation
- Four kinds of Specialization: complementary, disjoint, alternative and general
- Three kinds of aggregation: simple, collection and composition

The operational functionality models the dynamic properties which can range from the simple specification of insertion and deletion constraints to the modeling of operations. For each operation, called method, there is a public message selector together with formal arguments, this is called the method's signature. Signatures are visible while their implementation is private. The execution of a method is invoked by sending a message to the object using the signature with actual values for the arguments.

The insertion and deletion constraints used to maintain the integrity of the semantic database form one of the most important features of the model. The specification of these constraints constitutes the behavioural interpretation of the semantics of the model. If objects are interrelated, then the insertion, deletion, or modification of one object will impact the existence status of other objects connected to it through relationships. This means that according to the specific kind of relationship there will be existence dependencies between characterized by the side effects and constraints of insert and delete operations called *behavioural constraints*. The behavioural semantics of the different abstractions is given by these constraints. BLOOM incorporates this idea by means of the concept of meta-classes (see section 2). Then, individual objects being instances of instances of these meta-classes behave accordingly.

As the behavioural constraints are a consequence of the existence dependencies between the concepts related, in the next subsection we will give a characterization of the different kinds of existence dependency found in real world and incorporated in BLOOM's abstractions.

### 3.1 Existence Dependencies

#### 3.1.3 Strict Existence Dependencies

This dependency, as its name suggests, implies that if an object is *strict existence dependent* on another, then it cannot exist if the other doesn't exist; this means that its existence depends strictly on the existence of the other one. We will call them *dependent* and *dependor* respectively. The next figure clearly shows this concept by representing the lifetimes of the involved objects:

```
                      ---------→ time
                      |                      |
                      |                      |
                      |                         dependent
                      |--------------------------|
                      |                          |                      |                         dependor
```
This corresponds to two different real life situations. One where the object cannot exist in the reality independently of the other one. The other situation appears when the object continues existing in real life even if the dependor dies, but then it is not of interest anymore in our UoD.

The insertion constraint linked to this concept makes sure that the dependent object cannot be created if the object upon which it depends has not been created before. With respect to the deletion constraint, this can have two different implications: in one case the deletion of the dependor object entails the deletion of the dependent object, this corresponds to the Propagate_Strict_Dependency case, and in the other case called Block_Strict_Dependency the deletion of the dependor simply cannot be made if there are dependent objects on it. The cascade and block effects are used in [M90] for the rules associated to referential integrity constraints.

All these ideas apply to the case where the dependent object depends exclusively on one instance of the dependor object class, so the previous cases turn out to be renamed to Exclusive_Propagate_Strict_Dependency and Exclusive_Block_Strict_Dependency respectively. But it can be the case that the dependent can be shared by several instances of the dependor class, leading to other four kinds of dependencies. In two of them the dependent existence relies on that of at least one dependor instance. This implies that for its insertion it will be enough that one such instance exists, and that the deletion of a dependor instance will be either blocked or propagated to the dependent only if it is the last dependor instance. These are the Shared_Block_Strict_Dependency and the Shared_Propagate_Strict_Dependency respectively. In the other two cases the existence of the dependent relies on that of all of the dependor instances and the deletion of any of them will either block or entail the deletion of the dependent. These correspond to the All_Block_Strict_Dependency and the All_Propagate_Strict_Dependency respectively. [KBG89] makes use of the exclusive and shared concepts to classify composite (part-of) references.

To make these ideas clear, let's rephrase them in a rule style way. Let O be the object class that depends on object class O', then the insertion rule (IR) and the deletion rule (DR) corresponding to the previous cases are the following (referring only to the instances of O and O' involved):

- **Exclusive_Block_Strict_Dependency** [Exc_Blo_Str]

  IR: insert (O) iff exists (O')
  DR: delete (O') iff not exists (O)

- **Exclusive_Propagate_Strict_Dependency** [Exc_Pro_Str]

  IR: insert (O) iff exists (O')
  DR: delete (O') \rightarrow delete (O)

- **Shared_Block_Strict_Dependency** [Shr_Blo_Str]

  IR: insert (O) iff exists al least one (O')
  DR: delete(O') iff not last (O') or not exists (O)

- **Shared_Propagate_Strict_Dependency** [Shr_Pro_Str]

  IR: insert (O) iff exists at least one (O')
  DR: delete (O') \& last (O') \rightarrow delete (O)
Notice that for the shared type cases, if O' is not the last one, then its deletion doesn't have implications.

