# Modelling Constraints with Exceptions in Object-Oriented Databases

N. Bassiliades and I. Vlahavas

Dept. of Informatics,
Aristotle University of Thessaloniki,
54006 Thessaloniki, Greece.
Email: {bassiliades,vlahavas}@olymp.ccf.auth.gr

**Abstract.** This paper deals with modelling constraints in object-oriented databases, with emphasis given on exceptions. Constraints are restrictions on properties and relations of database objects that ensure the integrity of data. Therefore, they should be obeyed by every object, but as in real-life, there are some exceptions to this rule. Object-oriented databases provide rich semantic constructs, adequate to model real-world relations. Inheritance of constraints in object-oriented databases has been treated in a completely mandatory way, providing no room for exceptions. In this paper, an object-oriented constraint representation scheme is presented, along with a methodology for modelling constraint exceptions. Finally, an algorithm is described that ensures correct run-time resolution of constraint applicability. Since business is not in abstract, but in real-world, business database modelling would be greatly benefited from a tool that allows both a clear definition and an efficient enforcement of constraints with exceptions.

## 1 Introduction

The imperative need for integrity maintenance in database systems is a long recognized fact [14]. Constraints ensure the validity of data held in the database, according both to the system and the user, and also ensure that subsequent updates on data will not violate these restrictions. Modern business database applications impose many complicated relations and restrictions among data, named semantic integrity constraints. There are many important research topics associated with constraints in databases, like constraint semantics, specification and enforcement.

Object-oriented databases (OODB) gained considerable amount of interest [13], because they can model real-world concepts for complex applications. OODBs gave a new rise in the interest for integrity constraints because of the new structural and semantic capabilities they offer. Constraint maintenance in OODBs differs from that in relational databases [15] mainly due to the localized control and rich modelling constructs of the former.

However, the rich modelling capabilities of OODBs have not been adequately used to model semantic constraints. Constraints have been either treated as code embedded in methods, as facets attached to slots, as demons or as general purpose rules. Only recently it has been shown how to abstract a constraint from the participating objects and keep it as a separate object in the database [2]. However, little attention has been paid so far to the inheritance scheme of constraints and the ability to represent exceptions.

In this paper we propose an object-oriented representation of constraints with emphasis on the modelling of constraint exceptions in OODBs. More specifically, we: a) identify problems associated with the default constraint inheritance scheme, b) propose a constraint descriptor to include the modelling tools adequate for representing constraint exceptions, and c) we propose a constraint checking algorithm that takes into account the information about exceptions stored within the constraints-objects.

The main contribution of this paper to business modelling is the ability that the described method gives to the database designer to easily insert and maintain semantic integrity constraints, along with their interrelations and exceptions. In this way, the database meets the requirements imposed by the complexity of modelling real-world organizational aspects of institutions, corporations, etc. Furthermore, the correctness and efficiency of constraint checking is guaranteed by a sophisticated algorithm that resolves constraint applicability ambiguities at run-time.

This paper is structured as follows: Section 2 presents the problems associated with the default constraint inheritance scheme; Section 3 presents the proposed object-oriented constraint representation scheme for exceptions; Section 4 gives examples on the use of the method for representing constraint exceptions; Section 5 presents the algorithm for constraint checking using the information about exceptions, and finally, section 6 concludes the paper with a summary of the main points and a discussion of future work.

## 2 Constraints with Exceptions

Constraints are mandatory features of an OODB; every object should normally satisfy all of them. However, there might be exceptions either at the instance or the class level. For example, some constraints might be disabled for some instances or some constraints may not be applicable to a class. Furthermore, some times a constraint is being overridden by a more specific one, to cater for efficiency or correctness of constraint semantics.

The above alternative behaviours can be met in various forms, as constraint contradiction, exception, subsumption, etc. All are factors that alternate the default constraint checking mechanism. However, the above features do not lead to database inconsistencies when the transaction model is tuple-oriented, as in OODBs, and constraint checking is of preventive nature.

In the following we present some problematic cases, using the database schema of fig. 1, which provides a simplified view of the organizational aspects of a Greek university. Universities are good, real-world examples of complex and deep hierarchically organized institutions. The constraints are expressed in CoLan [2], a functional constraint description language designed for the ADAM object-oriented database system [13]. However, CoLan is portable to a wide range of object-oriented data models, as it is shown in [10], where the same constraint language has been implemented for P/FDM [12], integrated with a simple transaction mechanism, called backtrackable updates, that allows the declarative description of updates, rather than their procedural specification.

