Management Of Schema Evolution In Databases

José Andany
CUI
Université de Genève
12 rue du Lac
CH-1207 Genève
Switzerland
Tel. (22) 787.65.82

Michel Léonard
CUI
Université de Genève
12 rue du Lac
CH-1207 Genève
Switzerland
Tel. (22) 787.65.82

Carole Palisser
LIST
Université de Nantes
Faculté des Sciences
2 rue de la Houssinière
44072 Nantes cedex
France
Tel. 40.37.49.02

Abstract

This paper presents a version model which handles database schema changes and which takes evolution into account. Its originality is in allowing the development of partial schema versions, or views of a schema. These versions are created in the same database from a common schema. We define the set of authorised modifications on a schema and the rules which guarantee its coherence after transformation. Mechanisms allowing data to be associated with each version are also integrated in the model.

1. Introduction.

In the database life cycle, the schema evolution problem first comes up during the design phase. However, its evolutive aspect is not specific to this phase. During the post-design phase, a schema can be modified, for example after a significant evolution of the application domain, or again in refining the application description. Finally, this kind of transformation is sometimes necessary for performance reasons. Schema evolution handling during the operational phase of a database is a complex problem. During this stage, each schema change needs to take into account previously stored data. In particular, such transformations usually require storing previous schema in order to retain accessibility to the associated data. This leads at the same time to the problem of managing different schema versions and that of the correspondences between these versions and the data.

Software producers were undoubtedly the first to meet the need of taking data evolution into account. Numerous version managers were implemented in the software engineering field, in order to manage the different states generated during the design and maintenance of a program [Rochkind,75], [Tichy,85], [Kaiser,83], [Estublier,84]. The last few years have seen the version control problem in new application fields of DataBase Management Systems (DBMS) such as Computer Aided Design (CAD). It is an important direction for research and development in the field [Katz,84], [Katz,86], [Katz,87], [Kim,85], [Batory,85], [Chou,86], [Klahold,86], [Autran,87], [Fauvet,88], [Palisser,89], [Palisser,90a]. [Palisser,89] contains a synthesis of projects based on the version problem in software engineering and in CAD.

In the DBMS field, at the present time, existing version management systems are generally dedicated to particular applications, principally around CAD. Little research has been done on database version management systems independently of specific application fields. Notably, the study of database schema evolution control is a recent subject of investigation. Our research is situated in this area. A version schema model [Palisser,90b] has been defined for the Farandole 2 DBMS [Estier,89], [Falquet,89]. This system is based on a data semantics model close to the extended Entity Relationship and the object oriented ones. But the principles of the version model are general and can be applied to every model which allows the concept of context (§3.2) to be defined.

In this paper, we start (§2) by describing our motives for taking into account schema modifications and we present the principle methods of approach for the management of such modifications. §3 explains the data model used as a basis for the version model of §4. §5 introduces the set of transformations authorised on a schema. §6 explains the mechanisms defined in order to manage data corresponding to versions of schema.

2. Schema Modification Management.

2.1. Motivations, Principal Directions.

The motives behind schema modifications stem from having to reconsider the database structure, the needs to be satisfied and the computing environment. As an example, consider a database specification for a limited set of applications. It may be possible to extend the application domain by transforming the schema. Furthermore, running certain applications may also be too expensive in terms of time because of bad data organisation. Again, access to required information may be difficult because certain useful access paths are not available.

The organisational environment can also change: new administrative procedures are created, new information circuits are put in place. Certain

This work is part of the Rebirth project supported by the Swiss Research Fundation (FNRS no 1.603-0.87)

applications use the same information in new ways, or need new information. Information modelling is changed, the schema following in order to remain conform with the application field. Finally, the computing environment in which the database is run can evolve: new versions of systems, new DBMS, new distributions of applications on different sites in the case of distributed databases. The schema must be adapted to these changes.

These different points show that a schema is rarely totally static and illustrate the need for evolution mechanisms. To fill this need, there are three principal lines of approach.

The first consists of allowing schema modifications, without retaining the pre-modification state. Each schema change is applied irreversibly to the database, without taking into account possible consequences to the data. With the second approach, the method adopted is close to that used during the database design phase. At the start, the schema evolves independently of the data. Then, after stabilisation, transformations are reflected on the data. This means that they are converted in order to correspond to the new schema. With this technique, as with the preceeding one, the evolution of the database is not controlled. The validation of a new schema leads to the destruction of its predecessor, together with the corresponding data.