* **All_Block_Strict_Dependency**
  
  IR: insert (O) iff exists every (O')
  DR: delete any (O') iff not exists (O)

* **All_Propagate_Strict_Dependency**
  
  IR: insert (O) iff exists every (O')
  DR: delete any (O') → delete (O)

### 3.1.2 Relaxed Existence Dependencies

As its name suggests, it is a relaxed version of the strict dependency. The idea is that the dependent object cannot be created if the object on which it depends doesn't exist, but once it is created, its existence turns out to be independent of it. This means that if the depender object is deleted, the dependent one continues existing. In practice this corresponds to the case where the dependent object entered the UoD because of its relationship to the depender one, but from that moment on it becomes interesting by itself independently of the other. Like in the previous case, we will depict this concept with the following life time graph

```
    time
    dependent
    depender
```

Again we can have different cases for this kind of dependency, but its implications are only with respect to insertion, because for deletion, as the graphic shows, the depender can be deleted without impact on the dependent. Thus, we only have three cases, the **Exclusive_Relaxed_Dependency**, where the dependent object requires the existence of the depender one to be created, the **Shared_Relaxed_Dependency** requiring that at least one of the depender instances exists and the **All_Relaxed_Dependency** that requires the existence of all the depender instances.

The following rule style rephrasing summarizes their effects. As previously, O represents the dependent object, and O' the depender one. IR stands for insertion rule, but there are not deletion rules:

* **Exclusive_Relaxed_Dependency** [Exc_Relax]

  IR: insert (O) iff exists (O')

* **Shared_Relaxed_Dependency** [Shr_Relax]

  IR: insert (O) iff exists at least one (O')

* **All_Relaxed_Dependency** [All_Relax]

  IR: insert (O) iff exists every (O')

Figure 3.1 depicts the generalization hierarchy of the existence dependencies in BLOOM.
Fig. 3.1 The existence dependencies
Now that we have explained the different kinds of dependency, we can link these concepts to the abstractions of BLOOM in order to characterize their behavioural semantics. So, in the rest of this section we give a short description of each abstraction, followed by its behavioural functionality, and an example for each case.

3.2 The Generalization/Specialization Dimension

All semantic models capture the notion of generalization/specialization described in section 2, but in BLOOM we go one step further by distinguishing different types of specialization in order to be able to express yet more semantics. These are:

- Disjoint Specialization
- Complementary Specialization
- Alternative Specialization
- General Specialization

As previously mentioned, the operational interpretation of every abstraction is given by its insertion and deletion behaviour, that is, the side effects of insertion and deletion on the related objects, as well as the constraints imposed on/to them. In object oriented models we must distinguish two kinds of insertion: one corresponding to the 'new' operation where we create a new object and insert it in a class, and the other, the 'insert' operation, corresponding to the insertion in a class of a previously created object. For the generalization/specialization abstraction these operations have a slightly different implication. The 'new' operation on a subclass implies an insert on the superclass as well. In contrast, the 'insert' operation implies an insert on the superclass only if it didn’t exist there previously.

Rephrasing this idea in our usual rule like informal style and assuming that O represents the subclass, O’ its superclass, and Os where s=1..n its sibling classes, we have the following insert rule (IR)

IR: new (O) → insert (O’)  
IR: insert (O) and not exists in (O’) → insert (O’)

For deletion, we also distinguish two kinds, one called 'kill' that makes the object disappear definitively from the object base, propagating its deletion to its super and subclasses. The other one called 'delete', only removes the object from the class indicated. The delete rules are

DR: kill (O) → delete from immediate subclasses and
    delete_up from immediate superclasses
DR: delete_up (O) → delete up from immediate superclasses
DR: remove (O) → delete from immediate subclasses

Recall that we are just describing the constraints and the side effects and that the rules apply recursively. There are other rules but they are specific to the type of specialization.
3.2.1 Disjoint Specialization

The specialization subclasses are disjoint, that is, each object of the superclass belongs at most to one subclass. The only implication that this restriction has is that for the 'insert' we must make sure that the object being inserted doesn’t exist in another sibling subclass. The insert rule associated to this is the next one:

