12

A Model for Specifying Individual Integrity Constraints on Objects

Y. Lahlou
GMD, German National Research Center for Information technology
Schloss Birlinghoven,
D-53754 Sankt Augustin, Germany.
Tel: +49 2241 142146.
Fax: +49 2241 142071.
E-Mail: lahlou@gmd.de

Abstract

This paper presents an approach for specifying individual integrity constraints on objects, which is based on a conceptual class-based model that emphasizes the individual structure for objects by allowing those belonging to a class to have structures not previously abstracted in that class.

Integrity constraints specification on objects makes use of this particularity of the model, which yields particular constraint definition on individual objects. For instance, one can define a constraint on object o, that lies on the fact that o is composed of another object o'. This constraint is taken into account as long as o' is composing o. If this happen to cease, the constraint is disregarded.

This approach finds a suitable application area in the domain of architecture, where such a kind of constraints is important during the design process.

Keywords

Integrity constraints, Individual constraints, Object model, Object Design

1 INTRODUCTION

When designing object-oriented applications (e.g. databases) the capability to attach integrity constraints to objects or object classes is an important tool for achieving a coherent and correct design. In particular, attaching individual constraints to specific objects allows direct assistance for managing their life-cycle and imposing coherence rules on it.

In database models, the conceptual schema requires that each object belongs to a class and hence implements a structure that is common to a group of objects (the class

Integrity and Internal Control in Information Systems Volume 1
S. Jajodia, W. List, G. McGregor & L. Strous (Eds)

instances). Consequently, even when the DBMS used allows one to attach individual constraints on objects, these constraints make use of the class structure. We show in this paper that it is sometimes useful that objects have individual structures which can be involved in defining integrity constraints.

The problem is simultaneously a data model problem and a constraint specification problem. In this paper, we introduce an integrity constraints specification technique which takes into account individual object structures. This technique is based on a data model that emphasizes individual object structures which we investigated in a previous research (LAHLOU & MOUADDIB 1996) and which we will extensively refer to in this paper.

We present a short bibliographic survey on integrity constraints handling in databases, in section 2. We introduce the main features of the data model in section 3. Section 4 develops our approach for integrity constraints handling. A special interest is given to individual constraints. Finally, we show in section 5 that this approach has been implemented within a research prototype.

2 INTEGRITY CONSTRAINTS HANDLING IN DATABASES

Integrity constraints specification and handling is mandatory for keeping databases in coherent states.

Would it be a relational model (DELOBEL & ADIBA 1982, BRIALES & DE TROYER 1991), a semantic model (CHUNG, RIOS-ZERTUCHE, NIXON & MYLOPOULOS 1988, PECKHAM, MARYANSKI, BESHERS, CHAPMAN & DEMURJIAN 1989, CORTE & PRESENZA 1992, COOPER & QIN 1992) or an object oriented model (NASSIF, QIU & ZHU 1991, SU & ALASHQUR 1991, KIM, LEE & SEO 1992, KHOSHOFIAN 1993), each data model involves means by which data coherence is achieved, according to its specific features.

One can enumerate two types of integrity constraints (DELOBEL & ADIBA 1982, NASSIF et al. 1991):

- static constraints specify conditions which the database have to obey at any time; they express valid combinations of data in the database, according to the entity constructors of the model (e.g. official cars have dark colors);
- dynamic constraints define correct database transitions from a state to another, or in other words valid state modifications (e.g. salaries can not decrease).

We are more particularily interested in static constraints; among them (COOPER & QIN 1992):

- global constraints express particular data types that can be handled by the data model (e.g. ASCII, sound, video, ...);
- data model constraints set out rules for correct use and combination of the model constructors (e.g. each object instanciates one and only one class);
- database schema constraints rely on a specific user-defined database schema; they precise particular conditions inherent to application semantics (e.g. a student is not more than 30 years old);
- concrete data constraints address particular objects in the database (e.g. the client X should not have a debit account greater than 4000).

The relational model (DELOBEL & ADIBA 1982) distinguishes among database schema constraints some meaningful categories (key unicity, individual constraints, intra-relation constraints, inter-relation constraints).

Semantic and object oriented data models involve other types of constraints (COOPER & QIN 1992, KHOSHOFIAN 1993): existential constraints (equivalent to referential integrity in relational databases), specialization constraints, disjunction constraints and covering constraints on subclass instances.

Integrity constraints specification can be achieved within different terminologies (NASSIF et al. 1991). Equations are atomic constraints, consisting of two expressions (defined by paths in the database schema) separated by an operator. Assertions are combinations of equations with boolean operators, but not involving quantifiers. More complex constraints may be defined by using restrictions of first order predicates calculus (COOPER & QIN 1992).

