Multiobjects to Ease Schema Evolution in an OODBMS

Lina Al-Jadir, Michel Léonard

Centre Universitaire d'Informatique (C.U.I.), Université de Genève
24 rue Général-Dufour, 1211 Genève 4, Switzerland
{aljadir, leonard}@cui.unige.ch

Abstract. The multiobject mechanism is a pertinent way to implement specialization in an object database and differs from the classical mechanism used in most object-oriented database systems. It supports multiple instantiation, automatic classification and object migration. Consequently it is well suited to take into account schema evolution. It makes schema changes more pertinent, easier to implement, and less expensive than with the classical implementation of specialization indeed. The multiobject mechanism is implemented in the F2 database system which supports schema evolution.

1 Introduction

In the classical implementation of specialization in object-oriented database systems (OODBMS) an object is an instance of one most specific class. It is completely stored in this class, i.e. all attribute values on local and inherited attributes are present in the object. This has several shortcomings for object modelling and object evolution. Since an object is an instance of only one class, one must use multiple inheritance to model real-world entities that have many facets at once. This can lead to a combinatorial explosion of sparsely populated classes, as pointed out in [34] [33] [27] [28]. Once an object is created in a class, it stays in that class until it is deleted from it. This is a serious limitation, since one is forced to model real-world entities that evolve dynamically with objects that can not, as pointed out in [30] [33] [2] [28]. Several approaches (see section 4) have been proposed to overcome these shortcomings. In this paper we address other shortcomings of the classical implementation of specialization which are related to schema evolution.

Schema evolution is an essential feature of a database system to allow database applications to run in a dynamic environment. Updating the schema of a populated database has repercussions on database objects in order to keep the database in a consistent state. Supporting single-instantiated and static objects restricts schema changes. Moreover, it requires to copy data when modifying the schema which is extremely time-consuming. Suppose for example that a class *Swiss* is added as a subclass of *Person*. With a classical implementation, it is not possible to make some of the existing *Person* objects belong now to the *Swiss* subclass. For all swiss citizens, the database administrator has to create copies of the *Person* objects in the new subclass *Swiss* and to delete the

corresponding objects in class *Person*. This may be a very expensive operation if there are many swiss persons.
Supporting classical inheritance makes many schema changes difficult to understand and to implement. *Orion* [8] requires complex propagation rules to disambiguate the effect of schema changes. *Gemstone* [29] does not support some schema changes because their motivation is to only provide schema changes which are “well understood and have a reasonable implementation”.

We propose the multiobject mechanism to implement specialization in an object-oriented database system. This mechanism supports multiple instantiation, automatic classification and object migration. It makes schema changes more pertinent, eases their implementation and understanding, and reduces their execution time. The multiobject mechanism is implemented in the F2 OODBMS. F2 is a general purpose database system developed at C.U.I. and used to experiment several features such as: updatable views [16], information system design methods [15], knowledge databases [17], database integration [13], schema evolution [7] [6] [3]. It is written in Ada and runs under SunOS, DEC/ALPHA, MacOS and Windows 95.

The remainder of the paper is organized as follows. In section 2, we present the multiobject mechanism. In section 3, we show its advantages with respect to schema evolution. In section 4, we describe related approaches and compare the multiobject mechanism to them. In section 5 we conclude with a summary.

2 Multiobject Mechanism

We describe in this section the multiobject mechanism. We introduce first the multiobjects and describe then the methods to manipulate them.

2.1 Multiobjects

Defining a Multiobject. In the F2 model [6] an *object* is an instance of a *class*. Objects structure is defined by class attributes. Objects behaviour is defined by primitive methods and triggered methods. A class, called *subclass*, can be declared as a specialization of another class called *superclass*. The class hierarchy is a forest, i.e. a set of *specialization trees*: a subclass has only one superclass (single inheritance), and there is not a root system-defined class. On a subclass may be defined *specialization constraints*. An object belongs to a subclass if and only if it satisfies the specialization constraints of the subclass. The *ancestors* of a subclass are its direct and indirect superclasses. The *descendants* of a class are its direct and indirect subclasses.

We assume that the reality consists of *entities*. Entities have several facets. For example, a human being may be seen as a person, an employee, a tennis player, a student, etc. An entity is implemented in the multiobject mechanism by a set of objects in distinct classes of a specialization tree, $M_o = \{o_{C1}, o_{C2}, ..., o_{Cn}\}$, called *multiobject*. Each object o_{Ci} denotes a facet of the entity and carries data specific to its corresponding class C_i. This is referred to as multiple instantiation. A multiobject M_o satisfies the following constraint: if o_{Ci}, $1 \leq i \leq n$, belongs to M_o and C_i is a subclass of C_j then there must be an object o_{Cj}, $1 \leq j \leq n$ and $j \neq i$, which belongs to M_o. In other words, if an entity possesses an object in class *C*, then the entity must also possess objects for all the ancestors of *C*. For example, the class *Student* is a subclass of *Person*. A student is implemented by a multiobject containing two objects $r_{Student}$ in *Student* and r_{Person} in *Person.*

Subclasses can be inclusive, i.e. a multiobject may contain two objects o_{Ci} and o_{Cj} where C_i and C_j are sibling classes. For example, the class *Person* has another subclass *Employee* (see fig. 1.a). A person who is a student *and* an employee is implemented by a multiobject containing three objects: r_{Person} in *Person,* $r_{Student}$ in *Student* and $r_{Employee}$ in *Employee* (see fig. 1.b). There is therefore no need to add an artificial intersection class *Student&Employee* as with the classical implementation of specialization.

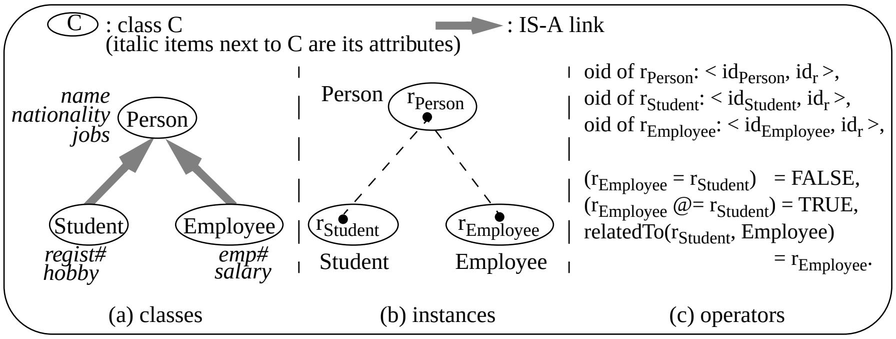

C : class C
(italic items next to C are its attributes)
: IS-A link
name
nationality
jobs
Person
Student
Employee
regist#
hobby
emp#
salary
Person
rPerson
rStudent
rEmployee
Student
Employee
oid of rPerson: < idPerson, idr >,
oid of rStudent: < idStudent, idr >,
oid of rEmployee: < idEmployee, idr >,
(rEmployee = rStudent) = FALSE,
(rEmployee @= rStudent) = TRUE,
relatedTo(rStudent, Employee)
= rEmployee.
(a) classes
(b) instances
(c) operators

Fig. 1. Implementing a student-employee

An object o_C in class *C* has the oid <id_C, id_o> where id_C is the class identifier and id_o the instance identifier within *C* (as in *Orion* [8]). Two objects p_E and q_F are *related,* i.e. they belong to the same multiobject, if:

– their classes *E* and *F* are in the same specialization tree, and
– they have the same instance identifier ($id_p = id_q$).