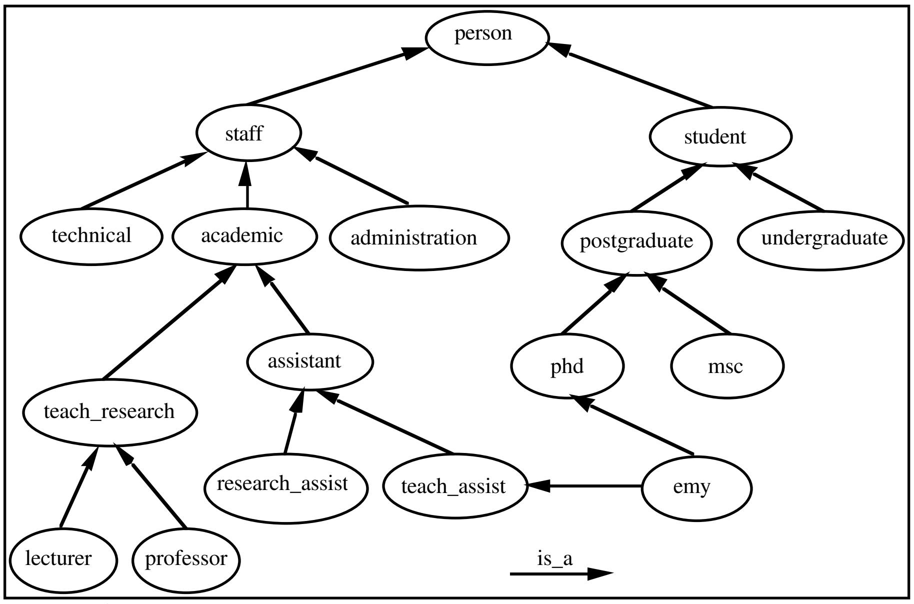

**Figure 1.** A (simplified) organizational schema of a Greek university

**Constraint Contradiction.** Contradicting constraints are the ones that try to impose opposite properties to objects. For example, constraints $C_1$ and $C_2$, with respect to the `emy` class[1], are contradicting:

```
C1:  forall s in staff
        income of s >= 100000
C2:  forall s in student
        income of s < 100000
```

When a message, like the following, is issued:

```
put_income([120000]) => 11%emy
```

then $C_1$ will not reject the update, while $C_2$ will do. The overall effect will be the rejection of the update. Since no update is introduced and the database was in a consistent state before the update, the new state is also consistent. However, the problem with contradicting constraints is that the database is over-constrained, so some valid transactions are wrongly rejected.

**Constraint Subsumption.** Constraint subsumption occurs when the set of the properties allowed by a constraint is a subset of the properties allowed by another. Then it is said that the first constraint subsumes the second. For example, constraint $C_3$ subsumes $C_4$:

---

[1] In the emy class belong PhD students, employed by the university to assist at teaching.

```
C3:  forall s in staff
        age of s >= 25
C4:  forall p in postgraduate
        age of s >= 22
```

An instance of class `emy` will not be allowed to be less than 25 years old, as $C_3$ imposes, while $C_4$ allows so. This happens because $C_3$ is more restrictive, i.e. it subsumes $C_4$. There are two alternative scenarios for the semantics of the database updates:

- The correct constraint is $C_3$, therefore the updates are (correctly) dominated by $C_3$. Checking $C_4$ is redundant, which will cause efficiency (but not correctness) problems.
- The correct constraint is $C_4$, but since $C_3$ dominates, young `emy` students will not be allowed. The database is again over-constrained.

**Constraint Cancellation.** Constraint cancellation is the exclusion of a constraint from guarding a class and its sub-classes. Constraint $C_5$ is valid for all employees (of the university), except for the academic staff, which do not have a fixed limit of working hours:

```
C5:  forall s in staff
        working_hours of s = 40
```

The above constraint is not valid for class `academic` and all its sub-classes, but since this class is a sub-class of `staff`, it inherits $C_5$ (incorrectly).

**Constraint Specialization.** Constraint specialization is the restriction of the set of allowed properties as we move downwards the class hierarchy. Constraints $C_6$ and $C_7$ apply to similar contexts, but $C_6$ is defined on a super-class of $C_7$:

```
C6:  forall p in person
        age of p > 0 and age of p =< 120
C7:  forall s in student
        age of s >= 18 and age of s =< 40
```

Constraint $C_7$ is more restrictive than $C_6$ and should be checked instead. However, both are checked and $C_7$ dominates. Efficiency reasons impose the need to avoid use-less checking of $C_6$.

**Constraint Overriding.** Constraint overriding is the replacement of a constraint by another one, which is defined at a sub-class of the former. The term "overriding" usually suggests that the constraints are contradicting, else this case is similar to constraint specialization. For example, constraints $C_8$ and $C_9$ apply to similar contexts, they are defined on the same branch of the class hierarchy and they are contradicting:

```
C8:  forall s in student
        years_of_study of s >= 3
```

```
C9:  forall m in msc
       years_of_study of m < 2
```

Both constraints are applicable to class `msc`, however only the second one should be enforced, otherwise no students in the `msc` course will be allowed, since either $C_8$ or $C_9$ will prohibit that.

