IQL(2): A Model with Ubiquitous Objects*

Serge Abiteboul (Serge.Abiteboul@inria.fr)
Cássio Souza dos Santos† (Cassio.Souza@inria.fr)

I.N.R.I.A., B.P. 105, 78153 Le Chesnay Cedex, France
tel: 1/39 63 56 79 fax: 1/39 63 53 30

February 1995

Abstract

Object-oriented databases have brought major improvements in data modeling by introducing notions such as inheritance or methods. Extensions in many directions are now considered with introductions of many concepts such as versions, views or roles. These features bring the risk of creating monster data models with a number of incompatible appendixes. We do not propose here any new extension or any novel concept. We show more modestly that many of these features can be formally and (we believe) cleanly combined in a coherent manner.

1 Introduction

We propose an extension of IQL [AK89], therefore the name[1] IQL(2), to encompass many new extensions to the core OODB models that have been considered separately in the past. The model is based on two not novel concepts: (i) contexts that are used to parameterized class and relation names; and (ii) views to define intensional data. This brings two kinds of ubiquity to objects, i.e., the same object may belong *really* or *virtually* to several classes at the same time. We propose a first-order language with static type-checking, under certain restrictions on the schemas. Most of the examples are given using a more convenient OQL-like syntax.

We briefly consider two technical issues: (i) quantification over contexts, and (ii) method resolution for ubiquitous objects. Quantification over contexts can be handled under some reasonable restrictions that we present. Uncontrolled ubiquity together with inheritance, leads to severe problems with respect to type checking and conflict resolution. We advocate here the use of strong restrictions so that standard resolution techniques can be used.

As illustrated by examples, the model captures in a coherent framework many features that have been considered separately in the past: (i) a model with objects, classes, inheritance, methods ala IQL or O_2 [BDK92]; (ii) a view mechanism ala O_2Views [SAD94]; (iii) a versioning mechanism with linear versions and also alternatives (see, e.g., [KC88]); (iv) a mechanism for objects with several roles [BD77, RS91] ala Fibonacci [ABGO93]; (v) the means of specifying distribution of data in several sites; (vi) a mechanism for data and schema updates (see, e.g., [Zic92]); (vii) specification of access rights (see, e.g., [RBKW91]).

*Partially supported by Esprit Project GoodStep.

†On leave from Departamento de Informática, Universidade Federal de Pernambuco, Brazil. Partially supported by CNPq grant number 200.803-92.1.

[1] No, Guido, this does not imply that there will be an IQL(3).

The paper is organized as follows. In Section 2, we introduce some notation and auxiliary concepts. A restricted form of the model (without views and inheritance) is presented in Section 3. The language is presented in Section 4. Section 5 deals with inheritance and Section 6 with views. The last section is a conclusion. Additional examples are given in Appendix A.

To conclude this section, we present in an example some of the features of the model.

Example 1.1 Consider a distributed database with two sites: Paris and Los Angeles. Paris and Los Angeles are two *contexts* of a unique database. Suppose that the database deals with persons, friends and researchers, i.e., we have classes *Person, Friend, Researcher*. Classes *Friend* and *Researcher* are subclasses of *Person* in both contexts. Let *Dupond* be an object. First, suppose that in Paris, *Dupond* is considered a friend, and in LA both a friend and a researcher, i.e., *Dupond* belongs to class $Friend(Paris)$, $Friend(LA)$ and $Researcher(LA)$. By inheritance, *Dupond* is also in classes $Person(Paris)$ and $Person(LA)$ (with possibly different behaviors in each).

Now, we may decide that the data on friends is recorded in LA. We therefore have a relation $Friends(LA)$, and see relation $Friends(Paris)$ as a view of $Friends(LA)$. This would mean that the store for *Dupond* is in LA and that *Dupond* is only *virtually* in class $Friends(Paris)$. This does not prevent *Dupond* from being *really* in $Researcher(LA)$ with a specific store there.

At one extreme, we may decide that one context is completely virtual and that no data is stored there. At another extreme, we can view the database as duplicated in contexts $Paris$ and LA. Each object has a store in Paris and one in Los Angeles. An update method on an object o in Paris context would modify the store in Paris. It may call immediately a method on object o in LA to propagate the change, or one may prefer to propagate updates in batches using a program that is called regularly. □

2 Preliminaries

In this section, we introduce some notation and some auxiliary concepts.

