The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System

Christine Delcourt(*), Roberto Zicari (**)
(*) Altaïr, France
(**) Politecnico di Milano, Italy
e-mail: relett15@imipoli.bitnet

Abstract

Schema evolution is an important facility in object-oriented databases. However, updates should not result in inconsistencies either in the schema or in the database. We show a tool called ICC, which ensures the structural consistency when updating an object-oriented database system.

1 Introduction

Schema evolution is a concern in object-oriented systems because the dynamic nature of typical OODB applications calls for frequent changes in the schema . However, updates should not result in inconsistencies either in the schema or in the database.

We present a tool which ensures the structural consistency of an object-oriented database system while performing schema updates. The tool has been implemented to evaluate the correctness of schema updates for the O_2 object-oriented database system [Ban91][LecRic89a].

1.1 Preliminary O_2 concepts

In this Section we briefly recall the fundamental concepts of O_2 which are relevant for our discussion. The reader is referred to [LecRi89a],and [LecRi89b] for a formal definition of the O_2 data model and to [Vel89] for the description of the system architecture. O_2 is an object-oriented database system and programming environment developed at Altaïr. Classically, in object-oriented data models, every piece of information is an object. In the O_2 data model, both *objects* and *values* are allowed. This means that, in the definition of an object, the component values of this object do not necessarily contain only objects, but also values. In O_2 we have two distinct notions: *classes* whose instances are objects and which encapsulate data and behavior, and *types* whose instances are values. To every class is associated a type, describing the structure of its instances. Classes are created using schema definition commands. Types are constructed recursively using *atomic types* (such as integer, string, etc.), *class names*, and the *set*, *list*, and *tuple* constructors. Therefore types can be complex. Objects have a unique internal identifier and a value which is an instance of the type associated with the class. Objects are encapsulated, their values are not directly accessible and they are manipulated by *methods*. Method definition is done in two steps: First the user declares the method by giving its *signature*, that is, its name , the type of its arguments and the type of the result (if any). Then the code of the method is given. In O_2, the schema is a set of classes related by inheritance links and/or composition links. The inheritance mechanism of O_2 is based on the subtyping relationship, which is defined by a set inclusion semantics.

Multiple inheritance is supported. O_2 offers a compile-time type-checker in an attempt to statically detect as many illegal manipulations as possible of objects and values. Objects are created using the "new" command. If a class is created "with extension" then a named set value is created which will contain every object of the class and will persist. O_2 allows object values to be manipulated by methods other than those associated with the corresponding class. This feature is obtained by making "public" the type associated with the class.

Methods in O_2 can call other methods of the same class, or "public" methods defined in other classes. They may access directly a type associated to a class (besides the class to which they are associated) if this type has been defined "public". The inheritance scope of a method can be changed by application of the "@" feature which allows a reference to a method from outside the scope of the method.

Example: Given two classes, C,C2 with C2 subclass of C, it is possible, in the body of method m2 defined in C2, to refer to a method m defined in C instead of method m redefined in class C2, as the scope rule would normally imply (see Figure 1).

```
C  m:(C->C')
|
C2 m:  (C2->C")
   m2: (C2->C")
       body.m2 : [...m@C...]
```

Figure 1

When a class inherits methods or types from more than one class (multiple inheritance) conflicts with names for methods and attributes have to be explicitly solved by the designer. For example, two methods with the same name defined in different superclasses will not be inherited by the common subclass. The designer has two possible choices to solve the name conflict:

- either redefine the method in the subclass or
- specify which method he/she wants to inherit using a "*from class* " clause which specifies the chosen inheritance path.

1.2 Schema Updates: What is the problem?

Informally, the problem with updates can be stated as follows: We want to change the structural and behavioral part of a set of classes (schema updates) and/or of a set of named objects (object updates) without resulting in run-time errors, "anomalous" behavior and any other kinds of uncontrollable situation. In particular, we want to assure that the semantics of updates are such that when a schema (or a named object) is modified, it is still a consistent schema (object). Consistency can be classified as follows [Zic90a]:

a. *Structural consistency.* This refers to the static part of the database. Informally a schema is structurally consistent if the class structure is a direct acyclic graph (DAG), and if attribute and method name definitions, attribute and method scope rules, attribute types and method signatures are all compatible. An object is structurally consistent if its value is consistent with the type of the class it belongs to.

b. *Behavioral consistency.* This refers to the dynamic part of the database. Informally an object-oriented database is behaviorally consistent if each method respects its signature and its code does not result in run-time errors or unexpected results.

In this paper, we will only consider the issue of preserving structural consistency.

We will consider "acceptable" only those updates that do not introduce structural inconsistency, while we will allow behavioral inconsistencies that do not result in run-time errors. Any kind of behavioral inconsistency that has been caused by an update will be reported to the user (designer). We have implemented a tool, the ICC which guarantees such consistency.

1.3 Paper Organization

The paper is organized as follows: Section 2 defines more formally the notion of structural consistency for the O_2 object-oriented database system. Section 3 presents the list of updates we allow on the schema, and give a few definitions which will be used in the rest of the paper. Section 4 presents by means of a selected example, the algorithms performed by the ICC to ensure structural consistency. Section 5 gives some concluding remarks.

2 Ensuring Structural Consistency

In this Section, we discuss one basic type of consistency relevant to the O_2 system (but in general to every object-oriented database system) , namely *structural* consistency.

Structural consistency refers to the static characteristics of the database.

We recall here some of the basic definition of O_2 as defined in [LecRic89a] which will help us to define the notion of a consistent schema.

We denote $T(C)$ the set of all types defined over a class C. T(C) includes atomic types, class names, tuple, set and list types.