Managing integrity constraints yields different strategies for restauring database integrity when a constraint is violated. The most obvious strategy is to prohibit every action that would violate a constraint (especially updates). More flexible strategies make use of the notion of transaction, which defines a set of actions between two coherent states of the database. While those actions are executed, some constraints may be temporarily violated (DELOBEL & ADIBA 1982, MEYER, WEIGAND & WIERINGA 1989, COOPER & QIN 1992).

Another approach aims at exhaustively managing the database behaviour by insuring its coherence after each elementary action (adding, update, suppression) (ABITEBOUL & VIANU 1989, WORBOYS 1991). Specifying integrity constraints is achieved in a dynamic way (rather than a predicative one) by listing all the operations that may violate the constraint along with mechanisms for avoiding it. This approach is particularily interesting in an object oriented environment, where encapsulation facilitates such a dynamic specification.

In such approaches, two main problems arise:

- insure elementary transactions coherence (correction),
- insure an equivalent predicative expressivity (completion).
  In (ABITEBOUL & VIANU 1989), completion is insured for some particular constraint types (especially functional dependencies).

While data model and application schema constraints (according to the classification in (COOPER & QIN 1992)) have been widely investigated, data (object) dependent constraints are derived from schema dependent constraints in most cases.

The omnipresence of database schemata in databases compels objects structures by "conceptual" ones (relationships, class structures, ...). This is of course a good design tool but also an obstacle against flexible design (one has to predict object structures before actually creating them).

This study aims at relaxing this constraint (!) by defining a flexible data model (LAHLOU & MOUADDIB 1996) and providing users with simple means for specifying individual and schema-dependent integrity constraints on objects. It is validated by examples taken from an architectural application.

We have not investigated integrity maintenance yet. As a first step, we only introduce a technique for specifying and checking integrity constraints.

3 THE DATA MODEL

Our data model is based on classical notions such as object classes, inheritance, ... but it allows objects related to a class to have individual features, not predicted within the class as instance variables.

Let us consider as an example the architectural project described in figure 1.

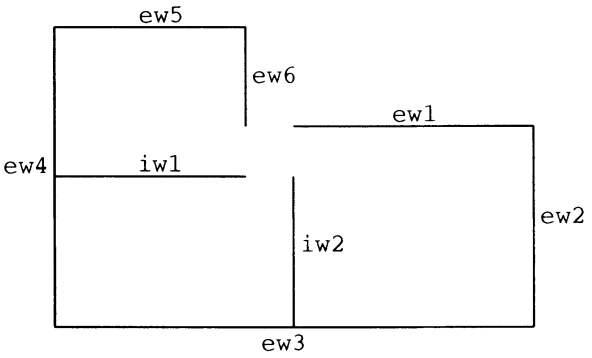

Figure 1 The plan for a construction (viewed from the top).

This construction is composed of six external walls (ew_1, ..., ew_6) and two in-

ternal ones (iw_1 et iw_2). It will be represented by object c which belongs to a class named $Construction$. In a classical class-based object model, one has to predict in that class some instance variables that would contain objects ew_i and iw_j composing the object c. But, in architectural design, objects structures are so different that it is not always easy to predict object structures and contents at a conceptual level (in classes).

In order to make it easier to design such applications, we proposed a data model (LAHLOU & MOUADDIB 1996) where objects belonging to a class instanciate its instance variables by attaching other objects to them, but also refer individually to other extra component objects. We also proposed a suited query language where queries address individual object contents as well as common conceptual class structures (LAHLOU & MOUADDIB 1996). From a structural viewpoint, this model can be considered as laying on both class-based and prototype-based object oriented models (LIEBERMAN 1986, DONY, MALENFANT & COINTE 1992).

In the following, we sum up the main features of the model. More details can be found in (LAHLOU & MOUADDIB 1996).

Fundamental notions in the model are types (cf. 3.1), classes (cf. 3.2), objects (cf. 3.3) and the realization link (cf. 3.4).

3.1 Types

We assume that there exists several terminal object types, corresponding to basic object types: integers (I), reals (R), strings (S), ... and a particular type named the *empty type*.

Object types, elements of **T** (set of all object types) are recursively defined by:

- terminal types and the empty type are types,
- if t is a type, then t^* is also a type, named *set type*,
- if t_1, ..., t_n are types, then $t_1 \times ... \times t_n$ is also a type, named *tuple type*.

Examples:
$R \times S^* \times I$ and $(R \times I)^*$ are types.