We consider the existence of the following pairwise disjoint and infinite countable sets:

1. **rel**: relation names $R_1, R_2, ...$
2. **class**: class names $C_1, C_2, ...$
3. **obj**: object identifiers (oid's) $o_1, o_2, ...$
4. **dom**: data values $d_1, d_2 ....$ The set **dom** is typically many sorted. It contains the sorts *int, real, bool, string* and a particular sort for context identifiers (*cid's*) that will be application dependent. The data sorts will be denoted $\mathbf{d_1}, \mathbf{d_2}, ...$ The values of sort $\mathbf{d_i}$ are $dom(\mathbf{d_i})$. The set of cid's will be denoted **cid**.

Given a set $\mathbf{O}$ of oid's, the set of values that one can construct is denoted $val(\mathbf{O})$:

1. $val(\mathbf{O})$ contains $\mathbf{O}$ and **dom**;
2. $val(\mathbf{O})$ is closed under tupling and finite setting. (Other constructors such as sequencing or multi-setting can be added in a straightforward manner and will not be considered here.)

The cid's will serve many purposes. If we take cid's in $[1..n]$, we model time versions. By organizing the cid's in a dag, we also model alternative versions. By taking cid's for instance in { London, Paris, LA, etc. }, we model distributed databases with the same object (with distinct repositories) possibly in many sites. By choosing cid's in { John, Peter, Max, etc. }, we model access rights for various users.

In practice, one may want to use cid's with a richer structure, i.e., use complex values or objects to denote contexts. For instance, in a versioned and distributed database, one would like the domain of cid's to be the set of pairs (timestamp,location). We ignore this aspect here since this would unnecessarily complicate the model, and view the cid's as atomic elements. Indeed, in most of the discussion, we assume that the domain **cid** of the cid's is an initial fragment of the integers. However, in examples, we sometimes use a richer structure for cid's.

We consider that the "names" of both the schema and the instance are indexed by the cid's. A class in our context is now $C(n)$ for some cid n, and a relation becomes $R(n)$. On the other hand, objects are not indexed by cid's. However, their values and behaviors depend on the roles that they are taking. For instance, a versioned object is the same object in all its different versions. Its value and behavior depend on the particular version that is considered.

Given a set $\mathbf{C}$ of classes and the set $\mathbf{cid}$ of cid's, $\mathbf{C}(\mathbf{cid})$ denotes $\mathbf{C} \times \mathbf{cid}$. Starting from sets $\mathbf{C}$, and $\mathbf{cid}$, the types $types(\mathbf{C}(\mathbf{cid}))$ are defined by the following abstract syntax:

$$\tau := \mathbf{d}_i \mid \mathbf{C}(\mathbf{cid}) \mid [A_1 : \tau, ..., A_n : \tau] \mid \{\tau\} \mid \tau + \tau \mid \perp$$

where $n \geq 0$, the A_i's are distinct and "+" is the union of types.

An *oid assignment* π is a mapping from $\mathbf{C}(\mathbf{cid})$ to $2^{\mathbf{obj}}_{fin}$ (the finite powerset of $\mathbf{obj}$). It gives the population of each class in each context. (Note that class populations are not required to be disjoint and objects may be explicitly in many different classes.) The set of oid's occurring in π is denoted $\mathbf{O}$.

The semantics of types is given *with respect to an oid assignment* π:

1. for each $\mathbf{d}_i$, $[\![\mathbf{d}_i]\!]_\pi = dom(\mathbf{d}_i)$;
2. $[\![C(n)]\!]_\pi = \pi(C(n))$;
3. finite setting and tupling are standard;
4. $[\![\tau_1 + \tau_2]\!]_\pi = [\![\tau_1]\!]_\pi \cup [\![\tau_2]\!]_\pi$;
5. $[\![\perp]\!]_\pi = \emptyset$.

Given an oid assignment π and the corresponding finite set $\mathbf{O}$ of objects, a *value assignment* ν is a mapping from $\mathbf{O} \times \mathbf{C}(\mathbf{cid})$ to $val(\mathbf{O})$; i.e., it associates to a triple (object,class,cid), a value.

Remark 2.1 Observe that the value of an object is depending on two parameters: the context and the class. Suppose that we have two contexts *business* and *personal*, modeling respectively my business phone-book and my private one. Suppose that we have two classes *Friend* and *Researcher*. Suppose that Jones is a friend and a researcher. Then, I may have phone informations for Jones in both contexts and in both classes. The fact that some data is stored and some may be derived is irrelevant (so far). □

3 Database Schema and Instance

We define the schemas and the instances. We ignore first an important aspect, namely, the specification of the "virtual database" (Δ below), which is the topic of Sections 5 (inheritance) and 6 (views).

Definition 3.1 A *database schema* $\mathcal{S}$ is a tuple $(\mathbf{R}, \mathbf{C}, \mathbf{cid}, \mathbf{T}, \Delta)$ where: (i) $\mathbf{R}$, $\mathbf{C}$, are finite sets of relation and class names; (ii) $\mathbf{cid}$ is the finite set of contexts;

(iii) $\mathbf{T} \quad : \quad \mathbf{R}(\mathbf{cid}) \cup \mathbf{C}(\mathbf{cid}) \quad \rightarrow \quad types(\mathbf{C}(\mathbf{cid}))$;

(iv) Δ is a view program to be defined later.