Inheritance between classes defines a class hierarchy: A class hierarchy is composed of class names with types associated to them, and a subclass relationship. The subclass relationship describes the inheritance properties between classes.

Definition 2.1 A *class hierarchy* is a triple (C, σ , $\prec$) where C is a finite set of class names, σ is a mapping from C to T(C), i.e. σ (C) is the structure of the class of name C, and $\prec$ is a strict partial ordering among C.

The semantics of inheritance is based on the notion of subtyping. The subtyping relationship $\leq$ is derived from the subclass relationship as follows:

Definition 2.2 Let (C, σ, $\prec$) be a class hierarchy, the subtyping relationship $\leq$ on $T(C)$ is the smallest partial ordering which satisfies the following axioms:

1. $\vdash$ c $\leq$ c', for all c, c' in C such that c $\prec$ c'. That is, a subclass is a subtype.
2. $\vdash$: [a_1: t_1,...,a_n: t_n,...,a_{n+p} : t_{n+p}] $\leq$ [a_1: s_1,...,a_n: s_n], for all types t_i and s_i, i=1,...,n such that $t_i \leq s_i$. This is subtyping between tuple types. We can refine tuples by refining some attributes or by adding new ones.
3. $\vdash \{s\} \leq \{t\}$, for all types s and t such that $s \leq t$. This is subtyping between set types.
4. $\vdash < s > \leq < t >$, for all types s and t such that $s \leq t$. This is subtyping between list types.
5. t $\leq$ *any*, for all types t. The symbol *any* is a type by definition.

As inheritance is user given, some class hierarchies can be meaningless.

In a class hierarchy an instance of a class is also an instance of its superclasses (if any). Therefore, if class c' is a superclass of class c, then we must have that the type of c is a subtype of the type of c'. More formally:

Definition 2.3 A class hierarchy (C, σ, $\prec$) is *consistent* iff for all classes c and c', if c $\prec$ c' then σ (c) $\leq \sigma$ (c').

Example: This is a consistent class hierarchy. Class Employee is a subclass of Person, (i.e. Employee $\prec$ Person):

```
class Person
type tuple [name:string,
            age: integer,
            address: tuple [location: City,
                            street: string] ]

class Employee
type tuple [name:string,
            age: integer,
            address: tuple [location:City,
                            street: string],
            profession: string,
            company: string ]
```

A schema is also constituted of methods attached to classes. Methods have signatures.

Definition 2.4 A method *signature* in class C is an expression m: c x t_1 x ...x $t_n \rightarrow$ t, where m is the name of the method, and c, t_1 ... t_n are types. The first type c must be a class name and is called the receiver class of the method.

We are ready to define a schema.

Definition 2.5 An O_2 *database schema* is a 5-tuple S=(C, σ, $\prec$,M,N), where:

- (C, σ, $\prec$) is a consistent class hierarchy (see def.3)
- M is a set of method signatures in C
- N is a set of names with a type associated to each name

A schema is therefore composed of classes related by inheritance which follow the type compatibility rules of subtyping and a set of methods. Attributes and methods are identified by name. Within the schema, type attributes and method names have a *scope rule* (see def. 7). When we do not want to distinguish between a type attribute and a method name, we simply use the term *property*.

Now we are ready to define what we mean with structural consistency for a database schema.

Definition 2.6 A database schema S is *structurally consistent* iff it satisfies the following properties:

- if c $\prec$ c' and the method m is defined in c with signature m:c x t_2...$t_n \rightarrow$ t, and method m' is defined in c' , and m and m' have the same name, with signature m': c' x t'_2...$t'_n \rightarrow$ t' , then $t_i \leq t'_i$ and t $\leq$ t' (covariant condition)
- the class hierarchy is a DAG
- if there are classes c1 and c2 having a common subclass c4, with a property name p defined in both c1 and c2, but not in C4, then there is another subclass c3 of c1 and c2 in which the property p is also defined and c4 is a subclass of c3.

The first property assures that method *overloading* is done with *compatible* signatures, the second property constrains the structure of the class hierarchy, and finally the last property eliminates multiple

inheritance conflicts (also denoted as *name conflicts*). Definition 2.6 is important, because we will always consider schemes which are structurally consistent. An update to a schema is a mapping which transforms a schema S into a (possibly) different schema S'. Schemes S and S' have to be structurally consistent. The semantics of the schema update primitives will have to ensure *at least* that structurally consistent schemes are produced as a result of an update. In our approach name or type conflicts occurring as a consequence of an update will not be solved automatically by the system.

We also give an auxiliary definition which will be used in the rest of the paper.

Definition 2.7 Given a property name p and a class C, the *scope* for p in C, denoted *scope(p,C)* is the set of C and all subclasses of C (recursively obtained) where p is not locally redefined plus all classes where p is referred to with the "*p from C*" clause. (The algorithm to construct a scope is given in Def.3.5)

An existing method or attribute in a class C can be

- locally defined in C or,
- inherited from a superclass or,
- specified with the "from C_p" clause, C_p being the class where p is locally defined.

When we do not want to distinguish between the above cases, we say that a property p "exists" in class C.

Figure 2 illustrates the three different cases of inheritance.

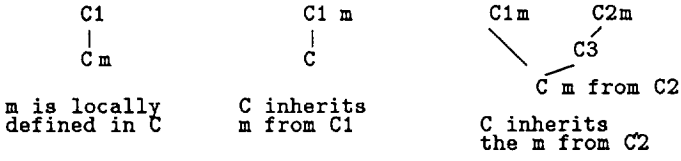


Figure 2

2.1 The ICC: A Basic Schema Update Tool

The way the designer updates the schema is a dialogue with an interactive tool called the Interactive Consistency Checker (ICC). The ICC is a basic update tool which, given a schema and a proposed update, detects whether structural inconsistencies may occur. It then refuses those updates which produce structural inconsistencies: the update is not performed. The reason for the refusal of the update is always given to the user.