IR: insert (O) or new (O) iff not exists in any O

3.2.2 Complementary Specialization

In this case the subclasses are complementary, in the sense that the union of the objects of the subclasses constitutes the whole population of the superclass but there can be some overlappings between the populations of the subclasses. The principle that summarizes this idea is that each object of the superclass belongs at least to one subclass. Then, when we make 'insert' or 'new' in the superclass, this insertion must be propagated to at least one subclass (system prompts for it). In contrast when there is a delete in a subclass and the object belongs to only one of the complementary subclasses, then either the deletion is blocked or else it is propagated to the superclass. This leads to two kinds of complementary specialization: the **blocks_complementary_specialization** and the **propagates_complementary_specialization**. The insertion rule is the same for both but the deletion rule differs in each case

IR: insert (O') or new(O') -> insert in at least one subclass

**Blocks_complementary_specialization** [Blo_comp]

DR: delete (O) iff exists in at least another O

**Propagates_Complementary_Specialization** [Pro_comp]

DR: delete (O) and not exists in any O -> delete_up (O')

3.2.3 Alternative Specialization

This is the conjunction of the previous two kinds of specialization. This means that the subclasses are complementary (in the same sense as before) but not overlapping. The principle behind this idea is that each object belongs to one and only one subclass. The rules are the following ones

IR: insert (O') or new(O') -> insert in one subclass

IR: insert (O) or new (O) iff not exists in any O

Like before, the deletion of a subclass object can be blocked or propagated giving rise to two kinds of alternative specialization

**Blocks_Alternative_Specialization** [Blo_alt]

DR: delete (O) always rejected

**Propagates_Alternative_Specialization** [Pro_alt]

DR: delete (O) -> delete_up (O')
3.2.4 General Specialization

This one has no restrictions at all so it doesn't present further implications.

3.2.5 Example

We have given the associated behaviour corresponding to the different types of the generalization/specialization abstraction. As this constitutes the operational interpretation of the abstractions and as such is an inherent part of them, BLOOM embodies this functional behaviour in the following corresponding metaclasses:

- General Specialization
- Disjoint Specialization
- Block Complementary Specialization
- Propagate Complementary Specialization
- Block Alternative Specialization
- Propagate Alternative Specialization

The metaclasses have insert and delete methods and their private implementation must make sure that the corresponding effect, side effects and constraints, above specified informally by means of rule like expressions, hold for each one of them.

The designing tool associated with BLOOM, will guide the user in the design. The designer is prompted by the system which relies on a rich set of natural common concepts that map directly onto the designer's concepts in order to identify the type of relationship. Once the system identifies it, the effect is to make the described class, an instance of the metaclass corresponding to the abstraction represented by the relationship. As such, the methods of the metaclass are instantiated by this particular instance. There can be different display formats, and we adopt only one of them for the examples. Notice that the system can give us both, the direct or the inverse construct corresponding to the relationship, accordingly to the class of interest. The criterion used for each specialization is shown in brackets.

```
Class persons
    Generalization_of
        General : [occupation] Student, Employee
        Propagate_Alternative : [gender] Male, Female
        ... {other abstractions}
End_Class

Class Employee
    Specialization_of
        General : persons
            Generalization_of
                Disjoint : [type_of_contract] Salesman, Manager
                ...
end_class
```
Fig. 3.2 The Generalization Dimension
The user could also select the option of 'operational interpretation' and the system would additionally display the implicit instantiated rules associated to the insert and delete methods of the corresponding metaclasses, so that the user can be aware of the operational implications of the relationships of the object of interest.

The figure 3.2 depicts the generalization dimension of the previous example.

3.3 The Aggregation Dimension

Aggregation [SS77] in general, is the abstraction by which several objects are aggregated into a new, complex object. In BLOOM we distinguish three types of aggregation according to the different semantics found in the real world

- Simple aggregation
- Collection aggregation
- Composition aggregation

3.3.1 Simple Aggregation

The simplest type, called *simple aggregation*, is employed to express that the attributes involved in it are just properties of the object being described. The object is not generally perceived as being the result of the aggregation of these attributes, which constitute merely the object properties that are of interest in our UoD, but it may be considered the aggregation of its attributes and this justifies its name. In this sense, aggregation provides a means for specifying the attributes of an object type. This is called 'reference' by some authors and 'weak reference' in [KBG89].