In the third approach, the state of the schema before modification is conserved. This means managing a set of schema versions. There are two ways of organising this, leading to two types of version: historical and parallel.

In the historical approach, any modification having important repercussions on the schema generates a new version. Each version is kept, along with its associated data. This allows the constitution of database archives. These versions, stored in separate memory regions, are independent. Old versions are only accessible in consultation mode. Any changes are carried out on the current version. The historical approach consists of managing as many copies of the database as there are versions.

In contrast to the preceeding approach, with parallel versions the different versions of schema are stored in a common zone. They evolve in parallel and operate on the same data collection. All the versions coexist and the same set of operations is applicable to each of them. They are accessible in consultation or in update mode. Considering this approach to be the more interesting, we will develop it in the paragraphs which follow.

2.2. Previous Work.

The problem of schema version control is a recent research topic. As far as we know, the principal work done in the field has been carried out in the systems Orion [Kim,88], [Kim,89] and Encore [Zdonik,86], [Skarra,86]. In the system Charly [Palisser,89], [Palisser,90a] versions of schema are also taken into account, although the approach adopted is not comparable to that of the other two projects. Each author proposes a different solution for managing schema changes.

In Orion [Kim,89], the versions of schema are conserved. Any change to the database structure creates a new version of the complete schema. Accessible objects are associated with each version. A version thus corresponds to the complete state of the database at a given moment. This means that testing the consequences on the data of transforming parts of schema cannot be carried out by developping partial, parallel versions. For this type of experiment, a version of the entire schema must be derived. This aspect is problematic, since it can lead to managing a considerable number of versions. In practice it is often not necessary to generate such versions, particularly when the modifications are minor and only concern a small part of the schema.

The Encore approach manages versions of classes (or of types). Any modification of a class creates a new version of the class and of its sub-classes. A version of the global schema is subsequently created virtually by taking advantage of the relationships between the versions of different classes. This last point is problematic. To represent the state of the schema at a given moment, the user must choose a particular version for each of the classes defined for the state and establish links between the different versions. In addition, the derivation of a version of a class requires generation of new versions of all its sub-classes. This creates a problem when the schema contains a large number of classes and when minor changes are made to the root of the lattice. In this case, a new version must be generated of each class derived from that modified.

In the two preceeding approaches, versions of schema and of objects are considered and treated independently. For each version of a schema or of a class, there exist several versions of objects. This means that links must be established and maintained between the versions of schema and those of corresponding objects. The solution proposed in Charly (a DBMS for CAD applications) [Palisser,89] consists of not separating the treatment of versions of schema and objects. An object version contains its complete description. It does not correspond to a particular instance of a fixed schema. In this way, versions of schema and of objects are treated uniformly. This means, in particular, that schema modifications are handled in exactly the same way as those of objects. Each modification generates a new version of an object, made up of the schema and object values. This approach, "version of schema by object", gives rise, however, to a problem. Different schema versions cannot be recovered. To do this requires considering the set of versions of the data-base objects.

In [Kim,88] a fourth approach is indicated. This consists of handling schema modifications by view definitions. Any number of views may be defined on the schema. From any given view, several others can be derived, each corresponding to schema changes. Each one operates on the same data collection. It will be seen that this method is close to that adopted in the system Farandole 2.

3. The Data Model.

The data semantics model of Farandole 2, which supported the version model defined here, can be considered as an extension of the Entity Relationship model. It is based on the concepts of object, class, role and generalisation/specialisation.

3.1. Basic Concepts.

In this model, objects of the same type are grouped in a same named class. There exist two types of class: atomic and composite. The former are terminal classes, such as strings, integers, booleans, etc. Objects of these classes are identified by their value. Thus, the integer 6 is identified by the value 6.

A role has a name and a degree. It corresponds to a function defined between two classes, an origin and a domain. A role establishes a link between objects of these classes. The origin class of a role is always composite. The domain of a role can be composite or atomic.

Objects of composite classes are each represented by an identifier independent of their value. The value of an object of a composite class is a tuple made up of objects linked to it by roles. Thus, as shown in figure 1, the value of an object of the class Vehicle is a tuple made up of a licence number, its horse-power and its chassis number, which are respectively objects of the classes String, Integer and Chassis, the latter being itself a composite class.