This is a conservative extension of IQL. First, $\mathbf{R}$ is the set of names of roots of persistence, $\mathbf{C}$ the set of class names, $\mathbf{cid}$ (is new and) is the set of contexts, $\mathbf{T}$ is the typing constraint. In IQL, the view program Δ is simply the inheritance hierarchy since there is no other mechanism for virtual data there.

It is important to observe that we associate types to pairs involving a name (relation or class) and a cid. This captures the fact that the same name may have different types in different contexts. For instance, if the contexts are versions, the type of a class is allowed to evolve in time. Observe also that the type of a class or a relation in some context may refer to a class in another context.

Example 3.2 We consider a database context *Global* that is the integration of the two local database contexts, *LA* and *Paris*.

The schema is as follows:
Let $\mathbf{R} = \{R_p, R_{la}, R_g\}$, $\mathbf{C} = \{Employee\}$, $\mathbf{cid} = \{Paris, LA, Global\}$ and $\mathbf{T}$ be defined by:

$class\ Employee(Paris) : [Name : string, Téléphone : integer]$
$class\ Employee(LA) : [Name : string, Phone : integer]$
$class\ Employee(Global) : [Name : string, Phone : integer, Téléphone : integer]$

$type\ R_p(Paris) : \{Employee(Paris)\}$
$type\ R_{la}(LA) : \{Employee(LA)\}$
$type\ R_g(Global) : \{Employee(Global)\}$

Observe that the type of R_p is only defined in the context of *Paris*. This should be understood as R_p does not exist in the contexts of *LA* and *Global*. (Similarly, for R_{la}, R_g.) □

We now consider instances.

Definition 3.3 An *instance* $\mathcal{I}$ of schema $\mathcal{S}$ is a triple (π, ρ, ν) with

1. an oid assignment $\pi : \mathbf{C}(\mathbf{cid}) \rightarrow 2^{\mathbf{O}}$;
2. a relation mapping $\rho : \mathbf{R}(\mathbf{cid}) \rightarrow 2^{val(\mathbf{O})}$;
3. a value assignment ν : for each o, C, n such that $o \in \pi(C(n))$, ν maps the pair $(o, C(n))$ to a value in $val(\mathbf{O})$;

where $\mathbf{O}$ is the set of oid's occurring in π.

Ignoring the view mapping, we now specify the notion of well-formed instance:

Definition 3.4 Let (ν, ρ, π) be an instance over a schema $\mathcal{S}$. The instance is *well-formed* if the following typing constraints are satisfied:

1. for each R, n, $\rho(R(n)) \subseteq [\![\mathbf{T}(R(n))]\!]_\pi$;
2. for each o, C, n, $o \in \pi(C(n))$, $\nu(o, C(n)) \in [\![\mathbf{T}(C(n))]\!]_\pi$.

Two well-formed instances are given in Figure 1. Intuitively, instance I_2 is obtained from instance I_1 by deriving some new data.

Instance I_1	*Instance I_2*
$\pi(Employee(Paris)) = \{o_1, o_2\}$	$\pi(Employee(Paris)) = \{o_1, o_2\}$
$\pi(Employee(LA)) = \{o_1\}$	$\pi(Employee(LA)) = \{o_1\}$
$\pi(Employee(Global)) = \emptyset$	$\pi(Employee(Global)) = \{o_1, o_2\}$
$\rho(R_p(Paris)) = \{o_1, o_2\}$	$\rho(R_p(Paris)) = \{o_1, o_2\}$
$\rho(R_l(LA)) = \{o_1\}$	$\rho(R_l(LA)) = \{o_1\}$
$\rho(R_g(Global)) = \emptyset$	$\rho(R_g(Global)) = \{o_1, o_2\}$
$\nu(o_1, Employee(Paris)) = [D., 55\ 37]$	$\nu(o_1, Employee(Paris)) = [D., 55\ 37]$
$\nu(o_1, Employee(LA)) = [D., 11]$	$\nu(o_1, Employee(LA)) = [D., 11]$
$\nu(o_2, Employee(Paris)) = [L., 53\ 30]$	$\nu(o_2, Employee(Paris)) = [L., 53\ 30]$
	$\nu(o_1, Employee(Global)) = [D., 11, 55\ 37]$
	$\nu(o_2, Employee(Global)) = [L., 0, 53\ 30]$

Figure 1: Two instances

4 A Query Language

We now define a many-sorted first-order calculus then give examples of queries in an OQL-like syntax. (As in IQL, we could have used here a rule based language but since recursion is not important here, we prefer to focus on a simpler language not to obscure the issue.) We first consider "fixed contexts" in the sense that we disallow quantifications over cid's.