In object oriented models, each attribute takes as domain a class which can be either a primitive class such as integer, string and boolean, or a user defined class with its own attributes itself. We distinguish this by the concepts *Primitive* and *Defined* respectively. Furthermore, attributes can be single valued or multi valued, so we have decided to assign single valued by default and explicate multivalued attributes by the *Set of* construct. Besides, attributes can be *Obligatory*, in case they don't accept nulls; by default they accept nulls.

With respect to dependencies, as there can be any type of dependency between the attribute and the object described, the kind of dependency must be made explicit. For this, we use the constructs corresponding to the different types of existence dependency explained in section 3.1. The corresponding behaviour is represented by the insertion and deletion rules given there. The following example illustrates this idea.
Class employee
   Simple_aggregation_of
       Primitive
           name: string [30]
           salary: integer [0,500]
           phone: set_of string [7]
   Defined
       Exclusive_Block_Strict_depends_on dept: departments obligatory
       Shared_Relaxed_dependent children: set_of person
       Shared_Propagate_Strict_dependent own_car: set_of vehicle
           company_car: vehicle

end_class

Class departments
   Simple_aggregates_in
       Exclusive_Block_Strict_dependent employee

end_class

Class person
   Simple_aggregates_in
       Shared_Relaxed_depends_on employee

end_class

Class vehicle
   Simple_aggregates_in
       Shared_Propagate_Strict_depends_on employee as own_car
           employee as co_car

end_class

With respect to dependencies, the only extension applied here is the explicit distinction between the dependent and depender roles. We make this explicit by appending to the construct that indicates the kind of dependency (Exclusive_block_strict, and so on), the words depends_on and dependent respectively. When nothing about dependency is indicated in the attribute specification (slot), then by default there is an existence independence between the aggregate object and the attribute one. The only effect that the deletion of the referenced object has is that of nullification, this means that the reference to this object must also disappear.

Now let's explain our example. An employee has a name, a salary and one or more phone numbers. He/she must belong to one department (obligatory), and the existence of the department
is required for him/her to exist in the database (exclusive_block_strict dependency). If it could be the case that there are employees which are not assigned to any department, then, the attribute department wouldn't be obligatory but the dependency would still hold. For our application their private cars have interest only as long as the employee also has, but it can be the case that the car is shared by two or more employees, so the car will exist in the database if at least one of its owners exists in it, otherwise the deletion of the last owner will entail the deletion of the car (shared_propagate_strict dependency). In contrast, the existence of the company car assigned to an employee optionally, is independent of the existence of the employee to which it is assigned. With respect to the children of an employee, they become interesting when the employee does, but if this one quits or gets fired, his/her children continue being interesting by themselves anyway (shared_relaxed dependency). Note that the attributes children and cars correspond to the weak entity of the E-R model [Ch76].

3.3.2 Not Simple Aggregations

In contrast to the simple aggregation by which we merely specify properties of the object, there are two other kinds whose common core semantics is that the aggregate object is perceived as being formed by aggregating its component objects. This means that it is the aggregation of these components objects that gives rise to a new object, so they are not merely simple properties of it, their existence is necessary for the object to exist.

Recall that our purpose is to have a rich set of abstractions that maps directly onto the designer's concepts found in the real world. For example, an employee is the simple aggregation of its name, salary, phone numbers, children and cars, but these represent just the properties of employees of interest for us, it is not the aggregation of these objects that gives rise to an employee in the reality. In contrast, it is the aggregation of the members of a collection of ships that gives rise to a convoy, so this is not represented by a simple aggregation but by another one. We need other kinds of aggregation that extend the semantics of the abstraction concept.

3.3.2.1 Collection Aggregation

In the case of the collection aggregation (or just collection), the added semantics is that the new objects are composed of collections of a given class. This abstraction permits to treat each collection of objects as a single one. This concept, more commonly known by cover aggregation, was introduced by Hammer and McLeod who simply called it aggregate in (HMc78). For the new object, we use in turn the simple aggregation concept to specify its properties.