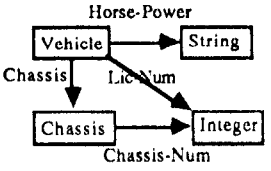

Figure 1: Origins and Domains of Classes

The roles Lic-Num, Horse-Power and Chassis-Num lead to atomic classes and can be considered as attributes. The role Chassis represents a link between two composite classes: Vehicle and Chassis.

The generalisation/specialisation mechanism allows specialisation of a class into sub-classes. We define the super-class of a sub-class C as the class from which it is directly derived and ancestors of C as being all classes higher up in the derivation hierarchy of C. A sub-class inherits all the roles of its super-class, and thus, by recursion, those of all its ancestors. The objects of a sub-class are those objects of its super-class about which particular information is desired.

A sub-class is defined by a specialisation condition. A specialisation condition is expressed as a triplet of the form (r, o, v), where r is a conditional role based on an ancestor class, o the condition operator and v its value. A sub-class is made up uniquely of the object set of its super-class verifying this condition. Every sub-class has a unique, direct super-class. This means that multiple inheritance is not authorised in Farandole 2.

Figure 2 presents an example which will be referred to in what follows. It describes the structure of an airline company. To improve readability, only those roles which link composite classes are shown in the figure.

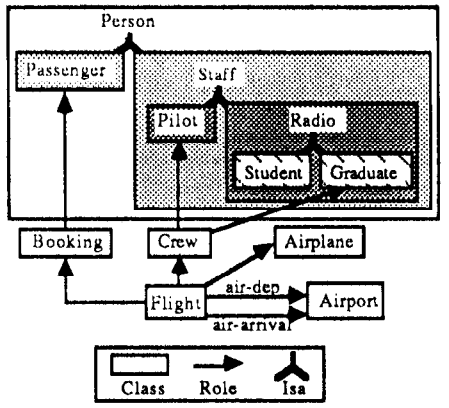

Figure 2: Airline Company

Rectangles represent classes, and arrows, roles. Sub-classes are contained one inside the other. Thus the sub-classes of Person are Passenger and Staff. Those of Staff are Pilot and Radio, etc. A crew associates a pilot and a radio operator. A flight is the association of a crew, an airplane, a departure airport and an arrival airport. Finally, a booking associates a passenger with a flight.

3.2. Contexts.

The data model of Farandole 2 was created to manage complex databases. In this field, it is often difficult for a user to understand a schema globally. The definition of partial views of the schema must also be allowed. The notion of semantic context [Falquet,89], [Falquet,91] is introduced for this reason.

A semantic context is an abstraction which allows the regrouping of certain elements of the schema while masking others. A context is used primarily to facilitate database querying. It corresponds to the particular semantics of links between constituant classes. The semantics come from the connection function [Maier,84] of the context. A connection function of a context is defined over the set of its classes. It delivers all the object tuples linked to the context. Consider a context Ct and a set C of classes $\{C_1, ..., C_n\}$ of Ct. The connection function of Ct over C delivers all the object tuples oc_t of instances of $C_1, ..., C_n$ which are associated with the roles of Ct.

A semantic context, defined on a database Db, can be symbolised by a connected graph (N, E), where N is a set of nodes and E a set of edges. Each node is a couple of the form (n, C), where n is the name of the node and C a class of Db. So the same class may appear in different nodes. An edge is a pair of nodes (n_i, n_j) labelled by a role R_i, such that R_i links the classes corresponding to n_i and n_j. The connectivity of the graph is seen through the edges and specialisation links. Any number of contexts can be defined on Db.

For example, in the schema of the airline company (figure 2), the context flight planning can be defined, as illustrated in figure 3, by the association of the following nodes and edges: Nodes: (p,Person), (st,Staff), (pi,Pilot), (r,Radio), (s,Student), (gr,Graduate), (c,Crew), (f,Flight), (a,Airplane), (a_dep,Airport), (a_ar,Airport).

Edges: {(c,Crew), (pi,Pilot)}, {(c,Crew), (gr,Graduate)}, {(c,Crew), (f,Flight)}, {(f,Flight)), (a,Airplane)}, {(f,Flight), (a_dep,Airport), (air_dep)}, {(f,Flight), (a_ar,Airport), (air_arrival)}.