Two objects p_E and q_F are *identical*, if:

– they have the same oid (i.e. same class identifier and same instance identifier).

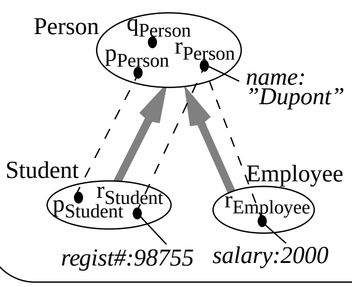
The operator "=" checks if two objects are identical, while the operator "@=" checks if two objects are related. The function *relatedTo(*o_D*, C)* returns the object in class *C* which is related to o_D (*C* and *D* being two classes in the same specialization tree). In other words, this function returns the object which has the oid <id_C, id_o>. If no such object exists in *C*, it returns the unknown object. Note that in other approaches this function is called casting or coercion. Examples are given in figure 1.c.

Since an entity may gain and lose facets during its life-time, objects can be added to and removed from its corresponding multiobject (see §2.2).

Querying a Multiobject. In the multiobject mechanism, attributes are not inherited in the classical sense of inheritance; they are *reached* by navigating in a specialization tree. This contrasts with the classical implementation of specialization. For the objects of a subclass *C*, only the values on attributes locally defined at *C* are stored. The values on attributes defined at the superclass *S* of *C* are not stored with *C* objects but with their related *S* objects. For example, in figure 1, if the name of r_{Person} is "Dupont" and $r_{Employee}$ in *Employee* is related to r_{Person} then $r_{Employee}$ is named Dupont. The *name* values are stored with the objects of class *Person*.

While traditional inheritance is upwards, reaching attributes in the multiobject mechanism can be upwards, downwards and sideways. The *get*(o_C, *att*) primitive method reads the value that the multiobject containing the object o_C takes on the attribute *att*. Its algorithm (see appendix) is the following: if *att* is a local attribute of *C* then it returns the value of o_C on *att*, else if *att* is a local attribute of a class *D* belonging to the specialization tree of *C* then it returns the value of o_D on *att* where o_D is the object in class *D* which is related to o_C. Figure 2 shows some examples related to figure 1.

let r_{Person} be named "Dupont", $r_{Student}$ having the registration number 98755, and $r_{Employee}$ earning a salary of 2000 francs. Let $p_{Student}$ be another student (not employee) and q_{Person} a baby (neither student nor employee).



name($r_{Student}$) = "Dupont", /* upwards */
salary(r_{Person}) = 2000, /* downwards */
regist#($r_{Employee}$) = 98755, /* sideways */
emp#($p_{Student}$) raises error2,
/* $p_{Student}$ is not related to an employee object */
regist#(q_{Person}) raises error2,
/* q_{Person} is not related to a student object */
birthdate($r_{Employee}$) raises error1.
/* birthdate is not an attribute in the specialization tree of Employee */

Fig. 2. Reaching attributes upwards, downwards and sideways

An attribute of class *C* *is* always reached in the descendants of *C* (because each of their objects is related to an object in *C*) while it *may be* reached in the ancestors and sibling classes of *C* if some of their objects are related to an object in *C*.
We forbid homonym attributes in classes belonging to the same specialization tree; nevertheless we have the same potential of information as O_2 [1] and *Goose* [26].

2.2 Manipulating Multiobjects

The algorithms for creating, deleting and updating a multiobject are provided in the appendix and are implemented in the F2 OODBMS. We briefly describe them hereafter.

Creating a Multiobject. The *create*(*C*, [a_1:v_1, a_2:v_2, ..., a_p:v_p]) primitive method creates a multiobject including an object in class *C*. The automatic classification algorithm searches the classes of the multiobject in the specialization tree of *C* (beginning from the root), *SC* = {C_1, C_2, ..., C_n}, according to the attribute values [a_1:v_1, a_2:v_2, ..., a_p:v_p] and to the classes' specialization constraints. If *C* does not belong to *SC* or if the origin class of one of the attributes a_j does not belong to *SC*, an error is returned.

Otherwise, an object o_{Ci} is added to each class C_i of *SC* and all these objects carry the same instance identifier. Each attribute value is stored with the object o_{Ci} which belongs to the origin class of the attribute. This contrasts with the classical implementation of specialization where an object is created in one most specific class.

For example, in figure 3.a, the class *Person* has two subclasses: *Employee* which in turn has two subclasses *ManEmp* (for men employees) and *WomEmp* (for women employees), and *Student* which in turn has a subclass *SwissSt* (for swiss students). The following expression (in F2-DML) creates a multiobject containing four objects: o_{Person} in *Person* (root class of the specialization tree), $o_{Employee}$ in *Employee* (the constraint *jobs includes "employee"* is satisfied), o_{ManEmp} in *ManEmp* (the constraint *sex = "m"* is satisfied) and $o_{Student}$ in *Student* (the constraint *jobs includes "student"* is satisfied) (see fig. 3.b).

```
    oEmployee := create Employee' [name: "Dupont", sex: "m",
nationality: "french", jobs: ("employee", "student"), emp#: 125,
salary: 2500];
```

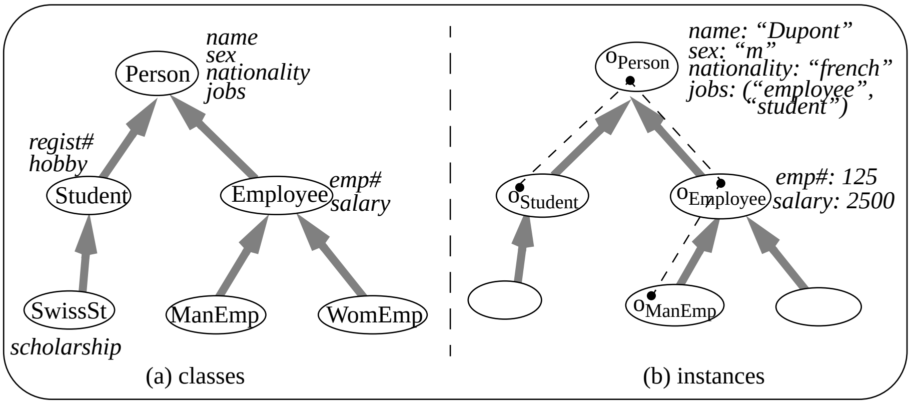


Fig. 3. Creating a multiobject

Note that if we did not give a value on the *jobs* attribute, the creation would be rejected because the multiobject would not include an object in *Employee* (*Employee* constraint not satisfied).

