Consistency Management in Object-Oriented Databases*

Hussien Oakasha, Stefan Conrad, and Gunter Saake

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, 39016 Magdeburg, Germany
e-mail: {oakasha,conrad,saake}@iti.cs.uni-magdeburg.de

Abstract. The paper presents concepts and ideas underlying an approach for consistency management in object oriented databases. In this approach constraints are structured as first class citizens and stored in a meta-database called constraints catalog. When an object is created constraints of this object are retrieved from the constraints catalog and relationships between these constraints and the object are established. The structure of constraints has several features that enhance consistency management in OODBMS which do not exist in conventional approaches in a satisfactory way. These features are monitoring objects consistency at different levels of update granularity, integrity independence, efficiency of constraints maintenance, controlling inconsistent objects, enabling and disabling of constraints globally to all objects of database as well as locally to individual objects, and the possibility of declaring constraints on individual objects. All these features are provided by means of basic notations of OO data models.

1 Introduction

The integrity aspect of OO data models underlying the mainstay OODBMSs today is generally weak although it is improved over the one of the pure relational data model. Apart from constraints like domain constraints, cardinality constraints, referential constraints and key constraints, other complex types of state constraints like inter-constraints [JQ92] are specified and maintained either by encoding them in application programs or methods of classes [BS97,BS96] (the so-called application-oriented techniques) or using event-condition-action (ECA) rules facilities of these OODBMS [BMP91,WC96].

Disadvantages of application-oriented techniques are redundancy, i.e. a constraint must be specified in every transaction that might violate it, and understandability for semantics of constraints since they are encoded in statements of a general-purpose programming language. Another disadvantage is that modifying constraints is a hard task. This is due to the fact that modifying constraints take place manually by users and through modifying transactions and application programs. Finally features like handling objects inconsistency and disabling and enabling of constraints are absent.

A possible approach to avoid some problems of application-oriented techniques like lacking possibilities of disabling and enabling of constraints is to maintain integrity using ECA rules [Día92,BMP91]. However, modifying constraints is still a problem. First, apart from general problems like confluence and termination, the specification of constraints is in general not mapped one-to-one to an ECA rule: a given constraint is represented by many ECA rules with different events and hence different actions. Second, events of ECA rules are considered to be calling of methods of classes which have side effects that may violate the condition part of a rule. Thus, ECA rules will be sensitive to modification of class methods. In addition, to modify a constraint users are required to determine which ECA rules represent the constraint. If it is determined that an ECA rule is relevant to the modified constraint then the user must modify this rule accordingly. As there is no overall control on the process of modification, this increases the possibility of inconsistencies; both by modifying rules which are irrelevant to the modified constraints and missing rules which should be modified.

* Supported by a PhD scholarship from the Federal State Sachsen-Anhalt, and ESPRIT Working Groups ASPIRE and FIREworks

We believe that a consistency management subsystem (CMS) for OODBMSs should have the following features which are not provided by either techniques mentioned above:

- *Object-orientation.* CMS should work with basic principles of object-orientation such as inheritance and encapsulation.
- *Specification.* Constraints should be specified declaratively and structured as first class citizens.
- *Interrelated Objects.* CMS should be able to maintain consistency of a large number of interrelated objects with complex structures.
- *Transactions.* CMS should work with advanced types of transactions such as interactive and long duration transactions.
- *Efficiency.* The constraint checking should rely on optimization techniques for improving constraint evaluation.
- *Integrity Independence.* CMS should be capable to change constraint specifications without changing application programs and update transactions.
- *Inconsistency.* CMS should control inconsistency of objects.
- *Persistence Style.* CMS should maintain consistency of all objects, that is persistent and non-persistent ones, and regardless of the persistence style of OODBMSs.
- *Disabling and Enabling of Constraints.* CMS should provide the tools for enabling and disabling constraints at various levels of abstraction like the whole database, or a class or a specific object.
- *Update Granularity.* CMS should work with different levels of update granularity of objects such as updating a simple attribute or a complex attribute or the whole state of an object.

In this paper we propose an approach for consistency management in OODBMS that support the features listed above. Main aspects of our approach are the *constraints catalog* and a novel technique for constraints structuring. The constraints catalog is a meta-database acting as repository of constraints specifications. The main purpose of the constraints catalog is to separate the constraints specification from transactions and application programs and hence to provide the feature of integrity independence to our approach. We consider constraints as first class citizens. To provide the other features to our approach constraints are structured as an aggregation of two parts. The first part is called the *kernel* which is an object that is sharable among interrelated objects that are subject to the same constraint. The second part is called the *shell* which is an object being sharable among objects of the same class.

The remainder of the paper is organized as follows: Section 2 presents the notations and definitions that will be used throughout this paper. In Section 3 we present the basic idea of the optimization technique for constraints evaluation that will be used in our approach. The structure of constraints catalog is described in detail in Section 4. In Section 5 we first motivate and then define the constraint structure. Finally in Section 6, we discuss the issue of consistency maintenance by using our approach.