A Fixed Context Calculus

The calculus is defined as follows:

Terms The *terms* of the calculus are:

1. d for each d in **dom**;
2. $R(n)$ for R in $\mathbf{R}$ and n in **cid** ($R(n)$ denotes the value of relation R in context n);
3. variables x_τ where the type τ does not refer to the sort **cid** (the type is omitted when clear from the context);
4. constructed terms with tupling ($[A_1 : t_1, ..., A_n : t_n]$), setting ($\{t_1, ..., t_n\}$), projection ($t.A$ for A an attribute), and dereferencing ($*t$ for t denoting an object).

The sorts of terms are defined in the straightforward manner.

Formulas, queries: *Atoms* are $t = t'$, $t \in t'$ for t, t' terms with compatible types, or $x \equiv x'$ where x, x' are of resp. sorts $C(n), C'(m)$. (This is interpreted as x and x' are the same object in different contexts.) *Formulas* are atoms, or $L \vee L'$, $L \wedge L'$, $L \Rightarrow L'$, $\neg L$, $\exists x_\tau(L)$ or $\forall x_\tau(L)$ where L, L' are formulas. A *query* is an expression of the form $\{x \mid \varphi\}$ where φ is a formula with only free variable x.

Range-restriction As standard, we restrict our attention to range-restricted formulas and queries. The range-restriction we adopt here is standard. From this point of view, the only novelty is the use of $\equiv$ that behaves exactly like equality for range-restriction. Contexts play no role for range-restriction since we assumed they are constant.

From a language viewpoint, the only (relative) novelty is the use of $\equiv$. We illustrate it with an example. Suppose that the cid's are timestamps and that the last two versions are denoted by the constants *previous* and *now*. Let *Persons* be a set of objects of class *Person*. We can obtain the phone number of persons that have not changed phone number since last version:

$$\{P.phone \mid \exists P' \in Persons(previous)(P \in Persons(now) \wedge P \equiv P' \wedge P.phone = P'.phone)\};$$

or using an OQL-like syntax:

```
select  P.phone
from    P in Persons(now)
where   P.phone in
        select   P'.phone
        from     P' in Persons(previous)
        where    P' ≡ P.
```

We could express the same query in a simpler manner if either (a) a field *previous* (possibly virtual – see below) contains the previous state of each object or (b) using casting:

```
select  P.phone                        select P.Phone
from    P in Persons(now)              from   P in Persons(now)
where   P.previous.phone = P.phone     where P.phone = P@Person(previous).phone
```

where $P@Persons(previous)$ denotes the casting of P to the same object in class $Person(previous)$. Such casting can be viewed as syntactic sugaring. Another form of syntactic sugaring would be to permit to test whether an object is also in some different contexts. This allows us to rephrase (more carefully) the above query:

```
select  P.Phone
from    P in Persons(now)
where   P is_also Person(previous) and P.phone = P@Person(previous).phone
```

Remark 4.1 To see a more complicated example with "structured" contexts, suppose that we are in a versioned database with one context for private data and one for professional one. To obtain the actual home phone numbers of friends who worked on OQL in 1990, we use:

```
select  P.phone
from    P in Persons(private,now), P' in Persons(prof,1990)
where   "OQL" in P'.works_on and P ≡ P'
```

where the domain of cid's is a set of pairs (context,timestamp). □

Quantifying over Contexts

We start with two examples and then consider some difficulties that are raised.

First, suppose that **cid** consists of two contexts, namely LA and Paris, and that we want to modify the salaries of employees by taking the maximum of the salaries in the two contexts. We may use one of the following programs:

Program 1

```
update E.salary = E'.salary
from   E in Emp(Paris), E' in Emp(LA)
where  E is E' and E.salary < E'.salary
update E.salary = E'.salary
from   E in Emp(LA), E' in Emp(Paris)
where  E ≡ E' and E.salary < E'.salary
```

Program 2

```
update E.salary = E'.salary
from   Site1,Site2 in { Paris,LA},
       E in Emp(Site1), E' in Emp(Site2)
where  E ≡ E' and E.salary < E'.salary
```

Observe that the second one, although clearly more desirable (imagine 20 sites!), uses cid variables, i.e., Site1, Site2, for specifying the context (whereas LA for instance is a constant). This is a quantification over some contexts.

From the example, it is clearly convenient to be able to quantify over contexts. However, this complicates the type checking of programs as illustrated by the following example.