**Instance Exceptions.** Some constraints might be disabled for certain objects. For example, $C_{10}$ prohibits students from having a permanent parking license at the university parking place:

```
C10:   forall s in student
         parking_license of s = false
```

However, student `23%undergraduate` is disabled and must have a parking place. Using the default scheme for constraint inheritance this is not possible. One may think that this information about disabled students should be added to $C_{10}$, using a complex condition:

```
C10':  forall s in student
         (is_disabled of s = true or
            parking_license of s = false) and
         (is_disabled of s = false or
            parking_license of s = true)
```

However, the complex condition imposes extra computational cost that is not necessary for the most instances. Furthermore, if there are other exceptions to constraint $C_{10}$, the condition (and the overhead) will grow more. Therefore, excepting one or more instances from $C_{10}$ would be more efficient.

### 2.1 Related Work

Representing exceptions in OODBs [5], in logic programming [16] and in Artificial Intelligence in general [6, 11] is a long-termed debate. The debate has been mainly centered around the inheritance scheme in frame systems and OODBs. Frames are allowed to override slot values and facets, but this can lead to cancellation of definitional properties [6].

In OODBs, class inheritance scheme has been treated in different ways, according to the kind of the properties being inherited. More specifically, a) slot values are not inherited at all, b) slot and method definitions are inherited and can be specialized, but not canceled, and c) method implementations can either be inherited, overridden or specialized.

According to this scheme, if constraints are represented and enforced through methods, then constraints defined at super-classes can be incorrectly overridden [9, 2]. Alternatively, constraints can be represented using active rules [9, 18] and enforced through an active rule interpreter. This approach is quite general and can cope with constraint interferences [18], but is not yet very efficient [8].

Recently, a hybrid approach to constraint representation and enforcement has been proposed [10, 2], where constraints are represented as distinct objects, for maintenance purposes, but they are enforced through a modified method execution mechanism. The advantages of this approach is that constraints are both represented explicitly and enforced efficiently. It must be noted here that we refer to passive constraints, i.e. constraints that prevent illegal database transitions and do not try to repair them.

Inheritance of constraints has been treated as obligatory, i.e. each constraint is inherited from every sub-class of the class that the constraint is defined. Some real-world applications require, though, that some objects should be excepted from obeying every constraint. Constraint exception has been treated in [5], in the context of the more general problem of modelling class hierarchies with contradictions and resolving multiple inheritance ambiguities. However, this approach does not treat constraints as first-class objects, but merely as slot facets.

The main contribution of this paper is an object-oriented scheme for representing constraints with emphasis on exceptions and an algorithm that decides at run-time the applicability of constraints to the attempted database transitions. We note, that it is the responsibility of the database designer to specify exceptions and there is no tool to automatically derive them.

## 3 Object-Oriented Constraint Representation

The object model we use throughout this paper consists of a set of objects `O`, such that each element in `O` can belong to one of three disjoint categories: instances (`I`), classes (`C`) and meta-classes (`M`). We will exclude meta-classes from our discussion, since classes and their instances are the most "useful" database objects. Objects of different categories are related through the `instance_of` relationship, subjected to the following restrictions:

$$(\forall X,Y)\ X\ \mathtt{instance\_of}\ Y \Rightarrow (X \in I \wedge Y \in C) \vee (X \in C \wedge Y \in M)$$

Furthermore, classes are organized in hierarchies through the `is_a` relationship:

$$(\forall X,Y)\ X\ \mathtt{is\_a}\ Y \Rightarrow X \in C \wedge Y \in C$$

Class `X` is called sub-class of `Y` and `Y` super-class of `X`. We will now define two relationships between objects, which are the transitive extensions of the `is_a` relationship:

$$(\forall X,Y)\ X < Y \Rightarrow (X\ \mathtt{is\_a}\ Y) \vee ((\exists Z)\ X\ \mathtt{is\_a}\ Z \wedge Z < Y)$$

$$(\forall X,Y)\ X \leq Y \Rightarrow (X = Y) \vee X < Y$$

The set of all super-classes of a class (including itself) is given from the following definition:

$$\mathrm{SUPER}(C) = \{\ X \mid C \leq X\ \}$$

The `is_a` (and consequently the ≤) relationship models inheritance and sub-typing in OODBs. If two objects are related with the `is_a` (or ≤) relationship, then two things hold:

- The sub-class inherits the structure (slots[2]) and behaviour (methods) of the super-class,
- The extension of the sub-class is a sub-set of the extension of the super-class:

  $(\forall X,Y)\ X \le Y \Rightarrow \mathrm{EXT}(X) \subseteq \mathrm{EXT}(Y)$

  where, EXT is the extension of a class, which is a set of all instances of a class and all its sub-classes:

  $\mathrm{EXT}(C) = \{ X \mid X\ \mathtt{instance\_of}\ Y\ \&\ Y \le C \}$

In the following we discuss a method for representing constraints in OODBs. We assume that constraints are associated with classes in order to be shared among the class instances. In [2] an object-oriented constraint representation scheme was presented, where constraints are represented by objects, instances of class `constraint`. However, the actual code that checks the constraints is stored within the classes that the constraint applies and is retrieved at run-time. Here this approach is extended in the following aspects:

- Constraint-objects are augmented to include the new information for exceptions, and
- A new constraint checking algorithm is proposed to resolve constraint applicability ambiguities at run-time.

In the following we present the augmented constraint representation. Each constraint-object is an instance of class `constraint`, with an object identifier `ConstOID`, and has the following structure:

- Slot `colan` (single valued[3] - mandatory[4]), which stores the declarative specification of the constraint.
- Slot `applies_to` (set valued - mandatory), which is a set of classes, for which the constraint applies. Note that the constraint applies to all sub-classes of the classes in `applies_to`, unless otherwise specified by the slot `except_class`. For this reason, the classes in `applies_to` must not be connectable through the hierarchy lattice:

  $\forall X,Y \in \mathtt{applies\_to}(\mathtt{ConstOID}) \Rightarrow \neg(\ X \le Y \lor X \ge Y\ )$

- Slot `except_class` (set valued - optional), which is a set of classes, for which the constraint does not apply, as an exception. The following restrictions hold for the classes in `except_class`:

  1. $\forall X,Y \in \mathtt{except\_class}(\mathtt{ConstOID}) \Rightarrow \neg(\ X \le Y \lor X \ge Y\ )$

     Classes in `except_class` must not be connectable through the hierarchy lattice.

---

[2]The terms “slot”, “attribute” and “instance variable” will be used interchangeably throughout the text.

[3]Single-valued slots can have at most one value stored, while set-valued slots can have many.

[4]Mandatory slots must have at least one value stored, while optional slots can have none.

2. $\forall$ X $\in$ except_class(ConstOID) $\Rightarrow$
X $\notin$ applies_to(ConstOID)

A class cannot be both applicable and exception to a constraint.

3. $\forall$ X $\in$ except_class(ConstOID) $\Rightarrow$
$\exists$ Y $\in$ applies_to(ConstOID) $\wedge$ X < Y

Every Class in except_class must be sub-class of a corresponding class in applies_to.

- Slot except_inst (set valued - optional), which is a set of instances, for which the constraint does not apply, as an exception. The following restrictions hold for the objects in except_inst:

1. $\forall$ X $\in$ except_inst(ConstOID) $\Rightarrow$
$\exists$ Y $\in$ applies_to(ConstOID) $\wedge$ X $\in$ EXT(Y)

All instances in except_inst must belong to the extension of one of the classes in applies_to.

2. $\forall$ X $\in$ except_inst(ConstOID) $\Rightarrow$
$\neg\exists$ Y $\in$ except_class(ConstOID) $\wedge$ X $\in$ EXT(Y)

All instances in except_inst must not belong to the extension of one of the classes in except_class, since this is redundant; they would be excepted any way.

However, there are cases where there is a constraint that at first is not applicable to only few instances of a certain class, but later during the database lifetime the exception is generalized to all instances of the class. To cater for this, we provide method generalize which deletes all such instances from the slot except_inst and adds the class in the slot except_class.

- Slot overrides (set valued - optional), which stores the object identifiers (OIDs) of constraints replaced by this one. At least one of the classes that the overriding constraint applies to, must be sub-class of one of the classes that the overridden constraints apply:

$\forall$ X $\in$ overrides(ConstOID) $\Rightarrow$
($\exists$ Y,Z) Y $\in$ applies_to(X) $\wedge$
Z $\in$ applies_to(ConstOID) $\wedge$ Z < Y

- Slot overridden_by (single valued - optional), which stores the OID of a constraint that replaces this one. The restriction that holds for the values of slot overrides, also holds here:

($\exists$ Y,Z) Y $\in$ applies_to(overridden_by(ConstOID)) $\wedge$
Z $\in$ applies_to(ConstOID) $\wedge$ Y < Z

Furthermore, slots overrides and overridden_by must be cross-referenced from the constraint-objects:

$\forall$ X $\in$ overrides(ConstOID) $\Rightarrow$ ConstOID = overriden_by(X)

ConstOID $\in$ overrides(overriden_by(ConstOID))

The last two slots that define constraint replacement impose further restrictions on the rest of the excepting slots, to avoid non-admissible situations, shown in fig. 2:

- $\forall$ X $\in$ except_inst(ConstOID) $\Rightarrow$
  ($\neg\exists$ C,D) X instance_of C $\wedge$
  D $\in$ applies_to(overridden_by(ConstOID)) $\wedge$
  C $\leq$ D

  This restriction prohibits instance exceptions of a constraint to belong to classes which are "controlled" by a second constraint that overrides the first one.