Deleting a Multiobject. The *delete(o_C)* primitive method deletes the multiobject containing the object o_C, i.e. it removes o_C and *all* its related objects. The algorithm searches the classes of the multiobject in the specialization tree of *C* (beginning from the root), $SC = \{C_1, C_2, ..., C_n\}$, and removes its object o_{Ci} from each class C_i of *SC*. This contrasts with the classical implementation of specialization where an object is deleted from one class.

For example, the following expression (in F2-DML) deletes a multiobject (representing a male student-employee) by removing all its objects o_{Person}, $o_{Employee}$, o_{ManEmp} and $o_{Student}$ (see fig. 4).

```
    delete oEmployee;
```

If instead one dismisses the employee, the entity remains as a student and a person. This can be done by updating the *jobs* attribute of the corresponding multiobject.

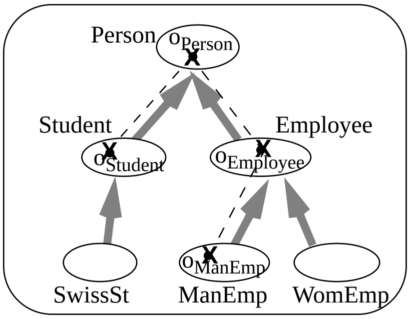


Fig. 4. Deleting a multiobject

Updating a Multiobject. The *update(o_C, [att:val])* primitive method sets the value of the multiobject containing the object o_C on the attribute *att* to *val*. Like the *get* method, *update* searches the attribute *att* upwards, downwards and sideways. Since the attribute *att* could be used in specialization constraints on the descendants *SD* of its origin class *Orig*, the multiobject may gain new objects or/and lose existing objects in *SD* because it may now (with the new value *val*) validate or invalidate those specialization constraints. This is referred to as object migration. The automatic classification algorithm searches in the specialization tree of *C*, beginning from *Orig*: (i) the set of gained classes and adds an object (carrying the same instance identifier as o_C) to each of them; (ii) the set of lost classes and removes the related object to o_C from each of them. This contrasts with the classical implementation of specialization where an object stays in its class until it is deleted from it.

For example, figure 5.a shows a swiss female student implemented by a multiobject containing three objects $\{p_{Person}, p_{Student}, p_{SwissSt}\}$. The following expression (in F2-DML) expresses that this person ceases to be a student and becomes an employee. As a result (see fig. 5.b), (i) $p_{Employee}$ is added to *Employee* and p_{WomEmp} is added to *WomEmp*, (ii) $p_{Student}$ and $p_{SwissSt}$ are removed from *Student* and *SwissSt* respectively. The multiobject contains now the objects $\{p_{Person}, p_{Employee}, p_{WomEmp}\}$.

```
update pPerson jobs:("employee");
```

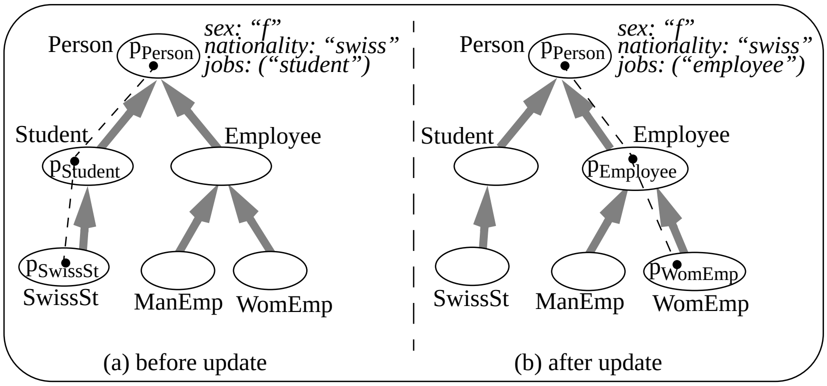


Fig. 5. Updating a multiobject (jobs attribute)

3 Advantages of the Multiobject Mechanism for Schema Evolution

In this section we first provide briefly the framework of schema evolution in F2. Then we discuss the advantages of the multiobject mechanism with respect to schema evolution.

3.1 Schema Evolution in F2

Set of Schema Changes in F2. An important feature of the F2 DBMS is the uniformity of its objects described in [6] [3]. We consider objects of three levels: database objects, schema objects and meta-schema objects. Uniformity of objects in F2 includes:

- uniformity of representation. The same structures are used in F2 to represent database objects, schema objects and meta-schema objects;
- uniformity of access and manipulation. The same primitive methods are used in F2 to access and manipulate database objects, schema objects and meta-schema objects.

Thanks to the uniformity of the F2 DBMS, we built the set of schema changes in F2 as follows [6] [3]: for *each* class of the F2 meta-schema we apply the primitive methods *create*, *delete* and *update* on its objects (see fig. 6).

Semantics of Schema Changes. We defined the semantics of each schema change in F2 with pre-conditions and post-actions [6] [3] such that the F2 model invariants are preserved. Pre-conditions must be satisfied to allow a schema change to occur; otherwise it is rejected. Post-actions are repercussions to be executed on schema objects and database objects in order to keep the database structurally consistent. We implemented pre-conditions and post-actions by triggered methods [6] [3].

Propagation of Schema Changes. In F2 schema changes are propagated immediately [3], i.e. the repercussions of a schema change are executed as soon as the schema change is performed.

3.2 Multiobject Mechanism and Schema Evolution

Since the multiobject mechanism implements specialization, we consider among the schema changes of F2 (fig. 6) those which are involved in specialization: create a subclass (1.3), delete a class (2), change the superclass of a subclass (3.4), update a class from non-subclass to subclass (3.5) and the reverse (3.6), create (4) and delete an attribute (5), change the domain class (6.4) and the origin class of an attribute (6.5), create (10) and delete a specialization constraint (11), change the list of subclasses on which is defined a specialization constraint (12.2). By examples we will show that the multiobject mechanism makes these schema changes more pertinent and easier to implement than with the classical implementation of specialization. We will compare F2 with the following OODBMS which support schema evolution: *Orion* [8], *Gemstone* [29], *OTGen* [23], *Cocoon* [37], *Goose* [26] and *O_2* [18]. All these systems support the classical

(1) Create a new class
- (1.1) Create an atomic class
- (1.2) Create a tuple class
- (1.3) Create a tuple subclass

(2) Delete an existing class
(3) Update an existing class
- (3.1) Change its name
- (3.2) Change its interval if atomic class
- (3.3) Change its maximal length if atomic string class
- (3.4) Change its superclass
- (3.5) Make it a subclass, i.e. attach it to a specialization tree
- (3.6) Make it a non-subclass, i.e. detach it from a specialization tree