3 Schema Updates

We present in this Section the complete list of basic updates one can perform on an O_2 schema.

Updates are classified in three categories: Updates to the type structure of a class, to methods of a class, and to the class as a whole. This classification is fairly similar to the one of [Ba87a,Ba87b]. However, the semantics of some updates is different. Updates have parameters and their semantics can be given in accordance to application's requirements. For details the reader is referred to [Zic90a] [Zic90b] where the syntax and semantics of the operators are defined.

SCHEMA UPDATES:

1. *Changes to the type structure of a class*

   Because in O_2 types can be arbitrarily complex, we have different ways to modify a class type. We can think of an update u which modifies the type structure T of a class C, as a mapping between types, $u : T \rightarrow T'$. Updates of this kind can be broadly classified in two categories: those for which $T' \leq T$ (we call them type-preserving), and those for which $T' \not\leq T$ (we call them non type-preserving). Of all possible type updates we list here only the most elementary ones:

   1.1 Add an attribute to a class type

   1.2 Drop an existing attribute from a class type

   1.3 Change the name of an attribute of a class type

   1.4 Change the type of an attribute of a class type

   Updates 1.1 and 1.3 are type-preserving. Update 1.2 is non type-preserving, while update 1.4 is type-preserving if new- type $\leq$ old-type.

2. *Changes to the methods of a class*

   2.1 Add a new method

   2.2 Drop an existing method

   2.3 Change the name of a method

   2.4 Change the signature of a method (this update may be also implied by a change to the class structure graph as defined below)

   2.5 Change the code of a method.

3. *Changes to the class structure graph*

   3.1 Add a new class

   3.2 Drop an existing class

   3.3 Change the name of a class

   3.4 Make a class S a superclass (subclass) of a class C

   3.5 Remove a class S from the superclass (subclass) list of C

The list of updates defined above can be reduced: There exists a basic set of updates which can be used to execute all other updates.

The basic set of updates is the following:

BASIC SCHEMA UPDATES:

1.1 Add an attribute to a class type
1.2 Drop an attribute from a class type
2.1 Add a method
2.2 Drop a method
3.1 Add a class
3.2 Drop a class
3.3 Change the name of a class
3.4 Make a class a superclass (subclass) of C
3.5 Remove a class from the superclass (subclass) list of C

The other updates in the previous list can be executed using sequences of basic updates.

```
(e.g.  1.3 = <1.2 , 1.1>, this equivalence does not hold at
                          instance level
       1.4 = <1.2 , 1.1>, this equivalence does not hold at
                          instance level
       2.3 = <2.2 , 2.1>
       2.4 = <2.2 , 2.1>
       2.5 = <2.2 , 2.1)
       3.3 = <3.2,  3.1>, this equivalence does not hold at
                          instance level)
```

The sequence of basic updates corresponding to a non elementary update has to be atomic, to avoid inconsistency.

3.1 Additional Definitions

We give a few more definitions which we will use through out the paper.

3.1.1 DAG

A DAG is the formal representation of the class hierarchy. It is defined as follows:

Definition 3.1 : DAG
A direct acyclic graph (DAG) is defined as a pair $(E_c, \prec)$ where

- E_c is the set of *nodes*. Each node represents a class.
- $\prec$ is a partial order with class Object as root.
  $\forall$ C, C' $\in E_c \times E_c$, C' $\prec$ C $\Leftrightarrow$ C is a direct superclass of C'
  $\Leftrightarrow$ C' is a direct subclass of C.
  C' $\prec$ C represents an *edge* between C and C', C being higher than C' in the hierarchy. □

Definition 3.2 $\forall$ C, C' $\in E_c \times E_c$, we can define a:

- **Path between two classes**
  C' $<$ C $\Leftrightarrow$ (C' $\prec$ C) $\vee$ ($\exists$ C_1, ..., $C_n \in E_c$ / (C' $\prec$ C_n), ($C_n \prec C_{n-1}$),..., ($C_2 \prec C_1$), ($C_1 \prec$ C)).
  C' $<$ C indicates that a path exists from C' to C going up: C and C' are related, C being higher than C' in the DAG. In this case, C is called a superclass of C' and C' a subclass of C. □
- **Set of all direct subclasses of a class**
  direct_subclasses(C) = $\{C_i$ / $C_i \in E_c \wedge (C_i \prec C)\}$.
  direct_subclasses(C) is the set of all direct subclasses of the class C. □
- **Set of all direct superclasses of a class**
  direct_superclasses(C) = $\{$ C_i / $C_i \in E_c \wedge (C \prec C_i)\}$.
  direct_superclasses(C) is the set of all direct superclasses of the class C. □
- **Set of all subclasses of a class**
  subclasses(C) = *direct_subclasses*(C) $\cup$ ($\cup_{(C_i \in direct\_subclasses(C))}$ *subclasses*(C_i)).
  subclasses(C) is the set of all subclasses of the class C. □
- **Set of all superclasses of a class**
  superclasses(C) = *direct_superclasses*(C) $\cup$ ($\cup_{(C_i \in direct\_superclasses(C))}$ *superclasses*(C_i)).
  superclasses(C) is the set of all superclasses of the class C. □

Example:
Given four classes C_0, C_1, C_2, C_3, with C_2, C_3 subclasses of C_1, C_1 subclass of C_0 as shown in figure 3, we have $C_2 < C_0$, $C_3 < C_0$, *direct_subclasses*(C_0)={C_1}, *direct_superclasses*(C_2)={C_1}, *subclasses*(C_0)={C_1, C_2, C_3} and *superclasses*(C_2)={C_1, C_0, Object}.

