Integrity Independence in Object-Oriented Database Systems

Hussien Oakasha and Gunter Saake

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

1 Introduction

Applying the feature of *integrity independence* means that we can modify constraints without modifying and hence recompiling updating transactions and applications. Implicit constraints of OODBS are a superset for almost all constraints that are considered as explicit for other data models, in particular for the relational model [JQ92,GJ91]. However, the principle of integrity independence is still missing in OODBS. Following one basic concept of the object-oriented paradigm, *encapsulation*, the explicit constraints are maintained through encoding them in methods.

The aim of this short paper is to present an approach for implementing the feature of integrity independence formally and in terms of basic OO data model concepts only. That is, we will consider the problem of how state constraints can be compiled, stored and manipulated using OO data model notions and operations. In this paper we use a generic OO data model that has basic features of the OODBMS manifesto presented in [ABD+89]. In addition, we assume that the inverse relationship is maintained by the model automatically. Examples of such a model are O_2 [BDK92] and GOM [KM94]. Here we follow the notations of O_2.

Integrity constraints are range-restricted wff in prenex normal form. Due to space limitation and for sake of exposition, we only consider local and inter-class constraints, that is constraints on relationships between different classes. We also assume that relationships between classes are one-to-one. Thus the quantifier structure of constraint will only contains universally quantified variables. As an example database we will use the following example.

Example 1. The database represents information concerning persons and their vehicles. The relationship between class $Person$ and class $Vehicle$ is one-to-one. This relationship is implemented through the attribute car in class $Person$ and the attribute $owner$ in class $Vehicle$. Thus car is an inverse attribute for $owner$ and vice versa. In other word, for a person p and vehicle c, $p.car = c \iff c.owner = p$. In every class there is an attribute named $constraints$ that will be described later (see definitions 8 and 10).
Classes:
Class $Person$ $[salary : numeric, age : numeric, car : Vehicle, constraints : H_{Person}]$
Class $Vehicle$ $[model : string, owner : Person, constraints : H_{Vehicle}]$
Constraints:
$W_1 : (\forall p : Person)(p.age \geq 40 \rightarrow p.salary \geq 2000)$.
$W_2 : (\forall p : Person)(\forall c : Vehicle)(p.car = c \wedge c.model = \text{“}X\text{”} \rightarrow p.age \geq 40)$.

2 Constraints

The presence of inverse relationships in our model leads to the fact that there is more than one form for specifying a given constraint. Thus for a given constraint there is a set of equivalent wffs that represent it.

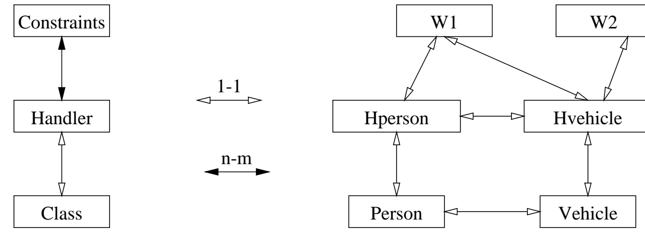

Fig. 1. The Relationship among Classes, Handlers and Constraints

Definition 2 (Equivalent Forms of a Constraint). Given a constraint W in prenex normal form $(\forall o_1 : C_n) \ldots (\forall o_n : C_n)(M)$, where $\{C_1, \ldots, C_n\}$ is a set of distinct classes and M is an open wff over attributes of objects o_i. The form of W w.r.t. the class C_i is a wff, M_{c_i}, obtained by applying the following steps:

- replacing every occurrence of o_j , $i \neq j$, in M by $o_i.p$, where p is a path expression that has as a destination an object of type C_j; and
- then replacing every term of form $exp = exp$ or $o_i.exp = o_i$ by $exp \neq nil$ and $o_i.exp \neq nil$ respectively.

The set of all forms of the constraint W will be denoted by $M(W)$. For every form $m \in M(W)$, the evaluation of m will be denoted by $|m|$.

Example 3. Consider the constraint W_2.
$M_{Person} : (p.car \neq nil \wedge p.car.model = \text{“}X\text{”} \rightarrow p.age \geq 40)$.
$M_{Vehicle} : (c.owner.car \neq nil \wedge c.owner.car.model = \text{“}X\text{”} \rightarrow c.owner.age \geq 40)$.

To separate constraints specifications from object schema we will consider constraints as first-class citizen. That is for every constraint W there is a class K_W that represents all information necessary for evaluating the constraint during running of application programs.

Definition 4 (Constraint Classes: The structure). Given a constraint W. The class of W is denoted by K_W. The structure of K_W includes the following attributes:

- For every class C subject to W there is an attribute c in K_W of type C,
- an attribute named “$form$ ”of type $M(W)$, and
- an attribute named “*valid* ”of type boolean.