(4) Create a new attribute of a class
(5) Delete an existing attribute
(6) Update an existing attribute
- (6.1) Change its name
- (6.2) Change its maximal cardinality
- (6.3) Change its minimal cardinality
- (6.4) Change its domain class
- (6.5) Change its origin class

(7) Create a new key of a class
(8) Delete an existing key
(9) Update an existing key
- (9.1) Change the class on which it is defined
- (9.2) Change its attributes
- (9.3) Enable / disable it

(10) Create a new specialization constraint
(11) Delete an existing spec. constraint
(12) Update an existing spec. constraint
- (12.1) Change its name
- (12.2) Change the list of subclasses on which it is defined

(13) Create a new trigger
(14) Delete an existing trigger
(15) Update an existing trigger
- (15.1) Change the event for which it is defined
- (15.2) Change the list of methods it triggers

(16) Create a new event
(17) Delete an existing event
(18) Update an existing event
- (18.1) Change the class on which it is defined
- (18.2) Change its kind
- (18.3) Change its attribute

(19) Create a new triggered method
(20) Delete an existing triggered method
(21) Update an existing triggered method
- (21.1) Change its name

Fig. 6. F2 schema changes

implementation, except *Cocoon* which supports multiple instantiation, class predicates and automatic classification.

Create a Subclass. Example: The class *Person* has several attributes including *nationality*. It has four objects {*a, b, c, d*}, two of them {*a, d*} take the value "swiss" on the *nationality* attribute. The class *Car* has an attribute *owner* whose domain is *Person*. Now one creates the class *Swiss* as a subclass of *Person*. The wanted effect is that *a* and *d* become objects of the *Swiss* class.

• Multiobject approach: The class *Person* has four objects {a_P, b_P, c_P, d_P} (see fig. 7.a). When the *Swiss* subclass and a specialization constraint on it are added, the objects a_S and d_S are automatically added to the *Swiss* subclass because they satisfy its specialization constraint (see fig. 7.b). Each of the multiobjects *a* and *d* contain now two objects. The attribute values of objects a_P and d_P are not copied because attribute values are locally stored and attributes of *Person* are reached in *Swiss*.
Cocoon supports this schema change.

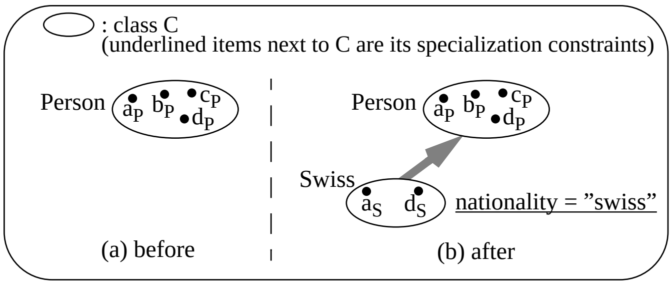


Fig. 7. Add the Swiss subclass with a specialization constraint

• Classical approach: Most of the OODBMS supporting schema evolution leave a subclass empty (without objects) when it is newly created. Thus in our example, a tool should be developed to: i) create two new objects in *Swiss*; ii) copy the value of objects *a* and *d,* on all the attributes of *Person,* to the newly created objects respectively; iii) delete the objects *a* and *d* in *Person*. If the objects *a* and *d* were referenced by *Car* objects through the *owner* attribute, the tool has also to update these references (to reference now the new objects in *Swiss*).
Only O_2 (thanks to migration functions) and *OTGen* (thanks to boolean expressions) allow subclass creation with object migration down.

Delete a Class. Example: The class *Swiss* is a subclass of *Person* and has two objects {*a*, *d*}. The class *Chalet* has an attribute *owner* whose domain is *Swiss*. Now one is no more interested to classify the swiss persons and decides to delete the *Swiss* class. The wanted effect is that *a* and *d* become objects of the *Person* class (keep the swiss people as persons).
• Multiobject approach: The *Swiss* class has two objects a_S and d_S which are related to a_P and d_P respectively in *Person* (see fig. 7.b). When the subclass *Swiss* is deleted, its objects are removed while their related objects $\{a_P, d_P\}$ remain in *Person* (see fig. 7.a). The attributes of *Swiss* are deleted and the domain of the *owner* attribute is updated to *Person,* as with the classical approach. The values on the *owner* attribute remain unchanged (see update the domain of an attribute).
Cocoon supports this schema change.

• Classical approach: Most of the OODBMS supporting schema evolution delete the objects of a class when the class is deleted (*Gemstone* prevents the deletion of a class if it is not empty). This implies loss of information. Thus in our example, a tool should be developed to: i) create two new objects in *Person*; ii) copy the value of objects *a* and *d,* only on the inherited attributes from *Person,* to the newly created objects respectively; iii) delete the *Swiss* class (consequently its objects are deleted). If the objects *a* and *d* were referenced by *Chalet* objects through the *owner* attribute, the tool has also to update these references (to reference now the new objects in *Person*).

Note that if the wanted effect in our example was not to keep the swiss persons, this could be achieved in the multiobject approach by first deleting the multiobjects containing *Swiss* objects and then deleting the *Swiss* class. Thus both semantics are possible in our approach and the database administrator can choose the most suitable for a given situation.

Update the Superclass of a Subclass. Example: The class *Person* has two subclasses *Student* and *Employee*. *Student* has a subclass *Young* (see fig. 8.a). Now one updates the superclass of *Young* from *Student* to *Employee*. The wanted effect is that the *Young* class stores the young employee objects instead of the young student objects, and that it inherits the attributes of *Employee* instead of those of *Student*.

• Multiobject approach: The class *Young* has the specialization constraint *age < 30*. Changing its superclass to *Employee* reclassifies automatically its objects: i) for each object in *Young*, if it is not related to an *Employee* object, it is removed from *Young*; ii) for each object in *Employee* whose *age* value is under 30, a related object to it is added to *Young* (if it does not already exist). The *Young* class reaches now another set of attributes; the physical storage of its objects remains unchanged.

• Classical approach: Most of the OODBMS supporting schema evolution keep the same objects in a subclass when modifying its superclass. This may lead to an inconsistent semantics. They re-evaluate the inheritance of the subclass according to defined rules. In our example, the *Young class* inherits now the attributes of *Employee* instead of those of *Student*. This results in the modification of the physical storage of *Young* objects. To reclassify objects in the *Young* class, a tool should be developed to: i) migrate the *Young* objects up to *Student* (like when deleting a class); ii) migrate some objects of *Employee* down to *Young* (like when creating a subclass). Note that *Gemstone* does not support this schema change in order to only provide well understood schema changes.

In F2, updating a subclass to non-subclass and the reverse are special cases of updating the superclass of a subclass. Due to lack of space we do not describe them.