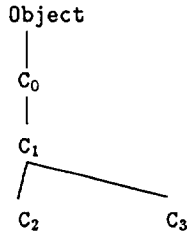


Figure 3

3.1.2 Virtual edge

We introduce the notion of *virtual edge* to represent a "from" clause in the DAG. All definitions here will be helpful in the detection of name conflicts and type incompatibility in the schema.

Definition 3.3 : A virtual edge represents a "from" clause
Given a property p and two classes C' and C*, if p in C' is defined using the "from C*" clause this requires that p is locally defined in C*. This reference is represented by a *virtual edge* in the DAG. This virtual edge is added between C* and C', it is labeled by the property with name p and it is denoted by: *virtual_edge(C*, C', p)*. □

Example: Consider three classes C, C', C* with C and C* superclasses of C and a property m in C1 defined using the from clause (see figure 4).
This corresponds to the virtual edge *virtual_edge(C*, C', p)* which is equivalent to p in C' from C*.

Figure 4

Note: A *virtual edge* is "stronger" than ordinary DAG edges. Given a *virtual_edge(C*, C', p)* to determine the inheritance of p in C' , only this virtual edge is considered and not the ordinary edges. In our example, C' inherits p from class C* and not p from class C or another class.

In fact, a "from" clause it is used to "force" the inheritance of properties to avoid name conflicts.

Note: In the actual implementation of O_2, the notion of virtual edge is constrained as follows:
A virtual edge *virtual_edge(C*, C', p)* exists if and only if there exists a path C'<C*.
Our definition of virtual edge is more general than this and allows a reference to a class C* even if it is not connected to C'.

Definition 3.4 : Set_virtual_edge
Consider a property p existing in a class C, we have :
set_virtual_edge(p, C) =
{V / virtual_edge(C, V, p) exists} in case p of V is locally defined in C.
$\emptyset$ otherwise.
The set *set_virtual_edge(p, C)* contains all classes of C which have a virtual edge going from the class C where p is locally defined. □

3.1.3 Scope

Inheritance of properties is based upon the *scope* and *use* concepts which are defined in this section. These arising concepts will be helpful in the detection of possible name conflicts arising after a schema update.

Each property existing in a class has an associated scope. A scope is defined as follows.

Definition 3.5 : Scope of a property in a class
The *scope(p, C)* of a property p in a class C is the set of classes (including C) which inherit this property by inheritance or by a virtual edge. □

We now show the algorithm to define the scope of a property p for a class C.
Scope constructive algorithm

Given a class C in which the property p exists, the algorithm is used to build the set *scope(p, C)*.
Begin
temp = $\emptyset$.
For each C' $\in$ *direct_subclasses(C)*
if p is inherited then *temp* = *temp* $\cup$ *scope(p, C')* endif
endfor
For each C' $\in$ *set_virtual_edge(p, C)*
temp = *temp* $\cup$ *scope(p, C')*
endfor
scope(p, C) = {C} $\cup$ *temp*.
End

Example:
Consider the classes C_0, C_1, C_2, C_3, C_4 of figure 5 and the property p locally defined in C_0, C_1 and C_5. In C_2, p is defined with the "from C_0" clause.
We have *scope(p, C_0)*={C_0, C_2, C_3, C_4}, *scope(p, C_5)*={C_5}, *scope(p, C_1)*={C_1}, *scope(p, C_2)*={C_2, C_3}, *scope(p, C_4)*={C_4}.

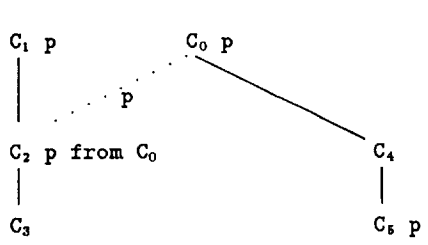


Figure 5

Definition 3.6 : A name conflict in a class
A name conflict occurs in a class C for a property p
⇔ there exist two classes C_1 and C_2 where p exists and is not locally defined in a common ancestor class and $C \in scope(p, C_1) \cap scope(p, C_2)$

Note: If a class inherits at least twice a property p locally defined in the same ancestor class then this property will exit in C only once.

Example: Consider the classes C_0, C_1, C_2, C of figure 6 and the property p locally defined in C_0. C inherits the property p locally defined in C_0 twice (by path C_0, C_1, C and by the path C_0, C_2, C) but p is considered in C only once. No name conflict occurs since those two p inherited are equal; they have the same local definition.

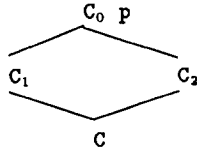


Figure 6

Definition 3.7 : A name conflict in a schema.
A *name conflict* in a schema occurs for a property p
⇔ there exist at least one class C in the schema where a name conflict occurs for p. □

To express how a property p is defined in a class, we introduce the notion of *use*.

Definition 3.8 : Use of a property in a class.
The *use* of a property p in a class C, denoted *use(p, C)* can be:

1. *well_defined*: if p exists in C and there is no name conflict in the class for p or,
2. *undefined*: if p does not exist in C or,

3. *not_well_defined*: a name conflict exists for p in C. □

We now give the algorithm to detect whether the use of a property is well defined, undefined or not well defined.
Given a class C and a property p, the algorithm returns: well_defined, undefined or not_well_defined. These correspond to the three values of *use* (see definition 3.8).