2 Basic Notation and Definitions

2.1 Object-oriented Data Model

In this paper we use a generic OO data model that has basic features of the OODBMS manifesto presented in [ABD+89]. We assume that the inverse relationship is maintained by the model automatically. Examples of such a model are O_2 [BDK92] and ODMG release 1.2 [Cat96]. Here we follow the notations of O_2. In addition we consider transactions as objects of the class $Transaction$ and there is an attribute of the class $Transaction$ named $updatedObjects$ of set-structured type $\{Any\}$, where Any is the name of the root class of all class hierarchies.

In the following we draw examples for constraints from an example database given in [Cat96]. The schema of this example database is shown in Figure 1. Each rectangle represents a class, thin arrows represent relationships between classes and thick arrows indicate isa-relationships in class hierarchies.

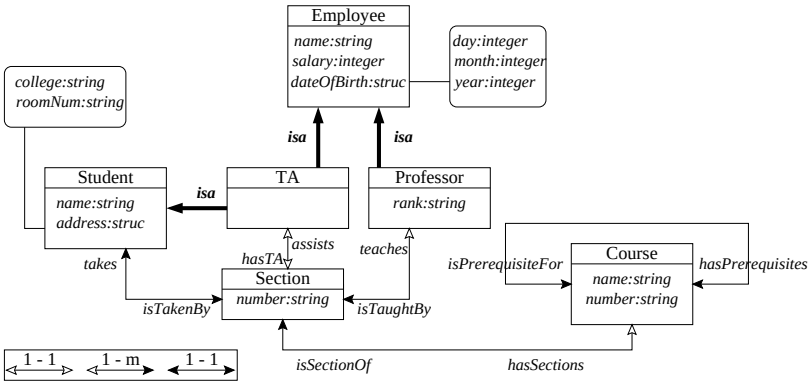

Fig. 1. Example Database Schema

2.2 Constraints

Part of our approach is to build a meta-database in which we will store some knowledge about constraints. The meta-database is used for consistency maintenance of objects as well as constraints manipulations. Among other knowledge, we store names of classes that are subject to a constraint and the path expressions that may violate the constraint. To easily obtain this information from the logical specification of constraints we define a canonical form for constraints. Before doing that we need the definition of *intra-paths*.

Definition 1. (Intra-Paths) An *intra-path* is a path expression $A_1 \ldots . A_n$ such that either $n = 1$; or $n > 1$ and for every $i \in [1, n]$ A_i is not a reference attribute[1] for a class. ■

Examples of intra-paths for class *Student* are the path expressions *takes* and *address.college* respectively. In this paper we will assume that the object state is modified only by means of intra-paths of the object. For instance, under this assumption we cannot modify a professor's salary by the path expression $se.isTaughtBy.salary \leftarrow 55$, where se is an object of class *Section* and $isTaughtBy$ is a reference attribute for class $Professor$.

A constraint is in *canonical form* if each path mentioned in it is an intra-path and each quantified object variable is running over the extension of its class. In the following definition we define canonical constraints.

Definition 2. (Canonical Constraints) A *canonical form* of a constraint is a well-formed formula (wff) in prenex normal form $(Q_1 o_1 \in E_1) \ldots (Q_n o_n \in E_n) M[o_1, \ldots o_n]$. Where $Q_i \in \{\exists, \forall\}$, E_i is an extension of a class, and M is a quantifier free formula in which all path expressions are intra-path expression.
We denote the quantifier structure $(Q_1 o_1 \in E_1) \ldots (Q_n o_n \in E_n)$ of a constraint W as $\mathbf{Q}_W$ and the matrix $M[o_1, \ldots o_n]$ as $\mathbf{M}_W$. ■

Example 3. Consider the following example constraint which specifies that "*each professor teaches at least one section belonging to a course which has no prerequisite courses*". The following two wffs are logical specifications for the constraint:

$$W_0 : (\forall pr \in Professors)(\exists se \in pr.teaches)(se.isSectionOf.hasPrerequisites = \emptyset).$$
$$W_1 : (\forall pr \in Professors)(\exists se \in Sections)(\forall co \in Courses)$$
$$(se \in pr.teaches) \wedge (co = se.isSectionOf) \Rightarrow (co.hasPrerequisites = \emptyset).$$

According to the definition of canonical constraints, W_0 is not a canonical specification because of the presence of the quantification $(\exists se \in pr.teaches)$ and the non intra-path expression:

$$(se.isSectionOf.hasPrerequisites).$$

[1] An attribute A is a reference attribute for a class C if the type of A is either C or $\{C\}$.