Example 5. For the constraint W_2. K_{W_2} has the following structure:
Constraint K_{W_2} $[person : Person,\ vehicle : Vehicle,\ form : M(W),\ valid : Bool]$.
Where the type $M(W)$ can be defined as $M(W) = \{M_{Person}\} \cup \{M_{Vehicle}\}$.

Definition 6 (Constraint Classes: Operations). The dynamic part of constraint class K_W consists of the following operations:

- For every class C subject to constraint W there is an operation with the signature: $c(o : C)$. $c(o)$ takes an object, o, of type C as an argument and assigns this object to the attribute c in K_W. In addition it assigns the form M_C to $form$ and $\boldsymbol{T}$ to the attribute *valid*.
- An operation named *evaluate* to evaluate the constraint. *evaluate*() sets the evaluation of $form$ to the attribute *valid*.
- An operation named $inform$. $inform()$ informs users upon violation of the constraint W.

Example 7. The behavior part of constraint class K_{W_2} consists of the following operations.

$person(o : Person)\{$ $valid \leftarrow \boldsymbol{T};$ $person \leftarrow o;$ $form \leftarrow M_{Person}.\}$	$vehicle(o : Vehicle)\{$ $valid \leftarrow \boldsymbol{T};$ $vehicle \leftarrow o;$ $form \leftarrow M_{Vehicle}.\}$	$evaluate()\{$ $valid \leftarrow \|form\|.\}$	$inform()\{$ **If** $\neg valid$ Inform User **EndIf**.}

3 Constraint Handler

Assume that the class C is subject to constraints $W_1 \ldots W_n$. Representing the relationship "subject to "between C and $K_{W_1} \ldots K_{W_n}$ directly, first will complicate the structure of C and second modifying constraints leads to recompile the schema of C and that is what we want to avoid. Thus we will define another class H_C that gathers all constraint classes $K_{W_1} \ldots K_{W_n}$ and makes it responsible for controlling all aspects of constraints $W_1 \ldots W_n$ such as evaluation. This motivates the definition of *constraint handlers.*

Definition 8 (Constraint Handler: The structure). Given the class C. The constraint handler for C is a class, denoted by H_C, that has the following attributes:

- An attribute named "*object*"of type C.
- An attribute named "*constr*"of type $[con_1 : K_{W_{i1}}, \ldots con_m : K_{W_{im}}]$, where $K_{W_{i1}}, \ldots K_{W_{im}}$ are constraint classes over class C.
- For every attribute A that represents a relationship between C and a class C' there is an attribute named "A "of type $H_{C'}$.
- An attribute named "*checked* "of type boolean.

Example 9. The constraint handlers for $Person$ and $Vehicles$ are respectively the following classes:

$\boldsymbol{Class}\ H_{Person}\ [object : Person,\ constr : [con1 : K_{W_1}, con2 : K_{W_2}],\ car : H_{Vehicle},\ checked : Bool]$
$\boldsymbol{Class}\ H_{Vehicle}\ [object : Vehicle,\ constr : [con1 : K_{W_2}],\ owner : H_{Person},\ checked : Bool]$

Definition 10 (Constraint Handler: Operations). The dynamic part of constraint class H_C consists of the following operations:

- For every attribute A in class C of type that is different from class names there is an operation named "$A()$ ". $A()$ executes a sequence of statements of forms $constr.con_i.evaluate()$. Where every con_i is a constraint class that could be effected by modifying attribute A.
- For every attribute B in class C of type, C' , where C' is a class name, there is an operation named "$B()$ ". $B()$ assigns the value of attribute B in class C to $B.object$ of $H_{C'}$. Formally $self.B.object \leftarrow self.object.B$, here self denotes the objects of class H_C.
- A method named "*Instantiation*". $Instantiation()$, sets the attribute *checked* to $\boldsymbol{F}$. Then, for every attribute A in class H_C of type $H_{C'}$, $Instantiation()$ tests whether $A.object$ is nil. If it is nil, $Instantiation()$ assigns $object.A$ to $A.object$ and then execute $A.Instantiation()$. Finally, for every attribute con in $constraints$ $Instantiation()$ calls $con.c(object)$.
- A method named "*check* ". $check()$ executes a statements of of the form

  If $\neg constraints.con_i.valid$
  $constr.con_i.Inform()$ **EndIf**;

  for every con_i in $constr$. Then it executes a statement of form $A.check()$ for every attribute of type constraints handler. Finally, $check()$ assigns $\boldsymbol{F}$ to the *checked* attribute.