Algorithm

Begin

Case 1: p is locally defined or is derived with the "p from" clause in C.
if there exist at least
- two local definitions for p in C, or
- two "from" clauses for p in C, or
- a local definition and a "from" clause for p in C

then *use(p, C)* is not well defined, a name conflict appears. Return not_well_defined.
else *use(p, C)* is well defined. Return well_defined.
endif

Case 2: otherwise
Look at all superclasses of C (recursively) until p is locally defined in each path leading from C to Object. Note: going up from a class to its direct superclasses takes only the virtual edge for p if one exists.
Case 2-1: If p is not encountered in at least one path then Return undefined.
Case 2-2: If a unique local definition for p is encountered in those paths
then Return well_defined.
Case 2-3: If at least two local definitions for p are encountered in two different classes for two different paths (and the definitions of p are not the same)
then a name conflict appears, return not_well_defined.

End

Example:

Consider figure Figure 7. We have C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8 classes. A property p is locally defined in C_2, C_3 and in C_6 and p is defined using the "from C_3" clause in C_5.
We have *use(p, C_4)*=undefined, p is undefined in C_4,
use(p, C_7)=well_defined, p is defined in C_7 and no conflict appears in C_7, and
use(p, C_8)=not_well_defined, p is not well defined in C_8 because p of C_8 is defined in C_6 and in C_3. A name conflict exists in C_8.

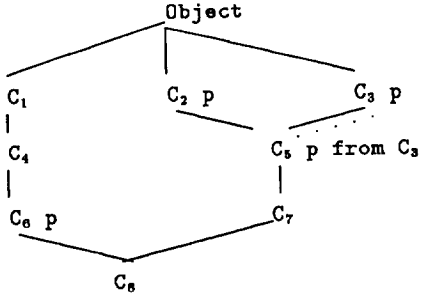


Figure 7

Definition 3.9 : Scope without conflict
In our system name conflicts are forbidden and all scopes need to be without name conflict. Given a class C and a property p, *scope(p, C)* is *"without conflict"* if and only if $\forall$ C' $\in$ *scope(p, C)*, *use(p, C')*$\neq$"not_well_defined".

Property 1 :
Given a class C and a property p locally defined in C, if *scope(p, C)* is *"without conflict"* then $\forall$C'$\in$*scope(p, C)*, *scope(p, C')* is *"without conflict"*

Property 2 :
Given two classes C and C' (C$\neq$C') and a property p existing in those two classes,
if C' $\in$ *scope(p, C)* then *scope(p, C')* $\subset$ *scope(p, C)*.

Property 3 : Name conflict detection
Given a property of name p, by property 1 and 2 we have:
No name conflict occurs in the schema for p $\Leftrightarrow$ $\forall$ C' $\in$ E_c such that p is locally defined in C', *scope(p, C')* is *"without conflict"*.

3.1.4 Scope frontiers and type incompatibility

In this section, we introduce the notion of *frontiers* of a scope. We will see that this notion is of interest for the verification of type compatibility after a schema update is performed.
Let us first define the root of a scope.

Definition 3.10 : Root of a scope
Given a property p existing in a class C, *scope(p, C)* has an associated *root*. This root is denoted *root(p, C)* and it corresponds to the class where p is locally defined. □

Given a property p existing in a class C, *scope(p, C)* has two kinds of *frontiers*, *top* and *lower adjacent limits*.

Definition 3.11 : Top of a scope
Given a property p existing in a class C,
the *top* of *scope (p, C)* corresponds to the set of classes belonging to this scope which contains, the class for which p is locally defined and the classes for which p is defined with the from clause. This set is denoted *top(p, C)* and is defined by:
top(p, C) = {C} in case *virtual_edge*(S, C, p) exists
= $\emptyset$ in case *root(p, C)*$\neq$C and $\nexists$*virtual_edge*(S, C, p)
in fact, p is inherited in C.
= *set_virtual_edge(p, C)* $\cup$ {C} otherwise □

We define here the *leaves* which allow the definition of the *lower adjacent limits* (see definition 3.13) for a given scope.

Definition 3.12 : Leaves of a scope
Given a property p existing in a class C,
the *leaves* of *scope (p, C)* correspond to the bottom classes of this scope. The set of those leaves is denoted *leaves(p, C)* and is defined by:
leaves(p, C) = {C_i / $C_i \in$ *scope(p, C)*, ($\nexists$ $C_j \in$ *scope(p, C)* / $C_j \in$ *subclasses*(C_i))}. □

Definition 3.13 : Lower adjacent limits (lal) of a scope
Given a property p existing in a class C,
the *lower adjacent limits* (lal) of *scope (p, C)* correspond to the direct subclasses of its leaves. This set is denoted *lal(p, C)* and is defined by: $lal(p, C) = \cup_{C' \in leaves(p,C)} \{C_i \ / \ C_i \in direct\_subclasses(C')\}$ □

Example:
Consider figure 8. We have C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8 C_9 classes. A property p is locally defined in C_2, C_5 and in C_9 and p is defined using the "from C_2" clause in C_6 and C_7.
We have $top(p, C_2)=\{C_2, C_6, C_7\}$
$leaves(p, C_2)=\{C_3, C_6, C_8\}$
$lal(p, C_2)=\{C_5, C_9\}$

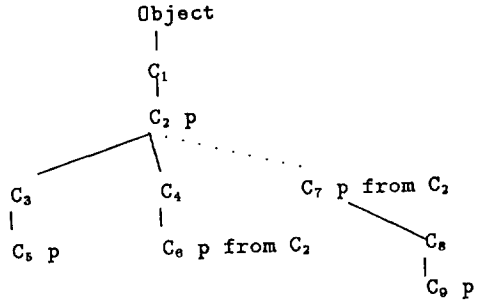


Figure 8

To have type compatibility between classes in the DAG, all properties in the DAG need to be type compatible as defined below. Notation: *typing*(p, C) is the signature or the type of p, p being a method or an attribute in a class C.