Assuming that its restriction to terminal types is known, we recursiveley define a partial order on **T** to represent the notion of sub-typing (LAHLOU & MOUADDIB 1996).

3.2 Classes

Object classes, elements of the set **C**, are defined as follows:

- a *terminal class* is the association of a class name with a terminal type,
- a *simple class* is defined by a class name c and a list $< a_1 : c_1, ..., a_n : c_n >$, possibly empty and named structure of c; we shall note $c =< a_1 : c_1, ..., a_n : c_n >$; the a_i are attribute names and the c_i are classes; if the structure of a simple class c is empty, c is an *empty class*,
- if c is a simple class, then c^* is also a class, named *set class*.

We assume that the recursive definition of classes does not yield cycles. In other words, a class c can not refer in its structure to a class c' which is one of its subclasses (according to the inheritance relation we will evoke later on).

Examples:
$Integer, String, Real, ...$, terminal classes
$Date =< day : Integer, month : Integer, year : Integer >$
$Person =< name : String, first\_name : String, age : Integer >$
$Construction =< architect : Person, date : Date, location : String,$
$characteristics : String^* >$
$External\_wall =< inf\_point_1 : Point, inf\_point_2 : Point, height : Real >$
$Internal\_wall =< inf\_point_1 : Point, inf\_point_2 : Point, height : Real >$
$Point =< x : Real, y : Real, z : Real >$

To each class c is associated an object type noted $[[c]]$, recursively defined from terminal classes types (LAHLOU & MOUADDIB 1996).

Example:
$[[Construction]] = [[Person]] \times [[Date]] \times [[String]] \times [[String^*]]$
$= ([[String]] \times [[String]] \times [[Integer]]) \times ([[Integer]] \times [[Integer]] \times [[Integer]]) \times$
$[[String]] \times [[String]]^*$
$= (S \times S \times I) \times (I \times I \times I) \times S \times S^*$

Just like the subtyping relation between object types, we define an inheritance relation between classes as a partial order on **C**, that respects the subtyping relation on related object types (LAHLOU & MOUADDIB 1996).

3.3 Objects

An object $o = (i, s)$, element of **O** (set of all objetcs), is defined by a unique and exclusive identifier $i(o) \in \mathbf{I}$ (where **I** is the set of all object identifiers) and a structure $s(o)$.

We distinguish three types of objects, according to their structure s:

- *terminal objects*, where the structure is a value associated to a given terminal type;
- *simple objects*, where the structure is a list $< a_1 : o_1, ..., a_n : o_n >$, possibly empty (in which case o is an *empty object*), where the a_i are attribute names and the o_i are objects named "components"; for components that are not in the class of the object (individual components), a special attribute, noted X, will be used;
- *set objects*, where the structure is a set $\{o_1, ..., o_p\}$, where the o_i are objects.

Examples:
Here, we give examples of object structures, related to figure 1; the whole construction is represented by object c.

$s(c) =< architect : a_1, date : d_1, location : s_1, characteristics : e_1, X : ew_1, X : ew_2, X : ew_3, X : ew_4, X : ew_5, X : ew_6, X : iw_1, X : iw_2 >$
$s(a_1) =< name : s_2, first\_name : s_3, age : n_1 >$
$s(s_2) =$ "$Duchemin$"
$s(s_3) =$ "$Emile$"
$s(n_1) = 50$
$s(d_1) =< day : n_2, month : n_3, year : n_4 >$
$s(n_2) = 12$
$s(n_3) = 12$
$s(n_4) = 1996$
$s(s_1) =$ "$Nancy$"
$s(e_1) = \{s_4, s_5\}$
$s(s_4) =$ "$By\ the\ sea$"
$s(s_5) =$ "$Sloping$"
$s(ew_1) =< inf\_point_1 : p_1, inf\_point_2 : p_2, height : r_1 >$
$s(p_1) =< x : r_2, y : r_3, z : r_4 >$
$s(r_2) = 14.0$
$s(r_3) = 11.0$
$s(r_4) = 0.0$
$s(p_2) =< x : r_5, y : r_6, z : r_7 >$
$s(r_5) = 23.0$
$s(r_6) = 11.0$
$s(r_7) = 0.0$
$s(r_1) = 4.0$

$s(ew_2) = ...$

To each object o is associated an object type noted $[[o]]$, recursively defined from terminal object types. For set objects, the related object type is the upper bound of the types of the composing objects, according to the sub-typing relation. This definition makes sense because each set of types admits an upper bound (in the worst case, the empty type, greatest element of $\mathbf{T}$) (LAHLOU & MOUADDIB 1996).