Suppose that the context is [1..now] and that in Version 15, we added an attribute to class Person, e.g., an email address. Consider the following queries asking for the name of persons such that their stored value has been modified at least once (since Version 17):

Query 1	Query 2
select = P.Name from N in Contexts, P in Persons(N), P' in Persons(now) where P is P' and not ($*$P = $*$P')	select P.Name from N in Contexts, P in Persons(N), P' in Persons(now) where P is P' and not ($*$P = $*$P') and N > 17

where *Contexts* is a relation containing the set of valid contexts.

Recall that "$*$" denotes dereferencing. Observe that Query 1 should raise an error since the type of a person now and say in Version 14 are different. The sorts of the values for a person now and at time 14 are not compatible and $*P = *P'$ is incorrect. On the other hand, Query 2 should be acceptable as far as we test for $N > 17$ before testing other conditions. However, an issue also of Query 2 is type checking since because of the schema update, we cannot assign a type to P. A first solution is to use dynamic type checking. Another one is to require that the quantification over N be outermost and apply the restrictions on context variables during type checking (i.e., at compile time).

More formally, we require the formula to be of the form:

$$Q_1 x_1 ... Q_m x_m (\varphi(x_1, ..., x_m) \ \theta \ \psi)$$

where $Q_1, ..., Q_m$ are quantifications over contexts, φ is a (range-restricted) formula that has no quantification over contexts, its only free-variables are contexts (φ restricts the range of the contexts), θ is $\wedge$ or $\Rightarrow$ and ψ contains no quantification over contexts.

Query 2 can be expressed in this form:

$$\{P.Name \mid \exists N((Context(N) \wedge N > 17) \wedge \exists P, P'(Persons(N)(P) \wedge Persons(now)(P') \wedge P \equiv P'))$$

Intuitively, this suggests the following evaluation. First φ is evaluated. Since it has no quantification over context, its evaluation raises no issue. Then, based on the results of φ, the global query is transformed into a boolean combination of queries with no quantification over context. Each of these queries can be typed checked and executed separately.

Observe that this form is restrictive since it does not allow expressing queries of the form $\{\ldots \mid \forall x \exists n ...\}$ where the value of context n depends on x. It is possible (although rather intricate) to find natural examples of such queries (for instance, see the example above where the field *previous* contains the previous state of each object).

5 Inheritance

In this section, we consider the addition of an inheritance relationship to the schema. Since classes in contexts play the role of standard classes, we need to consider statements such as $C(n) \ isa \ C'(m)$ that possibly relates two distinct contexts. We assume that the inheritance hierarchy is a dag.

A major issue in the presence of inheritance is method resolution. To simplify the model, we did not consider methods. Although we will not do it here formally, methods can be introduced very simply in the model as in [AK91]. In the following discussion on inheritance, we consider that methods are attached to classes in the style of, say [BDK92] or [AKRW92].

The semantics of an inheritance statement $C(n)$ *isa* $C'(m)$ is that each object in a class $C(n)$ (e.g., $o \in \pi(C(n))$) is also implicitly in class $C'(m)$. The value of this object in class $C'(m)$ is obtained by "coercing" its value in class $C(n)$. This imposes a constraint on types of classes related via inheritance (inclusion polymorphism semantics). The types have to be reconsidered to include this notion of inheritance as done for instance in IQL (e.g., a tuple of type $[A : int, B : int, C : int]$ is also of type $[A : int, B : int]$. Since this is standard, we do not insist on it here.

From a formal viewpoint, starting from an instance (π, ρ, ν), inheritance specifies a new instance (π^*, ρ, ν^*), which can be seen as a virtual, i.e., derived, instance. (See next section on views.) The typing constraints that are imposed on the instance now become constraints on the derived instance:

1. for each R, n, $\rho(R(n)) \subseteq [\![\mathbf{T}(R(n))]\!]_{\pi^*}$;
2. for each o, C, n, $o \in \pi^*(C(n))$, $\nu^*(o, C(n)) \in [\![\mathbf{T}(C(n))]\!]_{\pi^*}$.

Inheritance is complicated by the fact that the same object lives in several classes, a problem treated in [ABGO93] and that we reconsider in our setting.

First, we introduce radical restrictions that lead to standard resolution and probably suffice for many applications. Then, we analyze the general case which requires a more complex resolution. Finally, we consider a general restriction that is somewhat in between these two extremes.

A Simple World

The simple world is based on the following principles:

separate contexts: all inheritance statements are limited to a single context (i.e., $C(n)$ *isa* $C'(m)$ implies $n = m$). There is one class hierarchy defined for each context.

separate roles: the disjoint oid assignment is enforced in a given context, i.e., the same object cannot be *explicitly* in two distinct classes of the same context (it can however be in two distinct classes *implicitly*, i.e., through inheritance).