Create an Attribute. Example: One adds the attribute *hobby* to class *Person* which has several descendants. The wanted effect is that the descendants of *Person* inherit *hobby.*

• Multiobject approach: Adding the new attribute *hobby* to class *Person* does not need to be propagated to the descendants of *Person*; instead *hobby* will be reached in them.

• Classical approach: Most of the OODBMS supporting schema evolution, as *Orion* and *Gemstone,* propagate recursively this schema change to *Person*'s subclasses according to propagation rules. On storage, this requires to physically add the attribute to the objects of each descendant of *Person* inheriting it.

Delete an Attribute. Example: One deletes the attribute *hobby* from class *Person*. The wanted effect is that the descendants of *Person* do no longer inherit *hobby.*

• Multiobject approach: Removing the attribute *hobby* from class *Person* does not need to be propagated to the descendants of *Person*; instead *hobby* will not be reached in them.

• Classical approach: There are different approaches, *Orion* removes recursively the attribute *hobby* from subclasses inheriting it while *Gemstone* does not. In the latter case, one has to delete the *hobby* attribute from each class inheriting it. On storage, both approaches require to physically delete the attribute from the objects of each descendant of *Person* inheriting it.

Update the Domain Class of an Attribute. Example: The class *Book* has an attribute *owner* whose domain is *Person*. *Student* is a subclass of *Person*. The book *book1* is owned by *c* (student object) and *book2* is owned by *d* (person object). Now one updates the domain of *owner* to *Student*.

• Multiobject approach: As with the classical approach, the *owner* value of *book1* is unchanged: c_S in *Student* is related to c_P in *Person* and they have the same instance identifier. In F2, only instance identifiers are physically stored in attribute values (the class identifier is the same for all values on the same attribute; it can be obtained by getting the domain class of the attribute). The *owner* value of *book2* is replaced by the unknown object. Updating the domain class of *owner* does not need to be propagated to the descendants of *Book*, because the *owner* values are stored with *Book* objects. Note that one can update the domain of *owner* to an ancestor, a descendant or a sibling class of *Person*.

• Classical approach: All the OODBMS supporting schema evolution support this schema change but some with restrictions. For example, *Orion* allows only to generalize a domain while *Gemstone* can generalize and specialize it. *Orion* propagates this schema change to subclasses inheriting the attribute according to propagation rules while *Gemstone* does not.

Update the Origin Class of an Attribute. Example: The class *Young* is a subclass of *Student* which is a subclass of *Person* (see fig. 8.a). Class *Person* has the objects {*d*}, class *Student* has {*c*} and class *Young* has {*a*, *b*}. Class *Employee*, another subclass of *Person*, has the objects {*e*}. *Hobby* is a local attribute of *Student*. Now one wants to store the hobbies of all persons and updates the origin class of *hobby* to *Person*. Updating the origin class of an attribute is very useful and is not equivalent to dropping the attribute and adding it to another class because in this case the values taken on the attribute are lost.

• Multiobject approach: The classes and their objects are shown in figure 8.a. When the origin class of the *hobby* attribute is updated to *Person* (see fig. 8.b), the objects a_P, b_P, c_P keep the same *hobby* value as a_S, b_S, c_S respectively. The *hobby* value of d_P and e_P is set to unknown. Thanks to the transposed storage of objects [6] in F2, the *hobby* values are not copied. Note that one can update the origin class of *hobby* from *Student* to *Person* (ancestor), to *Young* (descendant) or to *Employee* (sibling class). Changing the origin class of an attribute does not need to be propagated to the descendants of its old and new origin classes; the attribute will be reached in another set of classes.
Cocoon supports this schema change.

• Classical approach: Most of the OODBMS supporting schema evolution do not support this schema change. In our example, a tool should be developed to: i) create a new attribute *hobby2* in *Person* (it becomes inherited in *Student*, *Young* and *Employee*); ii) copy the *hobby* values on this new attribute for *Student* objects {*c*} and *Young* objects {*a*, *b*}; iii) delete the attribute *hobby* of *Student*; iv) rename the attribute *hobby2* to *hobby*. Note that if instead one creates a new attribute *hobby* in *Person*, the semantics would be different because the attribute *hobby* of *Student* would not be considered as inherited from *Person*.
Only *Goose* allows to update the origin class of an attribute with retaining values for objects.

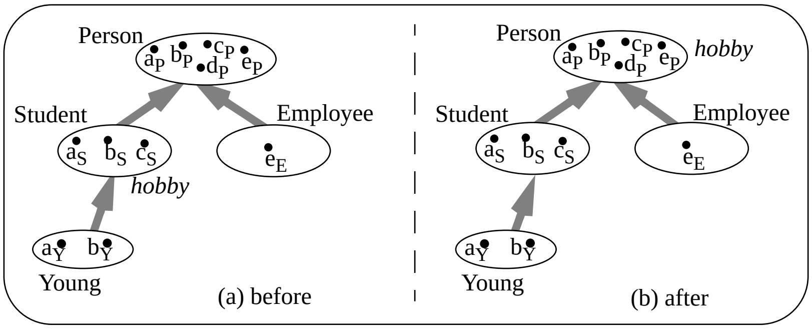


Fig. 8. Update the origin class of the hobby attribute

Create/Delete a Specialization Constraint. Example: The class *Person* has a subclass *European* (citizen of a country of the european community) which has a subclass *Young* (under 30 years old) (see fig. 9.a). Class *Young* has one object *a* (young european), class *European* has one object *b* (old european), class *Person* has two objects *c* (young african) and *d* (young swiss). Suppose now that Switzerland joins the european community. One replaces (delete followed by create) then the specialization constraint of *European* by a new one taking into account Switzerland. The wanted effect is that the object *d* belongs now to *Young* instead of *Person*.

• Multiobject approach: The classes and their objects are shown in figure 9.a. Specialization constraints are defined on the subclasses *European* and *Young*. When the specialization constraint of *European* is replaced, the objects d_E and d_Y are automatically added to the classes *European* and *Young* respectively (see fig. 9.b).
Cocoon supports class predicates and allows to change them.

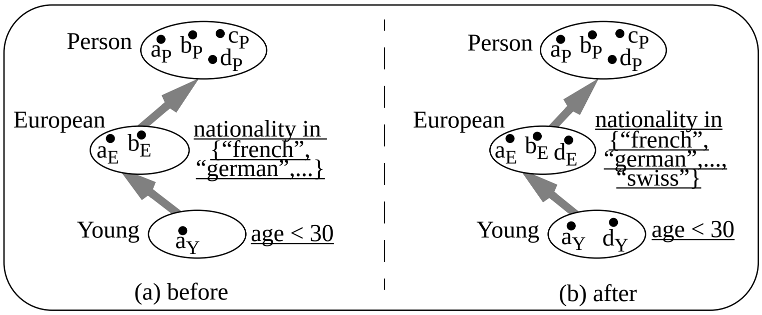

Person
European
nationality in {"french", "german",...}
Young
age < 30
(a) before
nationality in {"french", "german",..., "swiss"}
(b) after

Fig. 9. Replace the specialization constraint of European

• Classical approach: Most of the OODBMS do not support specialization constraints. In our example, a tool should be developed to migrate the object *d* down to *Young*.