3.4 Realization link

The realization link is a binary relation, noted $\leftarrow$, that is defined on the set $\mathbf{O} \times \mathbf{C}$. It takes the place of the instanciation link of class-based object models and is defined (LAHLOU & MOUADDIB 1996) in such a way that if $o \leftarrow c$, then $[[o]]$ is a subtype of $[[c]]$.

With analogous definitions in a class-based model, the two object types would have been identical.

Examples:
According to the previous definitions, we have:

$c \leftarrow Construction$ ($[[c]]$ strict subtype of $[[Construction]]$)
$a_1 \leftarrow Person$ (identical types)
$d_1 \leftarrow Date$ (identical types)
$e_1 \leftarrow String^*$ (identical types)
$ew_i \leftarrow External\_wall$ (identical types)
$iw_j \leftarrow Internal\_wall$ (identical types)
$p_k \leftarrow Point$ (identical types)
$s_u \leftarrow String$ (identical types)
$n_v \leftarrow Integer$ (identical types)
$r_w \leftarrow Reall$ (identical types)

Remarq:
In order to provide objects with richer structures than their classes', we defined separately the notions of type and class. Opposite to the terminology used in (HULL 1989), a class is not a set of objects; it refers to an object type that is more generic than that of its related objects (according the the subtyping relation).

3.5 Discussion

Such a data model allows for an easier definition of application schemata since schema designers do not have to predict at the class level the exact structures of objects. This model is especially suited for what we call "incremental design": minimal definition of classes, then object creation with richer structures and then redefinition of classes by use of actual common object properties.

The fact that objects can have individual structures can also be used for several interesting tools. In querying the database, it allows for specifying special content-based queries rather than the common navigational ones (LAHLOU & MOUADDIB 1996). In integrity management, it allows for specifying special individual integrity constraints on objects.

Now, we present integrity constraints specification according to the model features. The constraint model makes full use of the realization link in allowing particular individual integrity constraints to be defined on objects.

4 INTEGRITY CONSTRAINTS

In this section, we introduce integrity constraints specification for the model presented above. We provide means for specifying constraints and a technique for evaluating them on a given state of the database. We do not consider (in the purpose of this study) the integrity enforcement. Our aim here is only to show that the data model yields particular individual integrity constraints that can be very useful in several application areas (e.g. architecture).

4.1 Constraint specification

Constraint specification is achieved by means of *assertions*: predicates specified according to a particular formalism, that we introduce in the following.

Assertions may be defined on classes or on specific objects. When an assertion is defined on a class, it must be satisfied by all objects realizing that class or one of its sub-classes. When an assertion is defined on an object, it obviously must be satisfied by that object only.

In order to specify integrity constraints, we first define some preliminary notions: valid paths and destinations (*Dest* function) for classes and objects.

A path is a list of attribute names from a class or an object, separated by dots; e.g. $date.day$ or $architect.age$. As we introduced the particular attribute name X for individual object components, we allow paths defined on objects to contain component

object names (cf. b).

(a) Valid paths and their destinations for classes

Simple classes:

- A path of length 1: $p = a$, is valid for a class $c =< a_1 : c_1, ..., a_n : c_n >$ if and only if $\exists i \in [1, n], a_i = a$; then we have: $Dest(c, p) = c_i$,
- a path $p = a_1...a_q (q > 1)$, is valid for a class c, if and only if $p' = a_1...a_{q-1}$ is valid for c and $p" = a_q$ is valid for $Dest(c, p')$; then we have: $Dest(c, p) = Dest(Dest(c, p'), p")$.

Set classes:

a path is valid for c^* if and only if it is valid for c; then we have: $Dest(c^*, p) = (Dest(c, p))^*$.

Example:
$Dest(Construction, date.month) = Dest(Dest(Construction, date), month)$
$= Dest(Date, month)$
$= Integer.$

(b) Valid paths and their destinations for objects

First of all, a path p which contains no object names is valid for an object o, if and only if $\exists c \in C, o \leftarrow c$ and p valid for c.

As mentioned before, particular paths are defined for objects. They begin with object names: $p = o_1...o_n.a_1...a_m$. They are intended to be used for imposing constraints related to the fact that an object is composing another, when the component does not correspond to an instance variable from a class which the composed object realizes.

If one desires to attach a constraint involving an object o', *in its capacity as component of object* o, the constraint will be defined on object o (and not on the component object o'), by preceding each path involved in the constraint by (the name of) o'. The path is valid for o if and only if the remaining path from removing o' is valid for o'; this remaining path can in turn begin with (the name of) a component of o' (cf. examples below).