Example 11. The operations of constrain handler H_{Person} are:

```
Instantiation(){
checked ← F
 If object.car.constraints = nil
    car.object ← object.car;
    car.Instantiation();
 EndIf ;
For con in constr Do
      con.person(object)
EndDo. }
```

```
check(){
If ¬checked
   checked ← T;
   For con in constr Do
         If ¬con.valid
            con.Inform()
         EndIf;
   EndDo;
   car.check();
EndIf;
checked ← F . }
```

```
salary() {
constr.con1.evaluate().}

age() {
constr.con1.evaluate();
const.con2.evaluate().}


car() {
car.object ← object.car.}
```

4 Constraint Maintenance

Our description for how constraints may be specified separately from object schema and how this can be handled is completed. It remains to describe how constraints can be maintained using our approach. The general idea can be summarized into the following steps. Given an object o to be updated, first, create an instance for every constraint to which o is subjected. This will be done by executing the statement $o.Instantiation()$. Second, every time an update is made to an attribute of o by a statement of form $o.p.a \leftarrow v$, where p, a and v are a path expression, an attribute and a value, respectively, the statement $o.constraints.p.a()$ is executed. By this statement the constraint that could be violated by modifying the attribute a is evaluated and its evaluation is stored in its attribute *valid*. Finally there are three places where constraints should be checked:

1. At the end of every method for object o that has a side effect. In this case, statement $o.constraints.check()$ is executed.
2. At the commit time of a transaction. In this case, statements $o.constraints.check()$ and then $o.constraints \leftarrow nil$ are executed. Here the *constraints* attribute is set to *nil* for otherwise constraints instance will be persistent.
3. Immediately before a statement of form $o \leftarrow o'$ where o' is an object of same type as o. In this case, statements $o.constraints.check()$ and then $o.constraints \leftarrow nil$ are executed immediately before $o \leftarrow o'$. Then the constraints must be instantiated again to reset the *valid* attribute of every constraint class to $\boldsymbol{T}$.

Example 12. $RaiseSalary$ is a short transaction. The intended meaning of $RaiseSalary$ is to increase salary of every person whose salary less than 1000.

$RaiseSalary(precent : numeric)\{$
(1) $Vperson$ new $Person$;
(2) **For** $Vperson \in \{o | o \in Person \wedge o.salary < 1000\}$ **Do**
(2.1) $Vperson.Instantiation()$;
(2.2) $Vperson.salary \leftarrow Vperson.salary * precent + Vperson.salary$;
(2.3) $Vperson.constraints.salary()$;
(2.4) $Vperson.constraints.check()$;
(2.5) $Vperson.constraints \leftarrow nil$;
EndDo.}

In step (1) a new object of type $Person$ is created and named $Vperson$. Step (2) is an iterator, that is, every object of type $Person$ satisfies the condition that exists between **For** and **Do**, steps

(2.1) to (2.5) are executed. All constraints relevant to the class *Person* are instantiated by step (2.1). In (2.2) the salary is increased, this may affect some constraints, therefore in step (2.3) the constraints affected by this modification are evaluated, the evaluation will be stored in *valid* attribute of every constraint. Since there are no more modification, constraints are checked. This is the task of step (2.4), every constraint instantiated by step (2.1) is checked by testing the valid attribute of that constraint. In this step the user will be informed for every violated constraint. Before executing steps (3.1) to (2.5) for another object, *constraints* attribute of *Vperson* is set to *nil*.

Example 13. The *NewPerson* method is considered here as one of the encapsulated methods of class *Person*. Thus *self* here denotes an object of *Person*. Here constraints are instantiated outside the *NewPerson* method as object is created out too. The value *Vsalary* is assigned to attribute salary *Salary*, this may affect some constraints. These constraints are evaluated by *self.Constraints.salary*. In the same way constraints affected by modifying *age* and *car* attributes are evaluated. Here constraints are checked at the end of the method but neither instantiated again nor deleted as long as this modification apply only to one object namely *self*.

$NewPerson(Vsalary, Vage, Vcar)\{$
$self.salary \leftarrow Vsalary;$
$self.constraints.salary();$
$self.age \leftarrow Vage;$
$self.constraints.age();$
$self.car \leftarrow Vcar;$
$self.constraints.car();$
$self.constraints.check().\}$

5 Conclusion and Future Works

In this short paper we presented an approach for implementing the feature of integrity independence formally and in terms of basic OO data model concepts only. We have shown how state constraints can be compiled, stored and manipulated using OO data model notions and operations. This is made under two restriction. The first one for relationships between classes, here we only consider one-one relationships. The other one is for constraints, here, only constraints that have some pattern are considered (see definition 2).

Currently we investigate how this approach can be generalized. We do that in several directions. The first and the most important one is considering more general types of constraints, for example key constraints. Second, putting restrictions on cardinality of relationship is not realistic thus this must be avoided. Third, for maintaining constraints we present a general method, subjects like efficiency of instantiation and evaluation of constraints must be considered. Finally our final goal is to apply this work for long duration transactions.

References

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

[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.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *International Conference On Very Large Data Bases*, pages 327–336, Hove, East Sussex, UK, September 1991. Morgan Kaufmann Publishers, Inc.

[JQ92] H. V. Jagadish and X. Qian. Integrity maintenance in an object-oriented database. In *International Conference On Very Large Data Bases*, pages 469–480, San Mateo, Ca., USA, August 1992. Morgan Kaufmann Publishers, Inc.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management*. Prentice Hall, Englewood Cliffs, NJ, 1994.