The wff W_1 is a canonical specification for constraint W_0. Note that in W_0 objects of class *Section* and *Course* are implicitly referenced by $(\exists se \in pr.teaches)$ and $(se.isSectionOf.hasPrerequisites)$ respectively, whereas in W_1 they are explicitly referenced. □

2.3 Constraint Characteristics

Now we will define criteria in order to detect situations in which a given constraint could be violated. A constraint can be violated if any of the paths explicitly stated in the constraint appears as sub-path of the left hand side of an assignment operator '$\leftarrow$'. For instance if there is a constraint on the date of birth of an employee,

$$W_2 : (\forall Em \in Employees)(Em.dateOfBirth \geq v)$$

and one of the updates

$$Em'.dateOfBirth.day \leftarrow d, \text{or}$$
$$Em'.dateOfBirth.month \leftarrow m, \text{or}$$
$$Em'.dateOfBirth \leftarrow [dd, mm, yy]$$

occurs in an update unit then W_2 can be violated by this update. The common *characteristic* for these paths and the one appearing in constraint W_2 is the path $dateOfBirth$. The following definition describes the characteristic of constraints w.r.t. classes of the database schema.

Definition 4. (Constraint Characteristic) Let W be a constraint in the canonical form. The characteristic of the constraint W w.r.t. class C, denoted by $\chi_{(W,C)}$, is the set of all paths p such that there is an object variable o in W of type C and the path $o.p$ occurs in the matrix of the constraint W.

$$\chi_{(W,C)} = \{p \mid \text{there is } (Qo \in C) \text{ in } \mathbf{Q}_W \text{ and } o.p \text{ occurs in } \mathbf{M}_W\}$$

■

Example 5. Consider the constraint W_1 of Example 3. The characteristic of the constraint W_1 w.r.t classes *Professor*, *Section* and *Course* are respectively:

$$\chi_{(W,Professor)} = \{teaches\}$$
$$\chi_{(W,Section)} = \{isSectionOf\}$$
$$\chi_{(W,Course)} = \{hasPrerequisites\}$$

□

A similar criterion like the one we described in Definition 4 has also been defined in [BLR92] called *characterizations of constraints*. The definition of characterizations of constraints is more complex than the one we presented here. In the definition of [BLR92], extensions and names of classes are considered as a part of characterizations of constraints. For instance, according to the criterion of [BLR92], characterizations of constraint W_2 are the class extension *Employees* and the path expression $Employee.dateOfBirth$.

3 Simplified Forms

There is of course a trade-off in using canonical specifications of constraints instead of the original forms. This trade-off is between explicitly stating classes that are subject to the constraint, and hence introducing new quantified variables, and the evaluation of the constraint. The evaluation of canonical forms is generally inefficient and incomparable to the original form of the constraint.

A naive evaluation of the canonical specification W_1 of Example 3 using application-oriented integrity maintenance leads to three nested loops over objects of classes *Professor*, *Section*, and *Course*. If the sizes of extensions of these classes are n, m, and l (resp.), then the time complexity will be $O(n \times m \times l)$. In contrast, the complexity time for evaluating the constraint specification W_0 of Example 3 is $O(n \times k)$, where k is the average size of the set expression *pr.teaches*. Taking into account that $k \leq m$ the evaluation of W_0 is of course better that the general form. However, once a canonical specification of a constraint is obtained and all classes being subject to the constraint are identified, we can avoid evaluating canonical specifications by doing the following:

– first, we can derive a set of intra constraints such that each of them can be associated with one class which is subject to the constraint, and then
– optimize each of these intra constraints by using the specialization technique [Nic82].

We can do the first step by adapting the technique proposed in [JQ92] to our definition of canonical constraints. Here, we do not present all details of such an adaptation but we give an example for intra-constraints of constraint W_1.

Example 6. **(Intra Constraints)** Consider the canonical form of Example 3:

$$W_1 : (\forall pr \in Professors)(\exists se \in Sections)(\forall co \in Courses)$$
$$(se \in pr.teaches) \wedge (co = se.isSectionOf) \Rightarrow (co.hasPrerequisites = \emptyset)$$

Each of the following three wffs is equivalent to W_1.

$$W_{Pr} : (\forall pr \in Professors)(\exists se \in Pr.teaches)$$
$$(se.isSectionOf.hasPrerequisites = \emptyset).$$
$$W_{Se} : (\forall se \in Sections)(\exists se' \in se.isTaughtBy.teaches)$$
$$(se'.isSectionOf.hasPrerequisites = \emptyset).$$
$$W_{Co} : (\forall co \in Courses)(\forall se \in co.hasSection)(\exists se' \in se.isTaughtBy.teaches)$$
$$(se'.isSectionOf.hasPrerequisites = \emptyset).$$

Each of these intra-constraints is obtained according to an occurrence of an object variable in the quantifier structure of W_1. The intra-constraint W_{Pr} is obtained from W_1 according to the object variable pr. This is done by fixing object variable pr and restricting all other object variables mentioned in W_1 to those which have a relationship with pr and such that this relationship is mentioned in the matrix of W_1. For instance, not all objects of class *Section* are referenced in W_{Pr}, only those that can be reached by set expression $Pr.teaches$. In such a transformation some pre-valued terms may appear in the body of the obtained intra-constraint and hence they can be removed. For more details we refer to [JQ92]. Note that these intra-constraints can be further optimized using the specialization technique for improving integrity checking. For instance, the intra-constraint W_{Pr} can be specialized to objects of the class *Professor* by removing the universal quantifier $(\forall pr \in Professors)$ and replace pr by the parameter $self$ where $self$ refers to an object of class *Professor*.