Under these restrictions, resolution is standard (as in [BDK92] or in [AKRW92]). Part of the limitations brought by this approach will be removed when we introduce views.

Let us now consider the general case.

A Complex World

Consider the inheritance hierarchy of Figure 2. (We assume in this example the existence of a single context.) A call to method m on an object o'' in class C'' generates a conflict (multiple-inheritance) that can be statically detected. This is standard.

New kinds of conflicts may arise due to multiple roles. For instance consider an object o'_{12} living both in class $C1'$ and $C2'$. A call to m on o'_{12} may be ambiguous. Similarly, an access to attribute A for this object may be ambiguous.

We now have to be somewhat more precise. As mentioned before, we always access an object o for a role $C(n)$, i.e., we address the $C(n)$ interface of the object. Consider accessing o in some class $C(n)$. Suppose, object o is explicitly exactly in subclasses $C_1(n_1), ..., C_i(n_i)$ of $C(n)$. Two cases arise:

1. we are accessing some attribute A. This is legal if A is an attribute in exactly one of the $C_j(n_j)$.

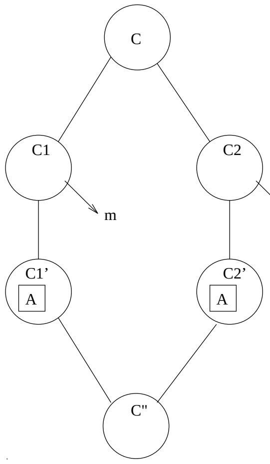

Figure 2: Inheritance and Conflicts

2. we access some method m. This is legal if

   - for some $C_j(n_j)$ $(j : 1..i)$, the resolution[2] of m in $C_j(n_j)$ is defined and is some class C'; and
   - for each $C_k(n_k)$ $(k : 1..i)$, the resolution of m in $C_k(n_k)$ is also C' or is not defined.

Multiple roles do complicate a lot the issue. Consider a class $C(n)$ with m subclasses. Then a variable of class $C(n)$ may denote an object o such that the set of subclasses of $C(n)$ where o is explicitly, may be any of the 2^m subsets of subclasses of $C(n)$. This leads to two important issues:

Problem (1): At run time, given an object o and a role $C(n)$ for this object, find fast the store for some attribute A and the code m for a method m.

Problem (2): At compile time, statically type check a program.

Both will be time consuming. Both can be simplified if we specify a *compatibility relation* γ that specifies where objects can be concurrently explicitly. More precisely, γ is an equivalence relation over $\mathbf{C(cid)}$, and $C(n)\gamma C'(m)$ indicates that an object may belong explicitly to both classes concurrently, so that multiple instantiation is constrained to classes in the same partition w.r.t. to γ. Type checking can be eased if, in addition, we make γ antisymmetric by constraining types of classes related by γ to be comparable w.r.t. to standard subtyping. This would define a role hierarchy, but we adopt a more general approach where role hierarchies can be defined, if necessary, through a view.

To see an example, consider a database of boats and airplanes with three classes, *Boat, AirPlane, Vehicle* and the schema:

class Boat : [Name : string, Price : integer, Propeller : string] isa Vehicle
AirPlane : [Name : string, Price : integer, Speed : integer] isa Vehicle
Vehicle : [Name : string, Price : integer]

If we know that the compatibility relation is empty, an access to the price of a vehicle is legal. Otherwise, there is a potential conflict since the same object may be in classes AirPlane and Boat explicitly.

The use of γ is investigated next.

[2]The resolution of m in some class $C_j(n_j)$ is the unique smallest super class of $C_j(n_j)$ where m has an explicit definition. This is unique since we disallow multi-inheritance conflict.

A Trade-off

It is standard to prohibit (or at least control) multiple-inheritance in the context of single-roles. We now add a condition to handle multiple roles.

A schema is *strict* if for each $C(n), C'(m)$, such that $C(n)\gamma C'(m)$ and $C(n), C'(m)$ are not comparable in the isa hierarchy, there is no $C''(p)$ such that $C(n)$ and $C'(m)$ are both subclasses of $C''(p)$ (i.e., $C(n)$ and $C'(m)$ have no common ancestor).

For *strict* schemas, the resolution issues above disappear, i.e., it is easy to see that for each object o and role $C(n)$, this leads to standard resolution for o in the unique class below $C(n)$ where it belongs explicitly. This leads to resolution with a parameter, the class $C(n)$ (i.e., Problem (1) disappears). For non-strict schemas, we can adopt multi-attribute resolution (to solve Problem (2) and techniques such as multi-attribute dispatch tables can be used [AGS94] (to solve Problem (1)).