Definition 3.14 : Type compatibility for a property p in a class C:
A property p existing in a class C is *type compatible* $\Longleftrightarrow$
$(\forall C_i \in$ direct_superclasses(C) such that p exists in C_i, *typing*(p, C) $\leq$ *typing*(p, C_i))
$\wedge$ ($\forall C_i \in$ direct_subclasses(C), *typing*(p, C_i) $\leq$ *typing*(p, C)). □

As we will see, because a type modification will affect the scope of the entire DAG and not only of a single class, it is interesting to define a type compatibility notion for a scope. Let us first define a well typed scope.

Definition 3.15 : Well typed scope
Given a property p existing in a class C,
scope(p, C) is *well typed* if and only if $\forall$ C' $\in$ *scope(p, C)*, p in C' is *type compatible* (see definition 3.14).
□
We introduce the predicate *scope_compatibility(p, C)*=True to express that the *scope(p, C)* is well typed.
scope_compatibility(p, C) = true $\Longleftrightarrow$
(R1): ($\forall C_j \in top(p, C), \forall C_i \in$ direct_superclasses(C_j) with p existing in C_i, *typing*(p, C)$\leq$*typing*(p, C_i))
$\wedge$
(R2): ($\forall C_i \in lal(p, C)$, *typing*(p, C_i) $\leq$ *typing*(p, C)).

Thus, the type compatibility problem for a property p is equivalent to the well typed scope problem.

Property 4 : Type incompatibility detection for a property
No type incompatibility occurs in a schema for a given property p $\Longleftrightarrow \forall C \in E_C$ such that p is locally defined in C, *scope(p, C)* is well typed.

This shows that using the definition of a well typed scope, the number of comparisons between signatures or types when a modification is done, can be limited to the frontiers of the modified scopes. In fact, for a given scope *scope(p, C)*, all classes have the same p, thus the same type; therefore only the frontiers are of interest.
Moreover, we have:

Property 5 *1. If the signature (type) of a property p is replaced by a superior signature (supertype) then the scope affected is well typed if (R1) is satisfied.*

2. If the signature (type) of a property p is replaced by an inferior signature (subtype) then the scope affected is well typed if (R2) is satisfied.

3.2 Basic and Parametrized Updates

The updates presented at the beginning of Section 3 are parametrized updates (see [Zic90a] for details). We can consider them still high level updates. In fact, among these updates, there exists a basic set of updates such that their corresponding structural check can be used to do the structural check of the other ones.
Therefore, we decided to define lower level schema updates. This level is composed of the following basic updates: (Note: The schema structural check is the same for the attributes and methods updates: we will in the following speak of property updates.) Thus we have defined a set of lower level updates

Changes to the properties of a class

- add a property local definition
- add a "from" clause for an existing property
- replace a local definition into a new local definition or a from clause for a property
- drop a property local definition
- drop a "from" clause for a property
- replace a from clause into a new local definition or a from clause for a property

Changes to the type structure of a graph

- add a node
- add an edge
- change the name of a node
- drop a node
- drop an edge

which can be used to implement all possible semantics of the updates (the ones of the high level) as indicated by the user.

In [DeZi91] it is shown the correspondence between user-parametrized updates and the basic updates.

When the user submits to the ICC a high level update u, this update is translated into a sequence of basic updates. For each of the basic updates the ICC performs a consistency check. If all basic updates composing the high level update are validated, then the high_level update is also validated. If one of the basic updates composing the high_level update induces structural inconsistency in the schema, the high_level update is refused and a warning is given to the user.

The ICC checks if a schema S after an update u is structurally consistent.
By definition, a schema S is structurally consistent iff the following invariants are satisfied:

Definition 3.16 : Invariants for a structural consistency

- Class lattice invariant:
  - All classes must be connected in the DAG
  - The root of the DAG is the class Object
  - The name of a class must be unique in the DAG
- Name conflict invariant: *use(p, C)* must be well_defined for all properties p existing in a class C.
- Type compatibility invariant: All classes in the schema have to be *compatible.*

□

The ICC is a *basic tool*; it only detects inconsistencies introduced by an update. It does not solve them automatically.
Given a schema and an update as an input, the ICC detects whether structural inconsistencies occurred or not. If a structural inconsistency arises the tool refused the update and provides all the detected inconsistencies to the user. If no structural problem occurs the update is done.

A study of a more sophisticated tool is considered in [De90a].

4 Schema structural consistency check

This section describes by means of a selected example some of the algorithms used by the ICC to verify schema structural consistency.

Notation:In the example, for each updates we consider an initial state and a final state. To represent those, each concept will be marked with the *after* or *before* marks. For example, given an update. $scope(..) = scope_{before}(..)$ before the update, and $scope(..) = scope_{after}(..)$ after the update.

While performing an update, the ICC check its effects on the schema to ensure structural consistency.

The structure of the checks is similar and include the following steps:

Algorithm:

1. C_check: A set of constraints are checked.
2. The DAG structure is checked.
3. NC_check: Name conflicts are detected.
4. TI_check: Type incompatibilities are detected.
5. CD_check: The dependency problem is studied.

If one problem occurs the update is refused otherwise it is performed.
end

4.1 An Example

We present in this section the algorithms to detect structural inconsistencies when performing a specific update: Adding a property in a class. The description of the algorithms for the other update primitives defined in Sect. 3.2 is reported in [DeZi91].