The behaviour of this abstraction is given by the shared_strict dependency because the existence of the aggregate object strictly depends on the existence of the collection of objects that constitute it. This collection can range from only one to any number of members, and at least one member must be in the collection for the aggregate object to exist. In BLOOM we distinguish two kinds of collection aggregation depending on the deletion behaviour. In one of them, the deletion of one object of the underlying class is blocked if this object is the only member of the collection that
constitutes the aggregate object, this is the \textit{shared\_blocks\_strict} dependency, so the corresponding aggregation is named \textit{Blocks Collection Aggregation}. The other one is the \textit{Propagate Collection Aggregation} that relies on the \textit{shared propagate strict} dependency, because the deletion of the last member of the collection entails the deletion of the aggregate object. We have omitted the terms \textit{shared} and \textit{strict} because they are inherent to the collection aggregation semantics. The next example uses this concepts

\begin{verbatim}
Class  club
    Propagate\_Collection\_Aggregation\_of  member:person
    Simple\_aggregation\_of  \{this abstraction previously explained\}
    
    Primitive
        name: string [10]
        founded: date
    
    Defined
        president: famous\_people \textit{obligatory}
    ...

end\_class

Class  person
    Propagate\_Collection\_Aggregates\_in  club
    ...

end\_class
\end{verbatim}

A club is a collection of people, it exists as long as there are members that belong to it, when the last one quits, the club will not exist anymore (\textit{propagate}). As any object, the club has its own properties described by means of the simple aggregation abstraction. A club must always have a president (\textit{obligatory}).

3.3.2.2 Composition Aggregation

In the \textit{composition aggregation} (or just \textit{composition}), there is a fixed set of component classes, and an aggregate object is formed by aggregating one (or more) object from each of these classes. This is called 'aggregation' by some authors, and 'cartesian aggregation' in [C79]. The component objects are not simply properties of the aggregate but it is their aggregation that gives rise to it. The aggregate object, in turn, has its own properties specified by the simple aggregation abstraction explained before. This kind of aggregation differs from the previous one, in that there, it was a collection of objects of the same class that was treated as one single object,

The behaviour of this abstraction is given by the composition of the different kinds of \textit{Strict} dependencies linked to each relationship between the component and the aggregate object. Two different situations can arise. One where the aggregate object cannot exist if any of its components objects don't exist, as in the following example:
Class command
  Composition_aggregation_of
    Exc_Block_Strict_dep_on supplier
    All_Block_Strict_dep_on set_of product
  Simple_aggregation_of
    Primitive
      cmd_date: date
    Defined
      ship_to_address: address
...
end_class

Class supplier
  Composition_aggregates
    Exc_Block_Strict_dependent commands
  ...
end_class

In this example, a command can only be placed if the supplier and all the products involved exist. Neither suppliers nor products can be removed if there are commands placed on them.

The other situation is similar but here it is possible to delete a component object as long as it is not the last object involved in the aggregation with respect to a dependor class. The following example corresponds to this situation:

Class project_assignments
  Composition_aggregation_of
    Exc_Propagate_Strict_dep_on project
    Shared_Block_Strict_dep_on set_of employee
  ...
end_class

Class employee
  Composition_aggregates
    Shared_Block_Strict_dependent project_assignments
  ...
end_class

Class project
  Composition_aggregates
    Exc_Propagate_Strict_dependent project_assignments
  ...
end_class
Fig. 3.3 The Aggregation Dimension
An instance of project assignments is composed of a project and one or more employees assigned to it. The assignment existence depends strictly on the existence of the project (exclusive) and on the existence of at least one employee (shared). If the project is deleted it is propagated to its assignments (propagate). An employee can be removed from a project as long as it is not the only employee assigned to it, if so then the operation is rejected (block).

Figure 3.3 depicts the aggregation dimension including some examples given for each kind of abstraction.

3.4 The Classification Dimension

According to our metamodel, presented in Section 2, there are four levels in the classification/instantiation dimension, and what characterizes a model is the level of the metaclasses. Therefore, we concentrate now at this level, and review its set of metaclasses.

To support generalization/specialization, BLOOM has the Specialization metaclass, and its subclasses: Disjoint metaclass, Complementary metaclass, Alternative metaclass and General metaclass, that specify behaviours as explained in 3.2. The Specialization metaclass is intrinsic in the metamodel, its subclasses are inherent to the model. This specialization of the Specialization metaclass into its four subclasses is alternative, and therefore an instance of the Alternative metaclass. This produces an instantiation loop, which is inherent to the model.

To support aggregation, BLOOM has the Aggregation metaclass, and its subclasses: Collection metaclass, Composition metaclass and Simple Aggregation metaclass, with their corresponding specifications of behaviours, as presented in Section 3.3. There are subclasses of these subclasses, in turn (Blocks or Propagate, Exclusive, Shared or All). All these metaclasses are inherent to the model.