This kind of paths allows for making full use of the free object structures according to their classes', in the model. We will discuss how important this particular feature is, in more details in section f.

Now, let us define the destination of a path for an object.

Simple objects:

- paths of length 1: $Dest(o, a) = o.a$: object corresponding to the attribute a in $s(o)$; if a is the name of a component o', then $Dest(o, o') = o'$.
- paths of length $q > 1$: $Dest(o, a_1 ... a_q) = Dest(Dest(o, a_1 ... a_{q-1}), a_q)$.

Set objects:

$Dest(o, a) = o'$ such that: $s(o') = \{Dest(o_i, a), o_i \in s(o)\}$.

Evaluating this destination yields the creation of a new object that did not exist in the database before.

Examples:
$Dest(c, architect.name) = Dest(Dest(c, architect), name)$
$= Dest(a_1, name)$
$= "Duchemin"$.

$Dest(c, ew_1.height) = Dest(Dest(c, ew_1), height)$
$= Dest(ew_1, height)$
$= 4.0$.

Remarq:
The destination of a path is: either a class, if the origin is a class, or an object, if the origin is an object.

In the following, we successively define the notions of expression, equation and assertion, which are the bases of the constraints model. Again, a constraint is defined either on a class (it must be satisfied by all objects realizing that class and its subclasses) or on an object (it must be satisfied by that object).

(c) Expressions

We define the notion of expressions (for classes and objects) along with the notion of types of expressions. Given an expression ε, we associate an object type $[[\varepsilon]]$ (element of **T**) to it.

- Objects (elements of **O**), represented by their names or their values (structures) for terminal objects, are valid expressions for all objects and classes. Their types are the related object types.

Examples:
ew_1, iw_2, 5, "$toto$", ...

- If $o \in \mathbf{O}$ is an object, and if p is a valid path for o, then $o.p$ is a valid expression for all objects and classes. Its type is $[[Dest(o,p)]]$, which is an element of $\mathbf{T}$.
  Examples:
  $c.architect.name$, $ew_1.height$, ...
- – Each valid path p for a class c is a valid expression for that class. Its type is $[[Dest(c,p)]]$, which is an element of $\mathbf{T}$.
  – Each valid path p for an object o is a valid expression for that object. Its type is $[[Dest(o,p)]]$, which is an element of $\mathbf{T}$.
  Examples:
  The path $height$ for class $External\_wall$.
  The path $height$ for object ew_2.
- If ε_1, ..., ε_n are valid expressions (for a class or an object) whose types are terminal types, and if φ is a function defined from $[[\varepsilon_1]] \times \ldots \times [[\varepsilon_n]]$ to a type t, then $\varphi(\varepsilon_1, ..., \varepsilon_n)$ is also a valid expression for the concerned class or object, whose type is t.
  This notation groups all arithmetic functions (addition, substraction, ..., sinus, logarithm, power, ...), functions on character strings (concatenation, ...), ...
  Examples:

  – $sup(inf\_point_1.y, inf\_point_2.y)$ is a valid expression for class $External\_wall$.
    The type of this expression is R (reals).
  – If a class c has an attribute a whose domain is of type R, then the following expressions are valid for c: $a + log(a) - 1$ and $sin(a)$.
    Their types are respectively R and R.

- If ε is a valid expression (for a class or an object) such that $[[\varepsilon]] = t^*$, $t \in \mathbf{T}$, then $card(\varepsilon)$ is also a valid expression for that class or that object and its type is I.
  Example:
  $card(characteristics)$ is a valid expression for class $Construction$.

(d) Equations

An equation generally consists of two expressions separated by an operator whose evaluation yields a boolean value; this operator my be equality, inclusion, superiority, ..., or their negation.

In the following, each time an equation uses two expressions, it is valid for a class (resp. an object) if both expressions are.

- If ε_1 and ε_2 are expressions such that $[[\varepsilon_1]] = [[\varepsilon_2]]$ or $[[\varepsilon_1]], [[\varepsilon_2]] \in \mathbf{T}$, are comparable object types, according to the partial order on object types, then $\varepsilon_1 =$

ε_2 and $\varepsilon_1 \neq \varepsilon_2$ are equations.
Examples:
$height = 4.0$ for class $External\_wall$.
$Architect = a_1$ for object c.