- $\forall$ X $\in$ except_class(ConstOID) $\Rightarrow$
  $\neg\exists$ C $\in$ applies_to(overridden_by(ConstOID)) $\wedge$ X $\leq$ C

  This restriction prohibits class exceptions of a constraint to be sub-classes of a class "controlled" by a second constraint that overrides the first one.

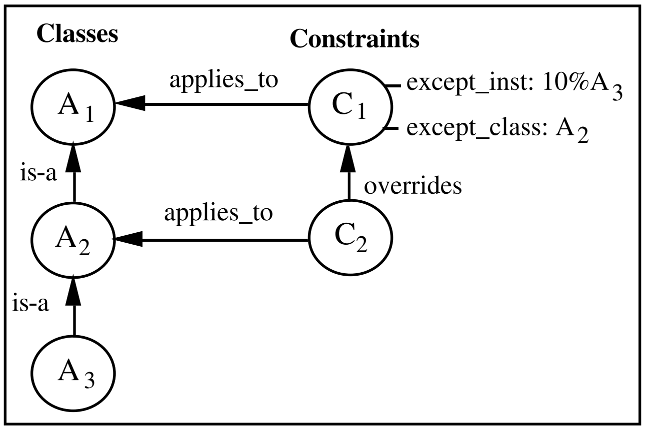


**Figure 2.** Example of incorrect use of exception slots

We will now explain the reason why both overrides and overridden_by slots are needed. If there is a constraint $C_2$ that overrides another one $C_1$ (fig. 3), then constraint $C_2$ may have class or instance exceptions. Therefore, these exceptions will not be guarded by $C_2$. Furthermore, they will not be guarded by $C_1$, either, since $C_1$ is overridden by $C_2$. In order to avoid having objects excepted from every constraint, we assume that when an overriding constraint is excepted for some objects, then the overridden constraint applies for these objects. This can only be achieved by keeping information in the overrides slot of the overriding constraint, to follow the link to the overridden constraint. Otherwise, one should search all constraint-objects, to find one that contains $C_2$ in its overridden_by slot, which is extremely inefficient.

On the other hand, the slot overridden_by is needed by the run-time constraint execution algorithm (appendix) because the order of constraint execution is bottom-up, thus a constraint of a super-class may have been overridden by an already executed constraint of a sub-class and its execution must be prohibited.

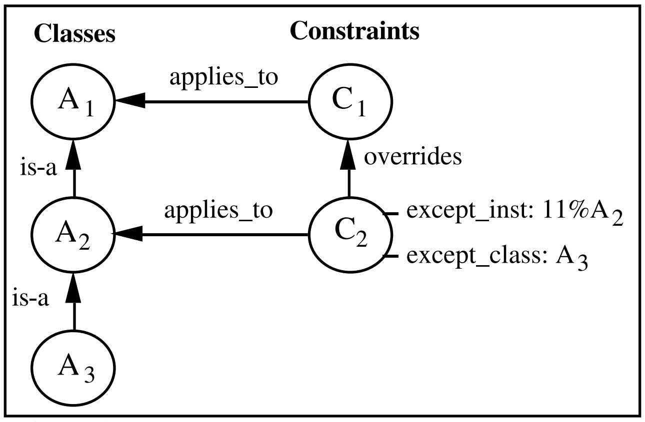


**Figure 3.** Example of excepting an overriding constraint

Furthermore, each class `Class` that is constrained, has the following slots (or class variables):

- Slot `constraints` (set valued - optional), which stores the OIDs of the constraints that apply to `Class`. These OIDs belong to constraints that have `Class` in their `applies_to` slot:

  $\forall$ X $\in$ constraints(Class) $\Rightarrow$ Class $\in$ applies_to(X)

- Slots `new_constraints`, `delete_constraints`, etc., which are optional, set valued slots that store the actual code for checking the corresponding constraints. Each element of these slots has the form:

  `code(ConstOID,Code),`

  where `ConstOID` is the OID of the constraint that corresponds to `Code`, and `Code` is the actual code for the corresponding method. We will not expand further here on the storage and maintenance of constraints. Details can be found in [1, 2].