$$W'_{Pr} : (\exists se \in self.teaches)(se.isSectionOf.hasPrerequisites = \emptyset).$$

In the rest of the paper we use the unprimed (resp. primed) notation to refer to the canonical (resp. optimized) form of a constraint. □

4 Constraints Catalog

It is essential for any consistency management approach to provide the feature of integrity independence [OS99,OS98], that is the ability to change constraints specifications without changing the application programs and transactions and vice versa. To provide this feature for our approach we must separate constraint specifications from transactions and class methods. For that, we use a meta-database called constraints catalog which is a repository for all information about constraints of a database.

The information stored in the constraints catalog is divided into two categories. The first category is about canonical specification of constraints. This kind of information will be stored in a class named IC. The second category of information describes simplified forms of constraints. This type of information will be stored in a class named Shell. The structure of classes IC and Shell is presented in detail in Subsection 4.1 and Subsection 4.2 respectively.

4.1 The IC Class

IC Class is one of two classes of the integrity catalog. The structure of the class IC is shown in Figure 2. An object of IC stores a complete specification of an integrity constraint defined for the database. For each constraint W there is an object o_W in the extension of IC. The object o_W stores the following information about the constraint W.

Class IC [
name : *string*,
classes : {*string*},
status : *string*,
mode : *string*,
wff : *string*,
shells : {Shell}]

Class Shell [
constraint : IC,
class : *string*,
form : *string*,
paths : {*string*}]

Fig. 2. Structures of Classes IC and Shell of the Constraints Catalog

Constraint Name. For querying and manipulating constraints there must be a way to identify them. Therefore, each constraint must have a unique name. The attribute *name* of type *string* holds the name of the constraint. For object o_{W_1} the value of *name* is W_1.

Restricted Classes. A constraint restricts objects of one or more classes. To associate a constraint with restricted classes appropriately, names of restricted classes must be stored as part of constraint specifications. The attribute *class* of set-structured type {*string*} holds names of classes that are subject to the constraint. For object o_{W_1} the value of attribute *classes* are all classes that appear in the quantifier structure of W_1. In our example these are the classes $Professor$, $Section$, and $Course$. Furthermore, any sub-class C of these classes in the class hierarchy must be added to *classes*. Since there is no such sub-class C in our example database, the value of *classes* is the set $\{Professor, Section, Course\}$. But for constraint W_2 of Subsection 2.3, the value of $o_{W_2}.classes$ is $\{Employee, TA, Professor\}$.

Status of Constraint. By the status of a constraint we mean whether the constraint is enabled or disabled to objects of *classes*. If a constraint is enabled then every object of a class in *classes* must obey the constraint. Otherwise the constraint is considered as if it does not exist. The attribute *status* of type *string* holds the status of the constraint. The initial value of *status* is *enabled*.

Constraint mode. The mode of a constraint specifies when a constraint should be checked. There are two possible modes for constraints. The first mode, called *immediate*, corresponds to constraints where checking should be done immediately after an update that might violate them. The mode *deferred* corresponds to constraints where checking should be deferred until the update unit commits. The attribute *mode* of type *string* hold the mode of the constraint. For object o_{W_1}, the value of *mode* is *deferred*. In general all constraints have *deferred* mode except for the domain constraints which have *immediate* mode. Thus for constraint W_2, $o_{W_2}.mode$ is *immediate*.

Logical Specification. The logical specification of the constraint is necessary for obtaining the information mentioned before as well as the simplified forms for the constraint. Thus logical specifications of constraints should be added to the information stored in the constraints catalog. The attribute *wff* of type *string* holds the logical specification of the constraint. For object o_{W_1}, the value of *wff* will be the canonical form of the constraint W_1.

Constraint Shells. An important requirement for consistency maintenance is the efficiency of constraint checking. Constraints shells are objects of a class named Shell that stores all information concerning constraints simplified forms. The description of the structure of the class Shell is the topic of Subsection 4.2.

4.2 Constraint Shells

The second class of the constraints catalog is the Shell class. The structure of the class Shell is given in Figure 2. Objects of Shell hold information that is necessary for efficient consistency maintenance. For each constraint W and object variable o of a class C that occurs in W there is an object S_o in the extension of Shell called the shell of the constraint W w.r.t. class C. The shell S_o stores the following information about the constraint W.

name	*classes*	*status*	*mode*	*wff*	*shells*
W_1	{*Professor*, *Section*, *Course*}	*enabled*	*deferred*	W_1	{#50, #51, #52}

(a) An Object of Class IC with id #25

constraint	*class*	*form*	*paths*
#25	{*Professor*}	W'_{Pr}	{*teaches*}
#25	{*Section*}	W'_{Co}	{*isSectionOf*}
#25	{*Course*}	W'_{Se}	{*hasPrerequisites*}

(b) Objects of Class Shell with ids #50, #51, #52

Fig. 3. (a) The object represents constraint W_1 of Example 3. (b) The three objects represent shells S_{Pr}, S_{Co}, and S_{Se} of constraint W_1 of Example 3

Simplified form. To improve integrity checking, we check an optimized form of the constraint rather than its canonical form. The attribute *form* of type *string* holds the simplified form of the constraint. For shell S_{Pr}, the value of the attribute *form* is the simplified form of W w.r.t. class *Professor*, that is wff W'_{Pr} of Example 6. Of course, at the implementation stage, wff W'_{Pr} must be translated to the query language of the underlying OODBMS.