- If ε_1 and ε_2 are expressions such that $[[\varepsilon_1]]$ and $[[\varepsilon_2]]$ are the same terminal type having a partial order (I, R or S), then $\varepsilon_1\ op\ \varepsilon_2$ is an equation, with $op \in \{<, >, \geq, \leq\}$.
  Example:
  $inf\_point_1.x < 20.0$ for class $External\_wall$.
- If ε_1 and ε_2 are expressions such that $[[\varepsilon_1]], [[\varepsilon_2]] \in \mathbf{T}$, with $[[\varepsilon_1]] = [[\varepsilon_2]]^*$, then $\varepsilon_2 \in \varepsilon_1$ and $\varepsilon_2 \notin \varepsilon_1$ are equations.
  Example:
  $"Concrete" \in characteristics$ for class $Construction$.
- If ε_1 and ε_2 are expressions such that $[[\varepsilon_1]]$ and $[[\varepsilon_2]]$ are set types, then $\varepsilon_1 \subseteq \varepsilon_2$ and $\varepsilon_1 \not\subseteq \varepsilon_2$ are equations.
  Example:
  $\{"Concrete", "Wood"\} \subseteq characteristics$ for class $Construction$.

(e) Assertions

Assertions (the term is borrowed from (CHUNG et al. 1988)) are the main primitives for specifying integrity constraints in our model. They are based on logical combinations of equations.

Again, in the following, each time an assertion uses some equations, it is valid for a class (resp. an object) if all equations are.

- Each equation is an assertion.
- If α is an assertion, then $\neg\alpha$ is also an assertion.
  Example:
  $\neg("Concrete" \in characteristics)$ is a valid assertion for class $Construction$.
- If α_1 and α_2 are assertions, then $\alpha_1 \vee \alpha_2$, $\alpha_1 \wedge \alpha_2$, $\alpha_1 \Rightarrow \alpha_2$ and $\alpha_1 \Leftrightarrow \alpha_2$ are assertions.
  Example:
  $(architect.name = "Duchemin") \Rightarrow ("Wood" \in characteristics)$ is a valid assertion for class $Construction$.

(f) Discussion

We have presented a language for specifying integrity constraints on our data model. This language attaches constraints either to classes or objects. The coherence of a database may then be defined at several levels, involving different descriptive elements of the database.

Especially, constraints making use of the flexibility of the realization link have

been defined by means of paths including successive object names. This kind of constraints is particularily useful for defining ponctual coherence rules on specific objects.

As an example, during the design process for the construction c described previously in figure 1, the designer may move an external or internal wall in a wrong way, according to some architectural criteria.

Let us assume that during the whole design process, the room delimited by external walls ew_4, ew_5, ew_6 and internal wall iw_1 should be sufficiently wide (width superior to a threshold).

This constraint can obviously not be defined within the class $Construction$, since it involves individual components of object c, absent in the class definition.

Our constraints language allows one to specify such a constraint with an assertion attached to object c, e.g. $ew_5.inf\_point_1.y - iw_1.inf\_point_1.y \geq 7.0$ which is a valid assertion for that object, according to the previous definitions.

This constraint obliges walls ew_5 and iw_1 to stay sufficiently distant from each other during the whole design process. If it happens that after an update they become too close, the constraint is violated. It shows how useful it is to include component names in individual constraints.

This feature of the constraint language allows for attaching criteria of coherence to an object, not only by using its class structure (with valid class paths) but also its individual components. The previous constraint is thus attached to object c but addresses its individual components ew_5 et iw_1.

As a matter of fact, this constraint addresses those objects *in their capacity of components of object* c, which justify the fact that it is attached to object c.

4.2 Evaluating constraints

Integrity constraints evaluation is achieved on database objects. The objective now is to give formal semantics for the constraints language presented above. We will define a method that accepts an object and a valid constraint for that object and returns a boolean value stating of its satisfaction ($TRUE$ if the object satisfies the constraint and $FALSE$ otherwise).

We note $eval(\rho, o)$ the result of evaluating a part ρ of a constraint (path, expression, ...) on object o. The evaluation function $eval$ yields boolean results.

(a) Evaluating expressions

Let ε be an expression and $o \in \mathbf{O}$ be an object.