We have said in Section 2 that metaclasses in our metamodel allow to specify the behaviour of the instances of their instances. We have shown, through the description of BLOOM, many examples of this feature.

4. Applications of the BLOOM model

The BLOOM model has been conceived in the context of interoperable databases. In this context, the conceptual schemas of the different databases that must interoperate, present syntactic and semantic heterogeneities. Syntactic heterogeneity derives from the use of different data models. Semantic heterogeneity is a consequence of the different perceptions that the designers may have about a same reality.

The solution commonly adopted to solve syntactic heterogeneity is the use of a common model that permits the description of the divergent local schemas using a single representation
formalism; this common model is called *canonical* model. But even if the schemas use a common model, there is a need to perform a semantic homogeneization which involves two different tasks: first, to detect semantic conflicts (a conflict occurs when a same concept has different representations) between the databases, and then to solve them. As stated in [CS91], the main obstacle for the detection of semantic conflicts is the lack of semantics in the local schemas, consequence of the poor expressiveness of traditional models. So the canonical model must be rich enough to capture the semantics already expressed by traditional data models but also to express additional semantics that is missing in the local schemas. Capturing additional semantics during the schema translation is a big help for the tasks of detection and solution of semantic conflicts. *Semantic enrichment* is the process that captures this additional semantics by making explicit the implicit structure and associating additional behaviour which is hidden in local application programs or even worse in informal local conventions. BLOOM has been designed as such a canonical model.

In [CS91] we have described a process of semantic enrichment based on inclusion dependencies. There, we can contrast the poor expressiveness of relational models against the richness of BLOOM representations. The process also detects and solves some design anomalies.

In [GS91] the BLOOM model is extended with discriminated operations to help in the task of schema integration.

We envision a variety of more applications for BLOOM, among them we can mention the most obvious ones: as the basis of tools for designing data (object) bases and as the object model upon which to implement an object oriented database system.

5. Related Work

As already mentioned, the classification/instantiation dimension of our metamodel corresponds to the intension/extension dimension of [DAFTG85], but our two upper levels have a different function. Metaclasses are found in several systems, such as CLOS [BDG88] and ObjVlisp [C87], but not to specify the behaviour of instances of their instances, as in our case. The four levels in our metamodel roughly correspond to the four levels of a VODAK class system [KNS90]; in our case, however, metaclasses play a different role, and the metaclasses of BLOOM are quite different from those of VODAK.

Previous work on metamodels is generally not based on O-O models, but on semantic models and variations of the ER family [S86].

- Many object oriented models have been defined for specific systems. They include Gemstone [M86], Vbase [AH87], CACTIS [HK89], O2 [B88] and Orion [K88]. BLOOM differs from these models in its richer set of abstractions, such as several kinds of specializations, aggregations and dependencies.
6. Conclusions and future work

A model captures the semantics of the UoD by means of the abstractions it supports in the generalization/specialization and aggregation dimensions, and how these abstractions are structured. Because BLOOM is based on a rich set of semantic concepts, which distinguish different kinds of dependencies, specializations and aggregations, it can capture more semantics than most existing models. This allows the definition of object bases with class structures which are really expressive as shown in the examples given along section 3. The nature of the relationships between a class and all the classes to which it is related is given in great detail, so as to reflect as much semantics as possible, including the impact that an object of the class has on their related objects. This behaviour of the individual objects is specified in the metaclasses, and apply to the instances of their instances through the metaclass level of the classification/instantiation dimension of the metamodel.

As stated in section 4, in the context of interoperability we need a canonical model rich enough to capture the semantics already expressed in the local schemas and the one captured by the enrichment process. BLOOM has been conceived with this purpose in mind. Simultaneously, we have developed some algorithms for converting relational schemas to BLOOM schemas based on inclusion dependencies [CS91] as part of the enrichment process. We are now working along two lines, one on refining BLOOM and defining its operations according to the requirements of enrichment and integration, and the other on research for building intelligent tools for the interoperability context. There will be different tools for the different stages of the integration process. The aim of one of those tools is to infer inclusion dependencies so that relational schemas of preexisting databases can be mapped to BLOOM. Then another intelligent tool will be in charge of detecting and solving semantic conflicts using the enriched local schemas expressed in BLOOM.

Acknowledgments

This research has been partially supported by the PRONTIC project TIC89/0303.

References