The slots in the constraint-objects and the constrained classes have methods for accessing, inserting, deleting and modifying their values. These methods reflect the restrictions imposed for each slot. Therefore, the above restrictions are built into the constraint subsystem of the OODBMS. However, one might view the constraint-slot restrictions as constraints about the instances of class `constraint`. This provides a uniform view of the object base, but since we do not yet strive for modularity and extendibility of the constraint subsystem, such functionality is not desired.

## 4 Representing Constraints with Exceptions

In the light of the object-oriented constraint representation we proposed, the problematic constraint cases can now be re-stated and resolved. Our main means are the new slots, which can easily model exceptions of constraints. However, the constraint checking algorithm must be re-considered in order to cater for exceptions, but this will be presented in the next section.

**Constraint Contradiction.** Constraints $C_1$ and $C_2$ are defined at different branches of the hierarchy of the OODB schema (fig. 1) but they are both inherited by class `emy`, since the two branches re-join at this class. In order to resolve the ambiguity the database designer must explicitly declare which is preferable. However, it is not possible to state that one of the constraints overrides the other, because restrictions on the slots `overrides` and `overridden_by`, impose that the constraints are defined in classes of the same hierarchy branch.

On the other hand, if one decides which of the two constraints dominates, it is very easy to cancel the other for class `emy`. In table 1, it is shown that $C_1$ is disabled for class `emy`, therefore $C_2$ dominates. Note that the slot `colan` stores the textual expression of constraints, as they appeared in section 2.

| | |
|---|---|
| constOID: $C_1$ | constOID: $C_2$ |
| colan: "forall ..." | colan: "forall ..." |
| applies_to: {staff} | applies_to: {student} |
| except_class: {emy} | except_class: {} |
| except_inst: {} | except_inst: {} |
| overrides: {} | overrides: {} |
| overridden_by: nil | overridden_by: nil |

**Table 1.** Resolving constraint contradiction

**Constraint Subsumption.** Constraints $C_3$ and $C_4$ are again defined at different branches of the hierarchy, but they "meet" at class `emy`. The problem is that $C_3$ consistently dominates over $C_4$ (subsumption) and the ambiguity of which constraint applies should be resolved by the designer. Since, neither constraint can override the other, as explained, one of the constraints should be excepted from class `emy`. If we decide that $C_3$ is the applicable constraint, then $C_4$ should be excepted for efficiency reasons. If, on the other hand, constraint $C_4$ is to be enforced, then $C_3$ must be excepted, because the latter subsumes the former. The exception of either constraint can be done by inserting the value `emy` in the slot `except_class`, as shown before.

**Constraint Cancellation.** Constraint $C_5$ applies to class `staff` and its sub-classes, except class `academic` (and its sub-classes). Therefore, the designer must exclude the latter class, by inserting its name into the slot `except_class` of constraint $C_5$. Note that in this case we do not resolve any ambiguity between competing constraints, but we merely "delete" a constraint from the "guards" of a class, without replacing it with another one.

**Constraint Specialization.** Constraints $C_6$ and $C_7$ are defined on the same branch of the class hierarchy and are both inherited by class `student` and its subclasses. However, only the latter should be checked, because it is more restrictive.

Checking $C_6$, as well, is not wrong, but is inefficient. Therefore, constraint $C_6$ should be replaced by $C_7$ at class `student` and further. Table 2 shows how this can be achieved, using slots `overrides` and `overridden_by` of both constraints.

| | |
|---|---|
| constOID: $C_6$ | constOID: $C_7$ |
| colan: "forall ..." | colan: "forall ..." |
| applies_to: {person} | applies_to: {student} |
| except_class: {} | except_class: {} |
| except_inst: {} | except_inst: {} |
| overrides: {} | overrides: {$C_6$} |
| overridden_by: $C_7$ | overridden_by: nil |

**Table 2.** Representing constraint specialization

Note that constraints are cross-referenced. However, the designer is not responsible for maintaining both references, since methods for inserting, deleting or updating these slots are restricted by referential integrity constraints, as explained in previous sections. The designer modifies just one (either) of the constraints and the other is automatically updated.

**Constraint Overriding.** Constraints $C_8$ and $C_9$ are defined on the same branch of the class hierarchy and are both inherited to class `msc`. However, only the latter should be enforced, because it overrides the former. Enforcing $C_8$, as well, is wrong, since no updates on `msc` class would be allowed. Therefore, constraint $C_8$ must be replaced by $C_9$ at class `msc`. This can be achieved, using slots `overrides` and `overridden_by` of both constraints, as shown before.