4.1.1 Property addition

Let us consider the addition of a property p in a class C. This addition consists of adding a local definition of p in C using the *add_local_property(p, C, ≺signature, type≻)* update.

Add_local_property: *add_local_property(p, C, ≺signature, type≻)*

- Semantics:

  1. The DAG structure is not affected.
  2. We consider two cases :
     - *Case 1*: $use_{before}(p, C)$="undefined" (p did not exist in C)
     - *Case 2*: $use_{before}(p, C)$="well_defined" (p existed in C).

  If neither a name conflict nor a type incompatibility occurs then
  - *case 1*: *scope(p, C)* is created.
  - *case 2*: $scope_{after}(p, R) = scope_{before}(p, R) - scope(p, C)$ with $R=root_{before}(p, C)$.

  A definition for p is locally added in class C. This property p is then propagated to the subclasses of C until a redefinition of p occurs.
  else the update is refused.

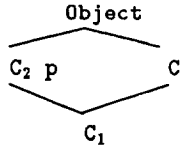
  We need to check name conflicts and type compatibilities. We first define the algorithm for name conflict detection then the one for type incompatibility detection.

- Name conflict detection:
  A conflict can occur only in classes of $scope_{after}(p, C)$.
  - *case 1*: $use_{before}(p, C)$="undefined"
    $use_{after}(p, C')$="not_well_defined" for $C' \in scope_{after}(p, C)$ $(C' \neq C)$
    $\Leftrightarrow$ p was inherited in C' before the update.
  - *case 2*: $use_{before}(p, C)$="well_defined"
    * $use_{after}(p, C)$="not_well_defined" $\Leftrightarrow$ $C \in top_{before}(p, R)$ with $R=root_{before}(p, C)$
    * $use_{after}(p, C')$="not_well_defined" for $C' \in scope_{after}(p, C)$ $(C' \neq C)$
      $\Leftrightarrow$ ($\exists X \in$ direct_superclass(C')/
      $(X \notin scope_{after}(p, C) \wedge$ p exits in X before the update)
      or $(X \in scope_{after}(p, C) \wedge use_{after}(p, X)$="not_well_defined")).

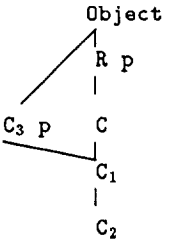
  Example: Let us look at figure 9 which gives two examples of name conflict after a property addition.

Algorithm: NC_add_prop(*DAG, p, C*)
Goal: This algorithm searches for the classes where a name conflict for *p* occurs after the addition of p in class C.
Input: the schema (*DAG*), the property *p* and the class *C* where *p* has to be added.
Output: the set of classes where a name conflict occurs for *p*.
Procedure:

- if $use_{before}(p, C)$="undefined"



Case 1:
We add a property p in class C
A conflict occurs in C_1



Case 2:
We add a property p in class C
A conflict occurs in C_1 and C_2

then for each class C' ∈ $scope_{after}(p, C)$ (C'≠C) do
if $use_{after}(p, C')$="not_well_defined" (see name conflict detection: *case 1*)
then **return CONFLICT in C'** endif
endfor
endif
- if $use_{before}(p, C)$="well defined"
then if $use_{after}(p, C)$="not_well_defined" (see name conflict detection: *case 2*)
then **return CONFLICT in C** and **exit**
else for each class C' ∈ $scope_{after}(p, C)$ (C'≠C) do
if $use_{after}(p, C')$="not_well_defined" (see name conflict detection: *case 2*)
then **return CONFLICT in C'** endif
endfor
endif
endif

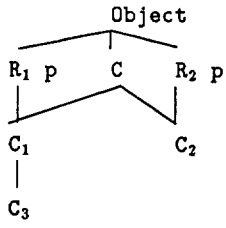
Endprocedure.

Remark: To improve this algorithm, only the first conflict encountered for each path of $scope_{after}$(p, C) going down by width has to be given: the others are implied by the upper one. Thus, to be more efficient the best way is to order the classes of $scope_{after}$(p, C). The order should be the order of its building when the hierarchy is traversed by width from C going downward.

Example: Let us look at figure 10. We have classes C, R_1 R_2, C_1, C_2, C_3 and a property p is locally defined in R_1 and in R_2. The addition of a property p in class C results in a conflict in C_1, C_2 and C_3. The reason for the conflict in C_1 and C_3 is the same: it is the existence of p in R_1. Thus, the detection algorithm would have to warn of the conflict in C_1 and C_2 and that is all.

- Type incompatibility detection:

The classes which may be affected by type incompatibility are those of *scope(p, C)*; thus we have to verify that the scope frontiers are well defined after the update.



Type compatibility will be satisfied after the update if *scope_compatibility(p, C)* = true (see definition 3.15).

By hypothesis, after the update we have: $top_{after}(p, C) = \{C\}$. We have two cases:

- *case 1*: $use_{before}(p, C)$="undefined"
  We just have to check type compatibility for the lower adjacent limits of $scope_{after}(p, C)$. That means to verify rule (R2) for the definition of *scope_compatibility*(p, C). The reason is because $top_{after}(p, C) = \{C\}$ and C has no superclasses where p exists.
- *case 2*: $use_{before}(p, C)$="well_defined"
  Type compatibility for $scope_{after}(p, C)$ has to be verified for all its frontiers (see section 3.14). Rules (R1) and (R2) of *scope_compatibility*(p, C) needs to be verified.