In F2, a specialization constraint (e.g. *age < 30*) can be defined on several subclasses (e.g. *YoungStudent* and *YoungEmployee*). Changing the list of subclasses of a specialization constraint is similar to create/delete a specialization constraint.

4 Related Work

Approaches for Multi-faceted and Dynamic Entities. We proposed and implemented a previous version of the multiobject mechanism in the extended entity-relationship DBMS *Ecrins* [20]. Several object approaches have been proposed to model the multi-faceted and dynamic nature of entities. We summarize them in the following table. Then we compare our approach to them. Null entries (--) mean not known (we do not have the data).

	approach (how an entity is implemented)	creation	extension	object internal represen-tation	inheritance
Object hier-archies [34]	entity: set of objects (object hierarchy) (parent attribute)	create an object hierar-chy	add/remove an object to/from an object hierarchy	local attributes	upwards in object hierar-chy (per-object)
Roles in ORM [30]	entity: object of a class + role instances	create an object in a class	add/remove a role instance to/from an object	--	no inheritance
Aspects [33]	entity: object of a class + aspect instances (same oid)	create an object in a class	add an aspect to an object	--	no inherit-ance (aspect exports selected parts of a class)
Roles in Fibonacci [2]	entity: object with a set of roles (same identity)	create an object with several roles	add/drop a role to/ from an object	local attributes	upwards in role hierarchy
Category classes [28]	entity: object of several classes (roles) (same oid)	create an object in sev-eral classes	add/remove a role to/from an object (manual), or update an object (automatic)	--	upwards in class hierar-chy
Object-slicing [22]	entity: concep-tual object + set of implementa-tion objects (bi-directional link)	create a con-ceptual object with imple-mentation objects	create/delete an implementation object	local attributes	upwards in class hierar-chy
Our approach: Multiobjects	entity: set of objects (multi-object) (same instance identifier)	create a multi-object (auto-matic classification)	update a multiob-ject (objects are added/removed to/ from a multiobject automatically)	local attributes	upwards, downwards and sideways in specializa-tion tree

The multiobject mechanism differs from *ORM roles* and *aspects* because they do not integrate the class hierarchy. As in *object hierarchies*, *Fibonacci roles*, *category classes* and *object-slicing*, an entity in our approach is implemented by a set of objects which can be enlarged and reduced (automatically as in *category classes*). However, our approach has several differences:

– an object can access attribute values of related objects not only upwards but also downwards and sideways in the class hierarchy.
– two related objects are neither linked by a parent attribute as in *object hierarchies*, nor by a bi-directional link via a conceptual object as in *object-slicing*. Both objects have the same instance identifier (same identity in *Fibonacci roles* and *category classes*). Thus there is no overhead when accessing related objects in our approach.
– we provide the algorithms to manipulate multiobjects.
– we use the multiobject mechanism not only to ease object modelling and object evolution but also to ease schema evolution.

OODBMS Supporting Schema Evolution. Several OODBMS support schema evolution. We can classify them in three categories: schema evolution without versioning (*Orion* [8], *Gemstone* [29], *OTGen* [23], *Cocoon* [37], *Goose* [26], *O2* [18]), schema evolution with versioning (*Encore* [35], *Orion*, *Goose*, *Closql* [25]), schema evolution with views (*Contexts* [7], *Goose*, *Views* [11], *TSE* [32]). The two last categories are out of the scope of this paper. Approaches of the first category differ by the set of supported schema changes, the semantics of schema changes, and the propagation of schema changes (immediate, deferred, mixed).

5 Conclusion

We presented the multiobject mechanism to implement specialization in object-oriented databases. We described how this mechanism models multi-faceted and dynamic real-world entities. We showed its advantages with respect to schema evolution. The multiobject mechanism makes schema changes more pertinent than with the classical implementation of specialization. It makes them easier to implement and less time-consuming since it needs neither to copy objects nor to propagate schema changes. It makes schema changes easier to understand since it avoids complex propagation rules. We implemented the multiobject mechanism in F2. We submitted the F2 DBMS to the OO7 benchmark [14]. We obtained in [3] good results for the queries and traversals, because objects in the multiobject approach have smaller size than objects in the classical object model; this consequently reduces the number of input/output operations. For the insert and delete operations of the benchmark, the multiobject approach is more expensive than the classical object model; this is due to the automatic classification (which is not supported in the classical object model) and to the fact that several objects are created/deleted instead of one. The execution times could be improved by using a parallel algorithm; this issue deserves to be investigated.

Extensions of the multiobject mechanism in F2 include: i) allow several objects of the same class in a multiobject; ii) allow several attributes with the same name in a specialization tree; iii) handle methods and virtual binding (see interpretation of messages in [2]); iv) model the life-cycle of an entity [4] and restrict the way objects may be added to and removed from a multiobject. Extensions of schema evolution in F2 include: i) take into account schema changes on methods; ii) study the behavioural consistency of the database; iii) test schema changes in a real application.