**Instance Exceptions.** Constraint $C_{10}$ applies to all students, except student `23%undergraduate`, who is disabled. Constraint exceptions for one or more distinct instances can be practically represented by inserting the OIDs of those instances in the slot `except_inst` of the constraint, as table 3 shows.

| |
|---|
| constOID: $C_{10}$ |
| colan: "forall ..." |
| applies_to: {student} |
| except_class: {} |
| except_inst: {23%undergraduate} |
| overrides: {} |
| overridden_by: nil |

**Table 3.** Representing constraint exception for instances

## 5 The Algorithm for Checking Constraints with Exceptions

The run-time algorithm for checking constraints must be extended to take into account the new functionality introduced by the exception slots. Since message recipients and methods are resolved dynamically at run-time in object-oriented systems, the same truth holds for constraints. Only at run-time the system can decide if a constraint that "seems" applicable, should be really checked or ignored. In the following we present:

- the algorithm CHECK that ensures that every applicable constraint of every super-class of the class, that the message recipient object belongs to, is considered for execution, and
- the algorithm GET_CODE that checks whether the current constraint should be excluded from execution or not and returns the corresponding code for actual constraint checking.

The algorithm CHECK is the top-level procedure that receives, analyses the update message and obtains the following information:

- The message recipient object,
- The class of the above object,
- The code-slot of the above class.

Then, the algorithm iterates over all super-classes of the above class (including this one) and all the constraints that apply to each of these classes. For each of these constraints, its applicability is checked with the GET_CODE algorithm and a piece of code is returned, in the following circumstances:

- The constraint is applicable and its code is returned.
- The constraint overrides some constraint defined higher in the hierarchy, but the recipient object or its class is an exception. The code of the overridden constraint is returned instead.
- The constraint is not applicable, therefore no code is returned.

If a piece of code is returned, then it is executed, otherwise the next iteration occurs. Note that the code consists of Prolog goals enhanced by Prolog-like OODB messages [1, 2]. If the execution is successful, then iteration proceeds normally, otherwise the violation of the constraint cause the rejection of the update. When all constraints are satisfied, the default method execution scheme is evoked. The full listing of the algorithm is presented in the appendix.

The order that the constraints are retrieved and checked is bottom-up among the class of the recipient object and its super-classes, and arbitrary within a class. Since constraints do not perform any updates on the database, their order of execution is not important. However, we believe that bottom-up execution should be more efficient because constraints defined at lower classes of the hierarchy tend to be more restrictive and more likely to fail. Therefore if they are checked and fail early, no time is spent on checking many constraints unnecessarily.

## 6 Conclusions

In this paper we have presented a scheme and a methodology for modelling constraints with exceptions in object-oriented databases. The representation scheme is based on the notion of constraints as first-class objects, explicitly defined, stored and maintained in an OODB. The constraint class descriptor is augmented with slots that represent object exceptions and/or constraint replacement.

This extra information can be used to model a variety of exceptional situations, like constraint contradiction, subsumption, cancellation, specialization, overriding and instance exception. It is demonstrated, through examples, the easiness that this modelling can be achieved and maintained through simple modifications of the constraint slots.

Finally, the constraint exception functionality is supported by a constraint checking algorithm that resolves at run-time the ambiguity whether a given constraint is applicable to a given update message, taking into account the constraint exception information. Furthermore, the restrictions that hold between the information stored in the exception slots are enforced through modified slot-accessing methods of constraint-objects.

The constraint exception modelling scheme we proposed is aimed at the database designer and not the end-user. Therefore, its manual nature is not a great drawback, since it will not be used by ignorant people. However, our tool would be greatly benefited from an automatic or semi-automatic analysis of constraint consistency [14]. The analysis will decide whether a newly inserted constraint is consistent with the rest of constraints and possibly suggest ways that this can be avoided by declaring some constraints or classes as exceptions. Previous work on constraint consistency is based on a logical formulation of both the constraint expressions and the resolution mechanism [7, 17].

Future work also includes the amalgamation of the constraint modelling scheme and the corresponding algorithm with the parallel constraint checking algorithm presented in [4], in order to achieve fast constraint checking performance, through the reduced number of constraints to be checked (due to exceptions) and the parallel execution on a multi-processor object-oriented database system, named PRACTIC [3].

### Acknowledgments

The first author is supported by a scholarship from the Greek State Scholarships Foundation (I.K.Y.).

## References

1. N. Bassiliades, Constraint description in ADAM, M.Sc. Thesis, University of Aberdeen, 1992.
2. N. Bassiliades and P.M.D. Gray, CoLan: A functional constraint language and its implementation, forthcoming in *Data & Knowledge Engineering*.
3. N. Bassiliades and I. Vlahavas, PRACTIC: A concurrent object data model for a parallel object-oriented database system, submitted for publication, 1993.
4. N. Bassiliades and I. Vlahavas, Constraint checking in a parallel object-oriented database system, to appear in *Parallel Algorithms and Applications*, 5(3+4).
5. A. Borgida, Modelling class hierarchies with contradictions, *Proc. ACM SIGMOD '88*, 434-443.
6. R.J. Brachman, I lied about the trees or, defaults and definitions in knowledge representation, *AI Magazine*, 6(3), 60-93.