Constraint Characteristic. To filter constraints that might be violated by an update from those which cannot be violated, the constraint characteristic is included in the constraint shells. The attribute *paths* of set-structured type {*string*} holds the constraint characteristic. For object S_{Pr}, the value of *paths* is the characteristic of constraint W w.r.t. class *Professor*, that is {*teaches*}.

Constraint. In general a constraint has many shells, each one corresponds to an object variable occurring in the constraint. Thus, there is a one-to-many relationship from IC to Shell. This relationship is represented by two reference attributes. The first attribute exists in the class IC, named *Shells*, and has the type {Shell}. The second attribute exists in the class Shell, named *constraint*, and is of type IC. For the shell S_{Pr}, the value of attribute *constraint* is the object o_{W_1} of IC that contains the specification of the constraint W_1.

Class. Each object of Shell holds the information of a constraint with respect to one of the subjected classes to the constraint. Thus the attribute *class* of type {*string*} holds the name of the class for which the further information of the shell is derived as well as any sub-class C of it. Since no sub-class of the class *Professor* exists in our example database, the value of *class* for object S_{Pr} only contains the class name *Professor*, i.e., $S_{Pr}.class = \{Professor\}$.

The complete representation of shells of constraint W_1 is given in (b) of Figure 3.

5 Constraint Structure = Shell + Kernel

In our approach we associate each object o of a class C with objects stored in the constraints catalog and which represent constraints that restrict the class C. Thus we assume that the root class *Any* has an attribute named *constraints*. This means that each class in the database schema inherits this attribute. In this section we define the type definition of the attribute *constraints* but first we motivate this definition by an example.

5.1 Motivation

Objects of Shell have a complete description of all integrity constraints as well as of all information that is necessary for checking them efficiently. Thus we may define the type of attribute *constraints* of class *Any* to be a set-structured type {shell}. This means that we associate each object with shells of constraints that restrict these objects. This, of course, must be done at the time of creating these objects. Consider the three objects Pr, Co, and Se of class *Professor*, *Course*, and *Section* (resp.). Assume that these classes are restricted only by the constraint W_1. The specification of this constraint and its shells are shown in Figure 3. We can associate them with each of the shells of constraints W_1 by the following general statements:

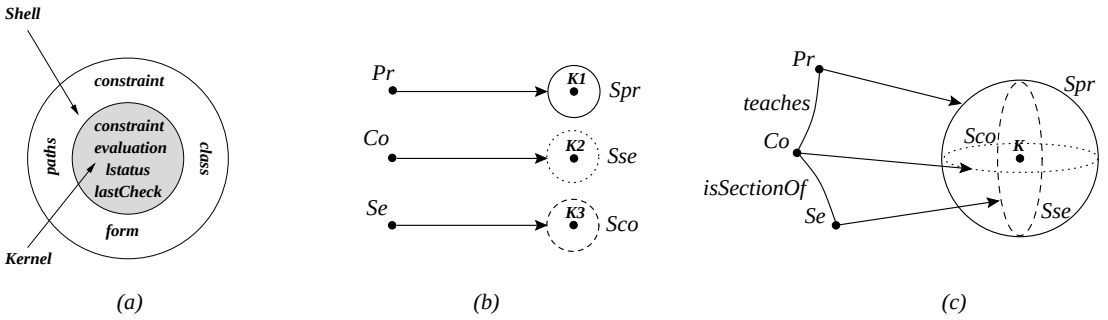

Fig. 4. (a) A Constraint Structure = Shell + Kernel. (b) Independent objects have different shells and kernels. (c) Interrelated objects have three different shells, but only one kernel.

$$\begin{array}{ll} \textbf{select} & s \\ \textbf{from} & \mathsf{Shell}\ \ s \\ \textbf{where} & s.class = type(o) \end{array}$$

In this statement, $o \in \{Pr, Co, Se\}$ and $type(o)$ is the type of object o which in this case is the name of the class to which the object belongs. Since types of these objects are different the shells associated with them, S_{Pr}, S_{Co}, and S_{Se}, are different, too. This situation is depicted in (b) of Figure 4 (ignore for a while symbols K_1, K_2, and K_3).