References

1. Adiba M., Collet C., *Objets et bases de données: le SGBD O_2*, Hermès, 1993.
2. Albano A., Bergamini R., Ghelli G., Orsini R., *An object Data Model with Roles*, Proc. Int. Conf. on Very Large Data Bases, VLDB, Dublin 1993.
3. Al-Jadir L., *Evolution-Oriented Database Systems*, Ph.D. thesis, Faculty of Sciences, University of Geneva, 1997.
4. Al-Jadir L., Falquet G. , Léonard M., *Context Versions in an Object-Oriented Model*, Proc. Int. Conf. on Database and Expert Systems Applications, DEXA, Prague 1993.
5. Al-Jadir L., Le Grand A., Léonard M., Parchet O., *Contribution to the Evolution of Information Systems*, in: Methods and Associated Tools for the Information Systems Lifecycle, A.A. Verrijn-Stuart & T.W. Olle (eds), IFIP, Elsevier, 1994.
6. Al-Jadir L., Estier T., Falquet G., Léonard M., *Evolution Features of the F2 OODBMS*, Proc. Int. Conf. on Database Systems for Advanced Applications, DASFAA, Singapore 1995.
7. Andany J., Léonard M., Palisser C., *Management of Evolution in Databases*, Proc. Int. Conf. on Very Large Data Bases, VLDB, Barcelona 1991.
8. Banerjee J., Kim W., Kim H-J., Korth H.F., *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, Proc. Int. Conf. on Management Of Data, ACM SIGMOD, San Francisco 1987.
9. Barbedette G., *Schema Modifications in the LISPO2 Persistent Object-Oriented Language*, Proc. European Conf. on Object-Oriented Programming, ECOOP, Geneva 1991.
10. Bellahsene Z., *An Active Meta-model for Knowledge Evolution in an Object-oriented Database*, Proc. Int. Conf. on Advanced Information Systems Engineering, CAISE, Paris 1993.
11. Bertino E., *A View Mechanism for Object-Oriented Databases*, Proc. Int. Conf. on Extending Database Technology, EDBT, Vienna 1992.
12. Bertino E., Jajodia S., *Modeling Multilevel Entities Using Single Level Objects*, Proc. Int. Conf. on Deductive and Object-Oriented Databases, DOOD, Phoenix 1993.
13. Bonjour M., Falquet G., *Concept Bases: A Support to Information Systems Integration*, Proc. Int. Conf. on Advanced Information Systems Engineering, CAISE, Utrecht 1994.
14. Carey M.J., DeWitt D.J., Naughton J.F., *The OO7 Benchmark*, Proc. Int. Conf. on Management Of Data, ACM SIGMOD, Washington 1993.
15. Estier T., Falquet G., Guyot J., Léonard M., *Six Spaces for Global Information Systems Design*, in: The Object Oriented Approach in Information Systems, F. van Assche & B. Moulin & C. Rolland (eds), IFIP, North-Holland, 1991.
16. Falquet G., *Interrogation de bases de données à l'aide d'un modèle sémantique*, Ph.D. thesis, Faculty of Sciences, University of Geneva, 1989.
17. Falquet G., Léonard M., Sindayamaze J., *F2Concept: a Database System for Managing Classes' Extensions and Intensions*, in: Information modelling and knowledge bases V, H. Jaakola et al. (eds), IOS Press, 1994.

18. Ferrandina F., Meyer T., Zicari R., Ferran G., Madec J., *Schema and Database Evolution in the O2 Object Database System*, Proc. Int. Conf. on Very Large Data Bases, VLDB, Zürich 1995.
19. Hauck F.J., *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*, Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, Washington 1993.
20. Junet M., Falquet G., Léonard M., *ECRINS/86: An Extended Entity-Relationship Data Base Management System and its Semantic Query Language*, Proc. Int. Conf. on Very Large Data Bases, VLDB, Kyoto 1986.
21. Kambayashi Y., Peng Z., *Object Deputy Model and Its Applications*, Proc. Int. Conf. on Database Systems for Advanced Applications, DASFAA, Singapore 1995.
22. Kuno H.A., Ra Y-G., Rundensteiner E.A., *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation*, Technical Report, CSE-TR-241-95, University of Michigan, 1995.
23. Lerner B.S., Habermann A.N., *Beyond Schema Evolution to Database Reorganization*, Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, Ottawa 1990.
24. Ling T.W., Teo P.K., *Object Migration in ISA Hierarchies*, Proc. Int. Conf. on Database Systems for Advanced Applications, DASFAA, Singapore 1995.
25. Monk S.R., Sommerville I., *A Model for Versioning of Classes in Object-Oriented Databases*, Proc. British National Conf. on Databases, BNCOD, Aberdeen 1992.
26. Morsi M.M.A., Navathe S.B., Kim H-J., *A Schema Management and Prototyping Interface for an Object-Oriented Database Environment*, in: Object Oriented Approach in I.S., F. Van Assche & B. Moulin & C. Rolland (eds), IFIP, North-Holland, 1991.
27. Nguyen G.T., Rieu D., Escamilla J., *An Object Model for Engineering Design*, Proc. European Conf. on Object-Oriented Programming, ECOOP, Utrecht 1992.
28. Odberg E., *Category Classes: Flexible Classification and Evolution in Object-Oriented Databases*, Proc. Int. Conf. on Advanced Information Systems Engineering, CAISE, Utrecht 1994.
29. Penney D.J., Stein J., *Class Modification in the GemStone Object-Oriented DBMS*, Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, Orlando 1987.
30. Pernici B., *Objects with Roles*, Proc. IEEE Conf. on Office Information Systems, 1990.
31. Peters R.J., Özsu M.T., *An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems*, ACM Transactions on Database Systems, vol. 22, no 1, march 1997.
32. Ra Y.G., Kuno H.A., Rundensteiner E.A., *A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views*, Technical Report, CSE-TR-215-94, University of Michigan, april 1994.
33. Richardson J., Schwarz P., *Aspects: Extending Objects to Support Multiple, Independent Roles*, Proc. Int. Conf. on Management Of Data, ACM SIGMOD, Denver 1991.
34. Sciore E., *Object Specialization*, ACM Transactions on Information Systems, vol. 7, no 2, april 1989.
35. Skarra A.H., Zdonik S.B., *Type Evolution in an Object-Oriented Database*, in: Research Directions in OO Programming, B. Shriver & P. Wegner (eds), MIT Press, 1987.
36. Smith J.M., Smith D.C.P., *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems, vol. 2, no 2, june 1977.
37. Tresch M., *A Framework for Schema Evolution by Meta Object Manipulation*, Proc. Int. Workshop on Foundations of Models and Languages for Data and Objects, Aigen 1991.

Appendix

1. Get Algorithm

Let att be an attribute and o_C be an object of class C (an object carries the class it is instance of, thus C can be known from o_C); origin_class(att) be a function which returns the origin class of attribute att; root(C) be a function which returns the root class of the specialization tree of class C.

```
function get(o_C, att) return val is
begin
   /* call the check_valid_attribute procedure which
   checks that att is a local attribute of C (sets obj to
   o_C) or a reached attribute by o_C (sets obj to the
   object in the origin class of att which is related to
   o_C) */
   check_valid_attribute(att, o_C, obj);
   return the value val of obj on att;
end get;

procedure check_valid_attribute(att, o_C, obj) is
begin
   Orig := origin_class(att);
   /* if att is a local attribute of class C */
   if (C = Orig) then
      obj := o_C;
   /* if classes Orig and C are in same spec. tree */
   elsif (root(C) = root(Orig)) then
      o_Orig := relatedTo(o_C, Orig);
      /* if o_C is related to an object in Orig */
      if (o_Orig ≠ unknown_object) then
         obj := o_Orig;
      else error2;
      end if;
   else error1;
   end if;
end check_valid_attribute;
```