6 Views

In the previous section, we already considered the specification of view mappings, but we restricted our attention to a special class of view mappings related to inheritance only. In this section, we use the entire power of the first-order language of the previous section to define view mappings.

A view program allows to specify from the value of the database composed of explicit information (instance (π, ρ, ν)), a well-formed virtual database (instance (π^*, ρ^*, ν^*) below).

Queries are first used to populate classes and relations as in:

$$\begin{array}{lcl} Employee(Global) & \sqsupseteq & \{x \mid Employee(Paris)\} \\ Employee(Global) & \sqsupseteq & \{x \mid Employee(LA)\} \\ R_g(Global) & \sqsupseteq & \{x@Employee(Global) \mid x \in R_p(Paris)\} \\ R_g(Global) & \sqsupseteq & \{x@Employee(Global) \mid x \in R_{la}(LA)\} \end{array}$$

We use two queries to define $Employee(Global)$ since a single one would be incorrectly typed.

Note also that the above definition does not not prevent the class $Employee(Global)$ to have explicitly objects in it.

Remark 6.1 In the presentation so far, we have implicitly assumed that the extensions of base classes are given and used to compute the extensions of derived classes. It is argued in [SAD94] that in many applications, it is not desirable to maintain the extensions of classes. Furthermore, some systems (such as O_2) do not provide extensions for base classes, and it would be unnatural to maintain that of derived classes in such context. If class extensions are not maintained, the definition of $Employee(Global)$ is not necessary and can be viewed as "derived". □

Using such rules, it is easy to specify the values of π^* and ρ^*. For the specification of ν^*, we can use two approaches.

In an explicit manner, we can specify or enrich the value of each object in its new class with rules of the form:

$$\begin{array}{l} var\ x : Employee(Global), x' : Employee(LA) \\ define\ x.phone = unique\{x'.phone \mid x' \equiv x\} \end{array}$$

This can also be achieved implicitly. We assume that *by default*, the values of objects are transmitted via derivations. For instance, if an object is in $Employee(Global)$ because of its presence in $Employee(LA)$, then it "inherits" its structure from that of the employee in LA. This implies some constraints on the types that are similar to constraints on types in presence of inheritance. (Recall that inheritance is just a special case of view.)

A problem is that the presence of an object in some class $C(n)$ may have its origin in the presence of the object in more than one other classes. For instance, an object may be in $Employee(Global)$ because it belongs to $Employee(Paris)$ and also because it belongs to $Employee(LA)$.

In such cases, the new value is obtained (a) by merging the values associated to the originating object/context pairs, and (b) projecting (casting) to the type that is expected. More precisely, suppose that we define the population of class C in context n as the union of φ_i where for each i, φ_i returns a set of objects of type $C_i(n_i)$. Then the value of an object o for $C(n)$ is defined by:

$$\nu(o, C(n)) = \Pi_T(\bowtie \{\nu(o, C_i(n_i)) \mid o \in \varphi_i\})$$

where merge ($\bowtie$) and projection (Π) are defined next.

Definition 6.2 The *merge* of two data values is defined by:

1. $v \bowtie v = v$ for each v;
2. if t_1, t_2 are tuples, $t_1 \bowtie t_2$ is the tuple t (if it exists) such that for each attribute A of t_1 and t_2, $t(A) = t_1(A) \bowtie t_2(A)$; and for each $i, j, j \neq i$, if t_i has attribute A and not t_j, $t(A) = t_i(A)$; t has no other attribute;
3. otherwise $v \bowtie v'$ is undefined.

Observe that two tuples with two non-merge-able values (e.g., integer 4 and 5) for the same attribute, are not merge-able. This does not prevent for instance an object o to have two distinct values, say 4 and 5, in two distinct classes. On the other hand, this cannot happen (in a correct instance) if these two versions of the same object are merged in a unique class.

The *projection* of a value on a type τ (given an oid assignment π^*) is defined recursively as follows:

1. if τ is $C(n)$ and $v = o$ is in $\pi^*(C(n))$, then $\Pi_\tau(v)$ is o;
2. if $\tau = [A_1 : \tau_1, ..., A_m : \tau_m]$ and $v = [A_1 : v_1, ..., A_n : v_n]$ for $m \leq n$ and for each $i \leq m$, $\Pi_{\tau_i}(v_i)$ is defined, then $\Pi_\tau(v) = [A_1 : \Pi_{\tau_1}(v_1), ..., A_m : \Pi_{\tau_m}(v_m)]$;
3. if $\tau = \tau_1 + \tau_2$ and either (i) $\Pi_{\tau_1}(v)$ or $\Pi_{\tau_2}(v)$ is defined and equal to v' but not both; or (ii) they are both defined and equal to v', then $\Pi_\tau(v) = v'$;
4. otherwise, $\Pi_\tau(v)$ is undefined.