A drawback of this representation is that it does not take into account that the three objects Pr, Co, and Se may become interrelated objects and, hence, they should share the evaluation of the constraint W_1. In this representation, if three objects are modified and their modification violates the constraint W_1, then at commit time three different simplified forms of the constraint W_1 are evaluated. As the three simplified forms are equivalent to W_1, evaluating only one of them is sufficient. In other words there is no connection between the three different shells of Pr, Co, and Se that indicates that, if a simplified form of one of these shell is evaluated, the evaluation of two others forms is superfluous.

Another drawback is that shells are sharable among all objects of the database. This means that users cannot disable or enable a constraint only for specific object(s). Also users cannot handle situations in which some objects are exceptions for some constraints, that is they violate constraints but they are stored in the database. To avoid these drawbacks we introduce the concept of the *constraint kernel*.

5.2 Constraint Kernel

To avoid drawbacks of considering shells as the only means to represent constraints, shells are associated with another type of objects called *constraint kernels*. Kernels will store information that is local to objects and which is common among interrelated objects. For instance, if a relationship exists among objects Pr, Co, and Se, e.g. $Pr.teaches = Se$ and $Se.isSectionOf = Co$, then there is a unique kernel K that is sharable among three objects pr, co, and se. This suitation is depicted in (c) of Figure 4. The kernel K stores the following local information of W_1 w.r.t. objects Pr, Co, and Se: the evaluation of the constraint W_1, whether W_1 is or is not applicable to these interrelated objects, and the last time a check was carried out to constraint W_1 by any of these object.

The last time a check was done to a constraint is stored to avoid the superfluous checking for the constraint in more than one place. Let TC denote the last time a check to W_1 was done by any of the objects Pr, Se, or Co. Also let TU denote the last time an update occurred to object $o \in \{Pr, Se, Co\}$ violating constraint W_1. We can avoid unnecessary checking of W_1 at object o by testing whether $TU > TC$ or not. If $TU > TC$ then this means that a check was carried out to W_1 *before* the last update to o happened and, thus, the constraint must be check again. Otherwise the check of W_1 is unnecessary as it was done *after* the last update to any of these objects.

This basic idea already determines the structure of kernels which is shown in Figure 5 and includes:

Constraints. Each kernel stores the truth evaluation of a constraint w.r.t. object(s). Thus there must be a link between the kernel and this constraint. This link is useful if the user wants to know which constraints are violated by object(s). The attribute *constraint* represents the relationship from *kernel* to IC. The type of *constraint* is IC. For kernel k in (c) of Figure 4 the value of *constraint* is the object that represent the constraint W_1 in class IC. This object has OID #25 in (a) of Figure 3.

Constraint evaluation. The main function of a kernel is to store sharable information of constraints among interrelated object(s). One part of this information is of course the evaluation of the constraint. The attribute *evaluation* of type *boolean* stores the truth evaluation of the constraint that is referred to by attribute *constraint* of the kernel. The initial value of *evaluation* is *true*.

Last checking time. The last time at which the constraint was checked is stored in the kernel to avoid superfluous constraints checking as discussed above. The attribute *lastCheck* of type *string* holds this value. The initial value of *lastCheck* is the instance of time at which the kernel is created.

The discussion above motivates the type definition of attribute *constraints* of class *Any* to be of the set-structured type $\{Constraint\}$, where the type *Constraint* is defined as an aggregation of a kernel, a shell, and an attribute named *lastUpdate* (see (a) of Figure 4). This type definition is shown in Figure 5.

6 Consistency Maintenance

For consistency maintenance we augment every class in the database schema with a set of methods. The purposes of these methods are twofold: First, they are the only means for objects manipulation. Second, their semantics includes all tasks of integrity control that we described before, that is:

(1) linking each newly created object with shells and kernels;
(2) unifying kernels among shells of interrelated objects;
(3) monitoring updating to objects states and checking immediate constraints;
(4) checking violated constraints locally w.r.t. an object and globally w.r.t. all objects that are updated by a transaction;
(5) disabling or enabling constraints.

The tasks of (1) are included in the semantics of the method *new* of class *Any*.

We augment each class C with a set of methods called *control methods*. The motivation for doing that is to include tasks (2) and (3) in the semantics of these methods. Each control method corresponds to an attribute in C and is the only means for updating that attribute. For each attribute A of a class C there is either one or two control methods defined according to its type.

If A is a simple or a reference attribute then there is only one control method $A(t) : string$ where t is the type of A. For instance, the simple attribute *salary* of class *Employee* has the

Class *Kernel* [
constraint : IC,
evaluation : {*boolean*},
lstatus : *string*,
lastCheck : *string*,

Type *Constraint* [
shell : *Shell*,
lastUpdate : *string*,
kernel : *Kernel*]

Class *Any* [
⋮ : ...,
constraints : {*Constraint*},
⋮ : ...]