7. F. Bry and R. Manthey, Checking consistency of database constraints: A logical basis, *Proc. 12th VLDB*, Kyoto, Japan, 1986.
8. O.G. Diaz, N.W. Paton and P.M.D. Gray, Rule management in object-oriented databases: A uniform approach, *Proc. 17th VLDB,* Barcelona, Spain, 1991, 317-326.
9. O.G. Diaz and S.M. Embury, Generating active rules from high-level specifications, *Advanced Database Systems - Proc. 10th BNCOD*, Aberdeen, Scotland, Springer-Verlag, 1992, 227-243.
10. S.M. Embury, P.M.D. Gray and N. Bassiliades, Constraint maintenance using generated methods in the P/FDM object-oriented database, *Proc. 1st Int. Workshop on Rules in Database Systems,* Edinburgh, Scotland, 1993, 364-381.
11. C. Esculier, Inheritances with exceptions: An approach based on semantic tolerance, *AI in Databases and Information Systems (DS-3)*, North-Holland, 1990, 99-111.
12. P.M.D. Gray, D.S. Moffat and N.W. Paton, A prolog interface to a functional data model database, in: J.W. Schmidt, S. Ceri and M. Missikoff, eds., *Advances in Database Technology - Proc. 1st Conf. on Extending Data Base Technology,* Venice, Italy*,* Springer-Verlag, 1988, 34-48.
13. P.M.D. Gray, K.G. Kulkarni and N.W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach*, Prentice Hall, 1992.
14. P.W.P.J. Grefen and P.M.G. Apers, Integrity control in relational database systems - An overview, *Data & Knowledge Engineering,* 10(2), 1993, 187-223.
15. H.V. Jagadish and X. Qian, Integrity maintenance in an object-oriented database, *Proc. 18th VLDB,* Vancouver, Canada, 1992, 469-480.
16. R.A. Kowalski and F. Sadri, Logic programs with exceptions, *Proc. 7th ICLP*, MIT Press, 1990, 599-613.
17. H. Noble and T. Abbod, Meta-rules and semantic integrity constraints in databases, *Proc. 5th BNCOD*, Canterbury, UK, 1986.
18. S.D. Urban, A.P. Karadimce and R.B. Nannapaneni, The implementation and evaluation of integrity maintenance rules in an object-oriented database, *Proc. IEEE Int. Conf. on Data Engineering*, 1992.

## Appendix - The Constraint Checking Algorithm

Listing of the algorithms described in section 5.

### Algorithm CHECK

*Global Variables*
    *X,C: class_name*
    *OID: oid*
    *S: slot_name*
    *Y: const_oid*
    *Pro: set of const_oid*
*Local Variables*
    *Code: oodb_program*

```
When the following message is issued:
            <update_message> => OID
      do the following:
            Set C, such that OID instance_of C
            Set S, such that <update_message> corresponds to one of the slots
            of class C that store constraint checking code.
Set Pro to {}
For each X ∈ SUPER(C), do:
      For each Y, such that Y ∈ constraints(X) ∧ code(Y,_) ∈ S(X),
      do:
            Call GET_CODE
            If GET_CODE returns Code<>nil then:
                  Execute Code
                  If Code fails then:
                        Inform user about constraint violation
                        Reject the <update_message>
                        End CHECK
Evoke the default method execution mechanism for <update_message>
```

**Algorithm GET_CODE**

```
Local Variables
      Sp,Q: class_name
      Constr: oodb_program
If X = C then:
      If OID ∈ except_inst(Y) then:
            Set O, such that O ∈ overrides(Y) ∧
                                          ∃ Sp ∈ applies_to(O) ∧ Sp > X
            Set Pro to Pro ∪ {O}
            Return Constr, such that code(O,Constr) ∈ S(Sp)
      Else return Constr, such that code(Y,Constr) ∈ S(X)
Else:
      If Y ∈ Pro then:
            Set Pro to Pro − {Y}
            Return nil
      Else:
            If overridden_by(Y) ≠ nil then:
                  Set Q, such that Q ∈ applies_to(overridden_by(Y))
                                                            ∧ Q < X
                  If Q ≥ C then return nil
            If OID ∈ except_inst(Y) ∨ C ∈ except_class(Y) then:
                Set O, such that O ∈ overrides(Y) ∧
                                          ∃ Sp ∈ applies_to(O) ∧ Sp > X
                Set Pro to Pro ∪ {O}
                Return Constr, such that code(O,Constr) ∈ S(Sp)
            Else return Constr, such that code(Y,Constr) ∈ S(X)
```