Algorithm: TI_add_prop(*DAG, p, t, C*)
Goal: This algorithm searches for the type incompatibilities which occur in the schema after the addition of a property *p* in a class *C*.
Input: the schema (DAG) which is modified, a property name p with its signature or type *t* (t=$typing_{after}(p, C)$) and the class C where p has to be added.
Output: the classes where a type incompatibility occurs for the property of name p.
Procedure:

if $use_{before}(p, C)$="well_defined" with R = $root_{before}(p, C)$

then we are in *case 2*:

- if t = *typing*(p, R) then **Return No_incompatibility, Exit** endif
- if t$\not\leq$*typing*(p, R) (see definition 2.3)
  then **Return Incompatibility in C, Exit**
  else for each class C' $\in$ $lal_{after}(p, C)$ do
    if *typing*(p, C')$\not\leq$t then **return Incompatibility in C'** endif
  endfor
  endif

else we are in *case 1*

for each class C' $\in$ $lal_{after}(p, C)$ do
  if *typing*(p, C')$\not\leq$t then **return Incompatibility in C'** endif
endfor

endif
Endprocedure.

5 Conclusions and future work

We have specified and implemented a tool (ICC) for schema evolution ensuring schema structural consistency.

This tool provides two levels of updates.

- The lower level is a set of basic updates. This level ensures the completeness of the tool since its provides all possible updates to obtain every consistent schemes.
- The higher level is composed of parametrized updates which are expressed using the lower level updates.
- The tool ensures the structural consistency of a schema while performing an update: the invariant properties for schema structural consistency are checked when an update is performed therefore only valid schemes are produced.

The ICC provides the basic mechanism for schema evolution. It has the advantage being flexible. The higher level updates can be redefined or completed by new parametrized updates. The ICC can be used to built on top a more sophisticated tool: an adviser helping the schema designer and providing the facility to define update transactions. So far, we have implemented two running prototypes of the ICC tool. The ICC prototypes are intended as experiments towards realizing a more powerful schema designer tool. The ICC tool has been developed independently from the O_2 product, and therefore, there is no direct relationships with the schema designer provided by the forthcoming O_2 product.

As future work, we plan to:

1. build a more sophisticate tool on top of the basic one described in this paper.
2. ensure behavioral consistency when updating the schema.A first proposal which uses a data-flow technique is reported in [CLZ91].

Acknowledgments
We thank the anonymous referees for valuable comments.
Luca Breveglieri provided invaluable support in consulting for LaTeX.

References

[CLZ91] Coen A., Lavazza G., Zicari R.,"Updating the Schema of an Object-Oriented Database", IEEE Data Engineering bulletin,July 1991, to appear.

[Ng&Ri88] Gio-toan Nguyen and Dominique Rieux, "Schema Evolution for Object-Oriented Database Systems", INRIA Research report, 1988 December.

ORION

[Ba&Al87a] J. Banerjee et al, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD 1987.

[Ba&Al87b] J. Banerjee et al., "Data Model Issues for Object-Oriented Applications", ACM TOOIS, vol.5,No.1, January 1987.

[Kim&Cho88] W. Kim, and Hong-Tai Chou, "Versions of Schema for Object-Oriented Databases", Proc. 14th VLDB, 1988, Los Angeles.

[Ki&Al88] W. Kim et al., "Integrating an Object-Oriented Programming System with a Database System, ACM OOPSLA , September 1988.

GEMSTONE

[Pen&Stei87] D.J. Penney, J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", ACM OOPSLA October 1987.

ENCORE

[Ska&Zdo(a)] A.H. Skarra,S.B. Zdonik, "The Management of Changing Types in an Object-Oriented Database", ACM OOPSLA, September1986.

[Ska&Zdo(b)] A.H.Skarra,S.B.Zdonik, "Type Evolution in an Object-Oriented Database", in Research Directions in Object Oriented Systems, MIT press.

[Zdo87] S.B. Zdonik, "Can Objects Change Types? Can Type Objects Change? (extended abstract)", Workshop Roscoff September 1987.

O2

[Ban91] F. Bancilhon, C. Delobel, P. Kannelakis, (eds.) "The O_2 book", Morgan Kaufmann publisher 1991 to appear.

[Ben&Al88] V. Benzaken et al.,"Detail Design of the Object Manager", Altair, October 1988.

[De90a] C. Delcourt, "The schema update problem for the O_2 object oriented database system", July 1990.

[De90b] C. Delcourt, "Schema updates: Integrity Consistency Checker for O_2 Object Oriented Database System", July 1990.

[DeZi91] . DelCourt, Zicari R.,"The Design of an Integrity Consistency Checker (ICC) for an Object-Oriented Database System", Politecnico di Milano, Report 91-021, November 1990.

[Gam89] S. Gamerman, "Detailed Specifications of the Type and Method Manager V04", Altair, January 1989.

[LecRic89a] C. Lecluse,P. Richard, "The O2 Database Programming Language", Altair Report 26-89, January 1989. Also in Proc. VLDB , Amsterdam, 1989.

[LecRic89b] C. Lecluse,P. Richard, "Modeling Complex Structures in Object-Oriented Databases", in proc of the PODS 89 Conference, Philadelphia, March 29,31, 1989.

[VelAl89] F. Velez et al., "The O2 Object Manager: an Overview", Altair, February 1989.

[Wal89] E. Waller, PhD Thesis in preparation.

[Zic90a] R. Zicari, A Framework for Schema Updates in an Object-Oriented Database System, in the O2 book, (F. Bancilhon, C. Delobel, P. Kanellakis, eds.), Morgan Kaufmann publisher, 1991 to appear. A short version in Proc. IEEE 7th Data Engineering Conf., April 8-12, Kobe, Japan 1991.

[Zic90b] R. Zicari, Primitives for Schema Updates in an Object-Oriented Database System, in Proc. OODBTG Workshop of the accredited standard committee, X3, SPARC,DBSSG,OODBTG, October 23, Ottawa, 1990.