2. Create Algorithm

Let C be a class and $[a_1{:}v_1, a_2{:}v_2, ..., a_p{:}v_p]$ be an array of <attribute:value> pairs; subclasses(C) be a function which returns the direct subclasses of class C; satisfy_constraints($[a_1{:}v_1, a_2{:}v_2, ..., a_p{:}v_p]$, C) be a function which indicates whether the given attribute values satisfy all the spec. constraints of class C; root() and origin_class() have been defined before.

```
function create(C, [a_1:v_1, a_2:v_2, ..., a_p:v_p]) return
o_C is
begin
   TheRoot := root(C);
   /* initialize SC to the empty set */
   SC := {};
   /* call the classify procedure which puts in SC the
   classes of the new multiobject */
   classify(TheRoot, [a_1:v_1, a_2:v_2, ..., a_p:v_p], SC);

   /* check that C belongs to SC */
   if (C not in SC) then error1;
   end if;
   /* check that all the given attributes will be
   reached by the new multiobject */
   for each attribute a_i in [a_1:v_1, a_2:v_2, ..., a_p:v_p]
   loop
      if (origin_class(a_i) not in SC) then error2;
      end if;
   end loop;

   /* add objects, store attribute values */
   ID := the instance identifier for the objects of
         the new multiobject;
   for each class C_i in SC loop
      add an object o_Ci (having the instance identifier
            ID) to C_i;
      store the value of o_Ci on the attributes whose
      origin class is C_i;
   end loop;
   return o_C;
end create;

procedure classify(C, [a_1:v_1, a_2:v_2, ..., a_p:v_p],
TheSet) is
begin
   /* add class C to TheSet */
   TheSet := TheSet ∪ C;
   Sub := subclasses(C);
   for each S_i in Sub loop
      /* if the attribute values satisfy all the spec.
      constraints of class S_i */
      if satisfy_constraints([a_1:v_1, a_2:v_2, ..., a_p:v_p],
                             S_i) then
         classify(S_i, [a_1:v_1, a_2:v_2, ..., a_p:v_p],
                  TheSet); -- recursive call
      end if;
   end loop;
end classify;
```

3. Delete Algorithm

Let o_C be an object of class C; root() and subclasses() have been defined before.

```
procedure delete(o_C) is
begin
   TheRoot := root(C);
   o_Root := relatedTo(o_C, TheRoot);
   /* initialize SC to the empty set */
   SC := {};
   /* call the facets procedure which puts in SC the
   classes of the multiobject containing o_Root */
   facets(o_Root, SC);

   /* remove objects */
   for each class C_i in SC loop
      o_Ci := relatedTo(o_C, C_i);
      remove the object o_Ci from C_i;
   end loop;
end delete;

procedure facets(o_C, TheSet) is
begin
   /* add class C to TheSet */
   TheSet := TheSet ∪ C;
   Sub := subclasses(C);
```

```
    for each $S_i$ in Sub loop
        $o_{Si}$ := relatedTo($o_C$, $S_i$);
        /* if an object in $S_i$ is related to $o_C$ */
        if ($o_{Si}$ ≠ unknown_object) then
            facets($o_{Si}$, TheSet); -- recursive call
        end if;
    end loop;
end facets;
```

4. Update Algorithm

Let o_C be an object of class C and [a:v] be an <attribute:value> pair; constraint_on_att(C, att) be a function which indicates whether class C has a specialization constraint involving the attribute att; satisfy_constraints_att([a:v], C) be a function which indicates whether the value v satisfies the specialization constraints of class C involving the attribute a; check_valid_attribute(), origin_class(), subclasses(), satisfy_constraints(), facets() and classify() have been defined before.

```
procedure update($o_C$, [a:v]) is
begin
    check_valid_attribute(a, $o_C$, obj);
    Orig := origin_class(a);

    /* initialize ToAdd and ToRemove to empty set */
    ToAdd := {};
    ToRemove := {};
    /* call the reclassify procedure which puts in
    ToAdd the classes to which a related object will be
    added and in ToRemove the classes from which a
    related object will be removed */
    reclassify($o_C$, [a:v], Orig, ToAdd, ToRemove);

    /* add objects */
    for each class $C_i$ in ToAdd loop
        add an object $o_{Ci}$ (having the same instance
                identifier as $o_C$) to $C_i$;
    end loop;
    /* remove objects */
    for each class $C_i$ in ToRemove loop
        $o_{Ci}$ := relatedTo($o_C$, $C_i$);
        remove the object $o_{Ci}$ from $C_i$;
    end loop;

    /* store the new attribute value */
    store the new value v of the object obj on the
    attribute a;
end update;

procedure reclassify($o_C$, [a:v], D, TheSetAdd,
TheSetRemove) is
begin
    Sub := subclasses(D);
    for each $S_i$ in Sub loop
        /* if class $S_i$ has a spec. constraint involving the
        attribute a */
        if constraint_on_att($S_i$, a) then
            /* call the migrate procedure which (i) adds
            $S_i$ (and possibly its descendants) to
            TheSetAdd if a related object $o_{Si}$ should be
            added, (ii) adds $S_i$ (and possibly its
            descendants) to TheSetRemove if the related
            object $o_{Si}$ should be removed, (iii) sets
            continue to true if the related object $o_{Si}$ exists
            and stays in $S_i$ */
            migrate($o_C$, [a:v], $S_i$, TheSetAdd,
                    TheSetRemove, continue);
            if continue then
                reclassify($o_C$, [a:v], $S_i$, TheSetAdd,
                    TheSetRemove); -- recursive call
            end if;
        /* else if an object in $S_i$ is related to $o_C$ */
        elsif (relatedTo($o_C$, $S_i$) ≠ unknown_object) then
            reclassify($o_C$, [a:v], $S_i$, TheSetAdd,
                TheSetRemove); -- recursive call
        end if;
    end loop;
end reclassify;

procedure migrate($o_C$, [a:v], D, TheSetAdd,
TheSetRemove, continue) is
begin
    continue := false;
    /* is_related indicates if $o_C$ is related to an object
    in D (before update); will_be_related indicates if
    $o_C$ will be related to an object in D (after update)
    */
    $o_D$ := relatedTo($o_C$, D);
    is_related := ($o_D$ ≠ unknown_object);
    if is_related then
        /* does the new attribute value satisfy the spec.
        constraints of D involving the attribute a */
        will_be_related :=
                satisfy_constraints_att([a:v], D);
    else
        /* does the object $o_C$ (take into account the new
        attribute value) satisfy all the spec. constraints
        of D */
        will_be_related := satisfy_constraints(
                attribute values of $o_C$, D);
    end if;

    /* case 1: there is a related object in D and it will
    stay in it */
    if (is_related and will_be_related) then
        continue := true;
    /* case 2: there is a related object in D but it will
    be removed from it */
    elsif (is_related and not will_be_related) then
        /* add D and each of its descendants where an
        object is related to $o_C$ to TheSetRemove */
        tempRemove := {};
        facets($o_D$, tempRemove);
        TheSetRemove := TheSetRemove ∪
                                        tempRemove;
    /* case 3: no related object in D but a new one will
    be added to it */
    elsif (not is_related and will_be_related) then
        /* add D and possibly its descendants to
        TheSetAdd */
        tempAdd := {};
        classify(D, attribute values of $o_C$, tempAdd);
        TheSetAdd := TheSetAdd ∪ tempAdd;
    /* case 4: no related object in D and no one will
    be added to it */
    else null; -- nothing to do
    end if;
end migrate;
```