To conclude this section on views, observe that we have two ways for an object to be virtually in a class. One is by inheritance and the other one is by the view mechanism. We advocated a strict policy for handling inheritance to simplify the treatment of inheritance conflicts. The view mechanism is handled differently. It may be more liberal at the price of being more costly.

7 Conclusion

In this paper, we have presented a model with many features that are usually considered separately. Our discussion on methods has been quite brief but we believe we covered the main issue, method resolution. Our treatment of views has also been rather short and many features of [SAD94] such as imaginary objects were not considered here. However, they would only have made more complicated the model at the cost of clarity and do not present any new difficulties.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 39–51, 1993.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Int. Conf. on OOPSLA*, Portland, October 1994. ACM.

[AK89] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 159–173, 1989. to appear in *J. ACM*.

[AK91] S. Abiteboul and P. Kanellakis. The two facets of object-oriented data models. *IEEE Data Engineering Bulletin*, 15:2:3–8, 1991. special issue edited by R. Agrawal.

[AKRW92] S. Abiteboul, P. Kanellakis, S. Ramaswamy, and E. Waller. Method schemas. Technical Report CS-92-33, Brown University, 1992. (An earlier version appeared in Proceedings 9th ACM PODS, 1990).

[BD77] C.W. Bachman and M. Daya. The role concept in data models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 464–476. Morgan Kaufmann, 1977.

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

[KC88] W. Kim and H.T. Chou. Versions of schema for object-oriented databases. In *Proc. of Intl. Conf. on Very Large Data Bases*. Morgan Kaufmann, 1988.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Trans. on Database Systems*, 16:1:88–131, 1991.

[RS91] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple independent roles. In *Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 298–307, 1991.

[SAD94] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT, Cambridge*, 1994.

[Su91] J. Su. Dynamic constraints and object migration. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 233–242. Morgan Kaufmann, 1991.

[Zic92] R. Zicari. A framework for schema updates in object-oriented database systems. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

A Additional Examples

We mentioned in the introduction that the model is a convenient model for specifying also access rights and updates. We illustrate this with three simple examples.

Access Rights

Users or groups of users are given the right to see only particular contexts. Assume that we have a class *Employee*, a relation *Salary* of type $[E : Employee, S : integer]$, and a relation *Manage* of type $[B : Employee, E : Employee]$. A relation *Emps* contains the set of employees. Each employee o is associated to a specific context, say o, that specifies his/her specific access rights. Besides that we have a general context *base* and a context *finance* for the financial services.

The rule is that each employee can see the *Manage* relation and that an employee is only allowed to see the salaries of the people that he/she manages. All data is virtual except for data in the base context that is explicit. The following program is used to specify accesses:

$$
\begin{array}{lll}
Manage(z) & \sqsupseteq & \{x, y \mid Emps(z) \wedge Manage(base)(x, y)\} \\
Salary(finance) & \sqsupseteq & \{x, s \mid Salary(base)(x, s)\} \\
Salary(z) & \sqsupseteq & \{z, s \mid Salary(base)(z, s)\} \\
Salary(z) & \sqsupseteq & \{z', s \mid Salary(base)(z', s) \wedge Manage(z, z')\}
\end{array}
$$

Observe that the recursion in the definition of *Salary* is only fictitious: $Salary(finance)$ and each $Salary(z)$ depend (with no recursion) on $Salary(base)$ that is stored.

The granting of access rights can be controlled similarly.

Data Updates

Suppose that Jane is a researcher in the professional context (i.e., a name of type Researchers(business). We can add a role to Jane in this context or insert her in another context using:

```
insert Jane    in Friends(personal)
        with   Jane@Friends(personal) . phone = "46262626"
insert Jane    in Friends(business)
        with   Jane@Friends(business) . phone = "46262626"
```

These are two examples of object migration. (See [Su91] for more on object migration.)

Schema Updates

Consider two classes C_1, C_2 with identical type to simplify. Suppose that we want to make a new version that merges the two classes into a single class C. Suppose that we have relations R_1 and R_2 that contain respectively C_1 and C_2 objects and R is the new relation that will contain the union of R_1 and R_2. Let *old* and *new* be the names of the versions before and after update. The type definitions and the program are given by:

relation $\mathrm{R_1(old) : C_1}$
relation $\mathrm{R_2(old) : C_2}$
relation R(new) : C
class $\mathrm{C_1(old), C_2(old), C(new)}$: T
insert X@C(new) in R(new)
from X in $\mathrm{R_1(old)}$;
insert X@C(new) in R(new)
from X in $\mathrm{R_2(old)}$;