Fig. 5. Structures of class *Kernel* and type *Constraint*

control method $salary(integer) : string$; and the reference attribute *teaches* of class $Professor$ has the control method $teaches(Section) : string$

If A is of tuple structure type t then there are two control methods that have the same name as the attribute A but with different arguments and semantics. This can be done by using method overloading. The first one has the signature $A(nil) : C$ and the second method has signature $A(t) : string$. This is necessary to allow for updating the whole structure of the attribute A as well as any of its components. For instance, for attribute *dateOfBirth* of class *Employee* there are two control methods, $dateOfBirth(nil) : Employee$ and $dateOfBirth(Date) : string$. Thus we can update the date of birth of an employee as a whole as in the statement, $e.dateOfBirth(\text{"}25.3.1967\text{"})$ or through one of its components as in the statement, $e.dateOfBirth().day(\text{"}25\text{"})$.

For tasks (4) and (5) we add some methods to the root class *Any* and the class $Transaction$. In the remainder of this section we present the semantics of integrity control methods and describe how they can be used for consistency maintenance.

Objects Creation; $new(o, C)$: A new object named o of class C is created. Constraints of object o are selected from integrity control and stored in *o.constraints*. If there are two or more shells in *o.constraints* of the same constraint, kernels of these shells are unified.

Updating Internal State of Objects; $o.A_1() \ldots A_n(v)$: An update to the state of o is carried out by means of assignment $o.A_1 \ldots A_n \leftarrow v$, where $A_1 \ldots A_n$ is an intra-path w.r.t. class C. All immediate constraints are checked. The update must be made undone if constraints are violated. The value of the attribute *lastUpdate* of each affected constraint of deferred mode is set to current time.

Updating Reference Attributes; $o.A(o')$: A link between two objects o and o' is established by the assignment $o.A \leftarrow o'$. If there are shells in *o.constraints* and *o'.constraints* of the same constraint then the kernel of shells in *o.constraints* becomes the kernel of shells of *o'.constraints*.

Local Check; $o.check()$: For each t in $con \in o.constraints$ the simplified form $t.shell.form$ is evaluated if $t.lastUpdate > t.kernel.lastCheck$. The evaluation is stored in *t.shell.evaluation* and the user is informed in case of constraint violation.

Global Check; $T.check()$: Each object modified by transaction T is stored in $T.updatedObject$. $T.Check()$ does a local check for every object o in $T.UpdatedObject$.

Global Enabling or Disabling of Constraints; $enabled(c : string)$, $disabled(c : string)$: A constraint W can be enabled or disabled globally, i.e. for all objects of the database, by using the methods $enabled(W)$ and $disabled(W)$, respectively. After calling $enabled(W)$ (resp. $disabled(W)$) the value of the attribute *status* of object o in the class IC which represent a constraint W is changed to *enabled* (resp. *disabled*).

Local Enabling or Disabling of Constraints: A constraint W can be enabled or disabled locally, i.e. for a specific object o of the database, by changing the value of *t.kernel.lstatus* to *disabled* for every t in *o.constraints* such that $t.kernel.constraint.name = W$.

Example 7. *SalaryRaise* is a short transaction. The intended meaning of *SalaryRaise* is to raise salary of each employee whose salary is less than 1000.

Begin $Transaction\ SalaryRaise(percent : numeric)$
(1) $VEmployee$: $Employee$;
(2) $NewSalary$: $real$;
(3) **For** $VEmployee \in \{o \mid o \in Employee \wedge o.salary < 1000\}$ **Do**
(3.1) $NewSalary \leftarrow VEmployee.salary * percent + VEmployee.salary$
(3.2) $VEmployee.salary(NewSalary)$;
(4) **EndDo**.
(5)$SalaryRaise.Check()$
(6)**End** $SalaryRaise$.

In steps (1) and (2) variables $VEmployee$ of type $Employee$ and $NewSalary$ of type $real$ are declared. Step (3) is an iterator, that is, for each object of type $Employee$ satisfying the given condition, steps (3.1) and (3.2) are executed. Since each of these objects is already stored in the database, it was already be associated with its constraints at its time of creation. Step (3.1) is a calculation for new salary. In step (3.2) the salary is increased and the affected constraints of immediate mode of the object that is referred to by variable $VEmployee$ is checked. If one of these constraints is violated the user will be informed for constraint violation and the transaction is rolled back. During this iteration every updated object is added implicitly to $T.updatedObject$. At the end of the transaction all affected deferred constraints of updated objects are checked by $SalaryRaise.Check()$. □

7 Related Work

Several approaches have been proposed for consistency management in object-oriented databases. This has been done in the context of passive databases and persistent programming languages [BS97,JQ92,EGB93], and active databases [CFP94,GJ91,Día92]. These works can be divided into two groups.

The first group takes the direction that constraints should be encoded in application programs. Their motivation is efficiency of consistency checking. Works in this group provide techniques either to generate checking code for constraints declaratively specified [EGB93,BLR92,BD95]; or to prove correctness of methods of classes w.r.t. integrity constraints [BS96,BS97].

Works in this direction are applicable to a specific types of integrity constraints such as intra-constraints for otherwise the generated checking code will be too complex and needs further manual optimization. Moreover, w.r.t. constraints management they inherit problems of application-oriented techniques that we have discussed in the introductory section. Although constraints are specified as a part of the class structure in ODE object-oriented system, "*its association with the class is merely a notational convenience*"[JQ92].