- If $\varepsilon = o' \in \mathbf{O}$, then $eval(o', o) = o'$.
  Examples:
  $eval(ew_1, o) = ew_1$
  $eval(5, o) = 5$
  $eval("toto", o) = "toto"$
- If $\varepsilon = o'.p$, having $o' \in \mathbf{O}$ and p valid path for o', then $eval(o'.p, o) = Dest(o', p)$.
  Example:
  $eval(ew_1.height, c) = Dest(ew_1, height) = 4.0$.
- If $\varepsilon = p$, having p valid path for objet o, then $eval(p, o) = Dest(o, p)$.
  Example:
  $eval(architect.first\_name, c) = Dest(c, architect.first\_name) = "Emile"$
- If $\varepsilon = \varphi(\varepsilon_1, ..., \varepsilon_n)$, then $eval(\varepsilon, o) = \varphi(eval(\varepsilon_1, o), ..., eval(\varepsilon_n, o))$.
  Example:
  $eval(sup(inf\_point_1.y, inf\_point_2.y), ew_1) =$
  $sup(eval(inf\_point_1.y, ew_1), eval(inf\_point_2.y, ew_1))$
  $= sup(Dest(ew_1, inf\_point_1.y), Dest(ew_1, inf\_point_2.y))$
  $= sup(0.0, 4.0)$
  $= 4.0$

  Remarq:
  Such evaluations on objects might yield errors (division by zero, logarithm of a negative quantity, ...) even if the expression is *a priori* valid. In that case, the constraint is automatically violated, e.g. $ew_2.inf\_point_1.z / ew_1.inf\_point_2.z$.
- If $\varepsilon = card(\varepsilon')$ then $eval(\varepsilon, o) = card(s(eval(\varepsilon', o)))$
  Example:
  $eval(card(characteristics), c) = card(s(eval(characteristics, c)))$
  $= card(s(e_1))$
  $= card(\{"By\ the\ sea", "Sloping"\})$
  $= 2$

(b) Evaluating equations

Let ε_1 and ε_2 be two expressions and $o \in \mathbf{O}$ be an object.

According to d, an equation takes the following form: $\varepsilon_1\ op\ \varepsilon_2$, where op is an operator in the set $\{=, \neq, >, <, \geq, \leq, \in, \notin, \subseteq, \not\subseteq\}$.

$$eval(\varepsilon_1\ op\ \varepsilon_2, o) = eval(\varepsilon_1, o)\ op\ eval(\varepsilon_2, o).$$

Example:
$eval("Sloping" \in characteristics, c) = "Sloping" \in eval(characteristics, c)$
$= "Sloping" \in \{"By\ the\ sea", "Sloping"\}$
$= TRUE$

(c) Evaluating assertions

- $eval(\neg\alpha, o) = \neg eval(\alpha, o)$.
  Example:
  $eval(\neg(\text{"}Sloping\text{"} \in characteristics), c) = FALSE.$
- If $lop \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$, then $eval(\alpha_1\, lop\, \alpha_2, o) = eval(\alpha_1, o)\, lop\, eval(\alpha_2, o)$.
  Example:
  $eval((architect.name = \text{"}Duchemin\text{"}) \Rightarrow (\text{"}Wood\text{"} \in characteristics), c) = FALSE.$
  It comes from the following results:
  $eval(architect.name = \text{"}Duchemin\text{"}, c) = TRUE$
  $eval(\text{"}Wood\text{"} \in characteristics, c) = FALSE.$

To conclude this section, we remind that we have only presented a constraint specification language and formal semantics for it (taking the form of an evaluation function). Techniques for enforcing such constraints have to be defined, and that is one of the future developments for this study.

5 IMPLEMENTATION

The data model we have presented in section 3 has been validated by a research prototype written in Smalltalk-80 (GOLDBERG & ROBSON 1983). This prototype actually implements a more sophisticated model, named EMIR (Extended Model for Information Retrieval) (LAHLOU 1996), which is based on the fundamental notion of realization link and integrates other features such as semantic relationships between objects.

We have also defined a formal language that expresses the integrity constraints model presented in section 4 (ISSELE 1995).

The prototype allows for:

- defining integrity constraints according to the constraints model presented in 4.1 within a graphical interface supporting automatic assistance for navigation through class and object structures when formulating expressions,
- checking the database coherence at a given state, according to the evaluation technique presented in 4.2 and displaying all violated constraints along with the related objects.

6 CONCLUSION

This paper presented particular aspects of integrity management in databases related to a class-based object data model that allows objects to have individual references to other objects.

We have proposed an integrity constraints specification model, by means of assertions on such databases. Assertions might be related to classes or particular objects. Constraint specification makes full use of the data model properties, especially the so-called realization link (paths with object names). This allows for attaching to objects some constraints related to the fact that particular objects are composing them, which is very useful as we have shown in domains such as architectural design.

The approach has been validated within a research prototype.

Future developments for this work will deal with actual integrity enforcement. We have only presented an evaluation technique for integrity constraints (for checking constraints satisfaction at a given state of the database) but in the future we are expecting to define means for forbidding any operation on the databases that would violate specified constraints.

REFERENCES

ABITEBOUL, S. & VIANU, V. (1989), 'A Transaction-Based Approach to Relational Database Specification', *Journal of the ACM* **36**(4), 758–789.