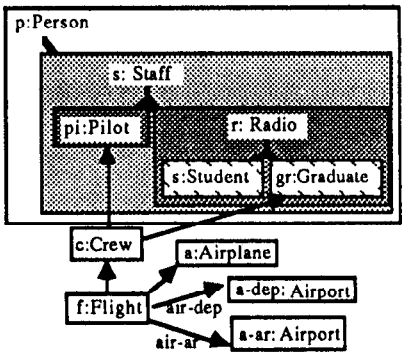

Figure 3: The Flight Planning Context

Note that if there exists only one role linking the nodes of an edge, it necessarily constitutes the implicit label of the edge and is thus not declared. This is the case of the edges {(c,Crew), (pi,Pilot)}, {(c,Crew), (gr,Graduate)}, {(c,Crew, (f,Flight)}, {(f,Flight), (a,Airplane)}.

In addition, the nodes (p,Person), (st,Staff), (r,Radio), (s,Student) are not to be found in the definitions of edges. They are however united by specialisation links to at least one node on a context edge, (pi,Pilot) and (gr,Graduate), thus assuring graph connectivity. Note finally that the class Airport participates in two nodes. This allows the rupture of the loop generated by the roles air_depart and air_arrival.

4. The Version Model.

In this paragraph, we present our version model. This model aims at managing changes made to a schema and storing its different versions.

4.1. Changeable Units.

The units which can be changed are the elements of the schema to which version management applies. In §2.2, four types of changeable units were given (versions of: schema, classes, objects with schema, views). These result in four methods of managing schema modifications.

Our approach can be considered as being the fourth one, which as far as we know has been little developped. The changeable units selected are contexts, notion close to that of views. A context corresponds to a portion of the schema. A context is made up of a set of classes associated by roles chosen among those defined in the database. It allows consultation, while masking the set of classes and roles which are not useful. It is thus versions of contexts which are to be considered in what follows.

4.2. Definition Of The Concept Of Version.

In this model, a version is defined as a stable and coherent state which the administrator or the designer desires to keep. Generating a new version of a context is a process which results from a human decision. This means that all context modifications do not necessarily generate new versions.

From a given context version several other versions of the same context can be derived. A context version may also be considered as derived from several versions. This means that the derivation organisation of versions can be symbolised by a directed graph. As in [Kim,88] and [Palisser,89], we introduce the notion of generic context to be able to globally apprehend the set of versions of a context.

4.3. The Different Types Of Versions.

We distinguish two version types, working and stable versions. Changes are always carried out on a working version. A stable version cannot be updated or deleted. A working version can be transformed into a stable one and vice versa This means that a stable version has to return to the working state in order to be modified or deleted. Queries concern stable and working versions.

Furthermore, at any time there exists one default version for each context. The default version is that which is selected when the user refers to the context without specifying a particular version. It corresponds to any version (working or stable) previously determined by the user.

4.4. Generic Contexts And Versions.

As shown in §4.2, with each version is associated a generic context. It is described as follows:

[id, name, first_version, default, [working_versions], [stable_versions], next_version, root_class]

Id is the internal identifier of the context calculated by the system. Name is an external identifier given by the designer. First_version delivers the identification of the first version of the version derivation graph. Default indicates the default version of the context. Working_versions and stable_versions correspond respectively to the working and stable versions. next_version gives the number of the next version derived for the context. Root_class indicates the root class of the generic context (cf §6).

A version is thus described as follows:

[id, gen_id, name, number,[successors], [previous], date, state, [[nodes], [edges]]]

Id is the calculated internal version identifier, Gen_id identifying its generic context. Name is the external name of the version. Number corresponds to its number: each version has a specific number which allows it to be referenced. Successors indicates the set of its successors. Previous identifies the set of versions from which it is derived. Date is the date of the last modification of the version. State says whether it is working or stable. Nodes and Edges correspond respectively to the lists of nodes and edges in the version.

5. Evolution Of The DataBase Schema.

5.1. Possible Transformation Types.

Within the adopted approach, a schema consists of a set of contexts. Each context can evolve individually into a set of versions. The transformations authorised

on the schema can thus affect it globally or partially in applying to a version. Two types of transformation are thus considered: those carried out on the complete schema and those operating on a context version. Among the former are considered the addition and suppression of context versions. Among the latter are considered the modifications of a context version. As seen in §3.2, a context is represented by a graph in which nodes correspond to classes and edges to links between classes. As a consequence, two types of modification affecting the graph are again distinguished: those which affect graph structure (addition or suppression of a node or edge) and those which apply to the contents of the graph (modification of a node or edge).