The second group takes the direction that constraints should not be specified in applications programs but as (ECA) rules. Their motivation is to avoid problems of application-oriented techniques. Works in this group provide techniques for deriving ECA rules from constraints declaratively specified [Día92,BMP91]. Most works in this direction are directed to a specific OODBMS such as [GJ91] (see [WC96] for more details).

We believe that none of these approaches are addressing the problem of consistency management from broader scope and how database catalog can be designed and used as repository for integrity constraints.

Approaches for consistency management have also been proposed in other domains such as software engineering. The features which we listed in the introductory section and which our approach provides meet requirements of these applications. The main difference between the work presented for instance in [TC98] and ours is that our approach provides these features only by means of basic notations of OO data models and hence on an abstract level rather than on the implementation level.

8 Conclusion

Our proposal for consistency management in OODBMS provides a significant improvement in comparison with currently existing consistency management techniques for OODBMS. Integrity constraints are treated as first class citizens in our approach by considering them as objects of special class. In this way constraints can in principle be manipulated in the same way as all other objects in an object database.

In order to achieve an efficient checking of constraints we introduced *shells* and *kernels*. These constructions allow us to check constraints only if it is really necessary. Of course, a certain overhead is needed to adequately administrate integrity constraints, shells, and kernels in order

to keep them consistent. However, this enables us to check constraints as locally as possible and only in cases where they can really be violated.

Future work will concentrate on a more general technique for consistency maintenance than the one we presented in Section 6, and the detailed aspects of implementing the proposed approach in existing object database systems.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the 1st Int. Conf. on Deductive and Object Oriented Databases*, pages 40–57, Amsterdam, December 1989. North-Holland.

[BD95] V. Benzaken and A. Doucet. Thémis: A Database Programming Language Handling Integrity Constraints. *VLDB Journal*, 4(3):493–517, July 1995.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System – The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[BLR92] V. Benzaken, C. Lécluse, and P. Richard. Enforcing Integrity Constraints in Database Programming Languages. In *Proc. of the 5th Int. Workshop on Persistent Object Systems*, Workshops in Computing, pages 282–299. Springer-Verlag, 1992.

[BMP91] C. Bauzer-Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proc. of the 5th European Conf. on Object Oriented Programming (ECOOP'91)*, volume 512, pages 219–230. Springer-Verlag, 1991.

[BS96] V. Benzaken and X. Schaefer. Ensuring Efficiently the Integrity of a Persistent Object Store via Abstract Interpretation. In *Proc. of the 7th Int. Workshop on Persistent Object Systems*. Morgan Kaufmann, May 1996.

[BS97] V. Benzaken and X. Schaefer. Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Transformers. In *Proc. of the 11th European Conference on Object-Object-Oriented Programming, (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 60–84, Berlin, 1997. Springer-Verlag.

[Cat96] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[CFP94] S. Ceri, P. Fraternali, and S. Paraboschi. Constraint Management in Chimera. *Bulletin of the IEEE Technical Committee on Data Engineering*, 17(2):4–8, June 1994.

[Día92] O. Díaz. Deriving Active Rules for Constraint Maintenance in an Object-Oriented Database. In *Proc. of the 3rd Int. Conf. on Database and Expert System Applications*, pages 332–337. Springer-Verlag, 1992.

[EGB93] Suzanne M. Embury, Peter M. D. Gray, and N. Bassiliades. Constraint Maintenance using Generated Methods in the P/FDM Object-Oriented Database. In *Proc. of the 1st Int. Workshop on Rules in Database Systems*, Workshops in Computing, pages 364–381, Berlin, 1993. Springer-Verlag.

[GJ91] N. H. Gehani and H. V. Jagadish. ODE as an Active Database: Constraints and Triggers. In *Proce. of the 17th Int. Conf. on Very Large Data Bases*, pages 327–336. Morgan Kaufmann, September 1991.

[JQ92] H. V. Jagadish and X. Qian. Integrity Maintenance in an Object-Oriented Database. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pages 469–480, San Mateo, Ca., USA, 1992. Morgan Kaufmann.

[Nic82] J.-M. Nicolas. Logic For Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[OS98] H. Oakasha and G. Saake. Integrity Independence in Object-Oriented Database Systems. In *Proc. of the 10th Workshop Grundlagen von Datenbanken, Konstanz, Germany*, number 63 in Konstanzer Schriften in Mathematik und Informatik, pages 94–98. University of Konstanz, May 1998.

[OS99] H. Oakasha and G. Saake. Foundations for Integrity Independence in Relational Databases. In T. Polle, T. Ripke, and K.-D. Schewe, editors, *Fundamentals of Information Systems*, chapter 10. Kluwer Academic Publishers, Boston, 1999.

[TC98] P.L. Tarr and L.A. Clarke. Consistency management for complex applications. In *Proc. of the 1998 Int. Conf. on Software Engineering*, pages 240–249. IEEE Computer Society, 1998.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.