BRIALES, M. & DE TROYER, O. (1991), Object-Oriented Integrity Enforcement in a Relational Environment, *in* 'British National Conference on Databases, BNCOD'91', Wolverhampton, UK, pp. 38–68.

CHUNG, K., RIOS-ZERTUCHE, D., NIXON, B. & MYLOPOULOS, J. (1988), Process Management and Assertion Enforcement for a Semantic Data Model, *in* 'Extending Database Technology, EDBT'88', Springer-Verlag, Lecture Notes in Computer Science, No 303, pp. 469–487.

COOPER, R. & QIN, Z. (1992), A graphical Data Modelling Program with Constraint Specification and Management, *in* 'British National Conference on Databases, BNCOD'92', Springer-Verlag, Lecture Notes in Computer Science, No 618, pp. 192–208.

CORTE, P. & PRESENZA, D. (1992), Understanding Data Behavior from its Static Structure, *in* 'The 5th International Conference on Putting into Practice Methods and Tools for Information System Design', Nantes, France, pp. 291–302.

DELOBEL, C. & ADIBA, M. (1982), *Bases de données et systèmes relationnels*, Dunod Informatique.

DONY, C., MALENFANT, J. & COINTE, P. (1992), Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation, *in*

'Object-Oriented Programming Systems, Languages and applications, OOPSLA'92', pp. 201–217.

GOLDBERG, A. & ROBSON, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

HULL, R. (1989), Four Views of Complex Objects: A Sophisticate's Introduction, *in* 'Nested relations and complex objects in databases', Springer-Verlag, Lecture Notes in Computer Science, No 361, pp. 87–116.

ISSELE, N. (1995), 'Environnement de conception et de recherche d'information dédié au modèle EMIR', Thèse CNAM.

KHOSHOFIAN, S. (1993), *Object-Oriented Databases*, Wiley Professional Computing.

KIM, W., LEE, Y. & SEO, J. (1992), 'A Framework for Supporting Triggers in Object-Oriented Database Systems', *International Journal of Intelligent & Cooperative Information Systems* **1**(1), 127–143.

LAHLOU, Y. (1996), 'Modélisation et recherche basées sur le contenu d'objets complexes. Le système EMIR', Thèse de l'Université Henri Poincaré - Nancy I.

LAHLOU, Y. & MOUADDIB, N. (1996), Relaxing the Instantiation Link: Towards a Content-Based Data Model for Information Retrieval, *in* 'Conference on Advanced Information Systems Engineering, CAiSE'96', Springer-Verlag, Lecture Notes in Computer Science, No 1080, pp. 540–561.

LIEBERMAN, H. (1986), Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, *in* 'Object-Oriented Programming Systems, Languages and applications, OOPSLA'86', pp. 214–223.

MEYER, J., WEIGAND, H. & WIERINGA, R. (1989), A Specification Language for Static, Dynamic and Deontic Integrity Constraints, *in* 'The 2nd Symposium on Mathematical Fundamentals of database Systems', Springer-Verlag, Lecture Notes in Computer Science, No 364, pp. 347–366.

NASSIF, R., QIU, Y. & ZHU, J. (1991), Extending the Object-Oriented Paradigm to Support Relationships and Constraints, *in* 'Object-Oriented Databases: Analysis, Design & Construction, IFIP DS-4', pp. 305–329.

PECKHAM, J., MARYANSKI, F., BESHERS, G., CHAPMAN, H. & DEMURJIAN, S. (1989), Constraint Based Analysis of Database Update Propagation, *in* 'International Conference on Information Systems, ICIS'89', Boston, Massachussets, pp. 9–18.

SU, S. & ALASHQUR, A. (1991), A Pattern-Based Constraint Specification Language for Object-Oriented Databases, *in* 'COMPCON Spring 91', San Francisco, California, pp. 522–531.

WORBOYS, M. (1991), Database Specification using Transaction Sets, *in* 'International Workshop on Specifications of Database Systems', Glasgow, UK, pp. 300–311.

BIOGRAPHY

Youssef Lahlou graduated from the "Institut d'Informatique d'Entreprise", Evry-France, as an "Ingénieur" in 1991. He then got his Ph. D. from the Henri Poincaré University, Nancy-France, in 1996. His domains of interest cover databases, information retrieval, object oriented design and digital libraries. He is currently on a post-doctoral position as an "ERCIM fellow" in GMD, Bonn-Germany, within the "Digital Library Initiative" of the European Research Consortium for Informatics and Mathematics (ERCIM).