On the other hand, name changes are not allowed. This means that it is not possible to change the name of a context, a node or a role. This restriction is established for manipulation reasons. In fact, one of the objectives of the version model definition presented here is that data manipulation concerning a context should be as independent as possible of its versions. Name invariance is necessary to guarantee the invariance of the manipulation programs applied to several versions of a same context. By forbidding name changes, the connection function of a generic context remains the same for all its versions (cf. §3.2).

5.2. Sharing Version Elements.

Versions of the same or different contexts are not developped in separate databases but in the same one, from a common schema. They are not disjoint and thus often share common elements. So when an operation is tried out on a version the possible consequences on other versions must be clearly circumscribed. No modification of a version should repercute on others.

In the considered model, a version is a set of nodes and edges. A node is a named class and an edge is a role associated with a pair of nodes. Each element which has just been enumerated can occur in more than one version. For example, consider two generic contexts Ct_1 and Ct_2, which each possess versions V_1, V_2 and V_3. The same nodes and edges can occur at the same time in versions V_1 and V_2 of Ct_1 and V_1 and V_3 of Ct_2.

What is more, even if a node or edge is not shared, their elements (classes and roles) may be. Any action carried out on a set of nodes and edges of a given version thus requires verification of whether they or their elements are present in other versions. We define a set of general rules which regulate the operations of addition (R1), suppression (R2) and modification (R3) when an element (or a set of elements) is shared. By element is understood node, edge, class or role.

- **R1**. Adding an element E to a version V constitutes either an addition or a creation, depending on whether E was or was not already defined in the schema. In the first case, E is integrated in V and is thus shared by several versions. In the second case, E must be created. The creation of an element is local to V. Addition of an element also requires application of the same procedure to its elements. For example, for a node, there must be verification that the associated class is already defined.

- **R2**. Suppressing an element E in a version V simply removes the element from V. Furthermore, after this, if E is not shared by another version and is isolated, then E is effectively deleted. This process is applied recursively to the elements which constitute E. The mechanism used is thus that of garbage collection.

- **R3**. Modification of an element shared by other versions is done on a copy and has thus no effect on the other versions. For example, the modification, in a version V, of a class C figuring in several versions is carried out on a copy of C.

5.3. Modification Of The Database Schema.

5.3.1. Adding A Version.

Addition of a version can be done either by creation in the initialisation phase, or by derivation from a version of the same context.

Creation of a version automatically creates the link with the associated generic context. A name must be attributed to the version and a list of its nodes and edges must be added or created. A node is either simple or the root of a specialisation tree. An edge corresponds to an association between two nodes or to an attribute. In the former case, two composite classes are linked and, in the latter, a composite class and an atomic one. Note that a specialisation link is not an edge. This type of link is not defined at the level of edges of the graph but appears in the definition of a node (class).

The creation of the first version of a context also necessitates a choice, among the set of context classes, of a particular class, the root of the context. This class, which corresponds to a semantically dominant node, must be defined for each generic context. It is the entry point to the context and is associated with all its versions. Its functions will be detailed in §6.

As an example, suppose that the first version of the generic context Flight planning has been created (cf. Figure 3, §3.2). A version of a new generic context (figure 4) can be created to manage reservations for the airline.

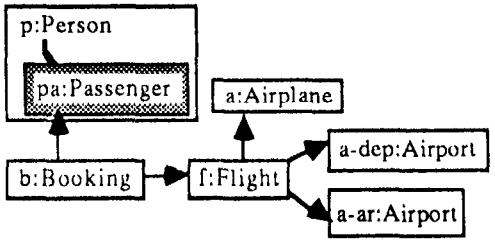

Figure 4: Version 1 Of The Context Reservation

Note that the nodes (p:Person), (f:Flight), (a:Airplane), (a_dep:Airport) and (a_ar:Airport), already defined in version 1 of the context Flight planning (figure 3, §3.2), are shared by this new version. But the nodes (pa: Passenger) and (b: Booking) did not exist in the schema and were thus created.

Derivation of a new version is always done from a previous version. A name must be given to the new version. It inherits by default the set of nodes and edges of the version from which it is derived.

5.3.2. Suppression Of A Version.

When a version is suppressed, all its elements become inactive. The suppression rule R2 (cf. §5.2) must be respected. The version elements are effectively suppressed if they are not linked to any other element of the version set of the database. Suppressing a version V leads to the suppression of the links with all its derived versions which are linked now to the versions from which V was derived.

5.4. Modification Of A Version.

This paragraph presents the set of modifications which can be made to a context version. It should be remembered that, in the definition of a context version (cf. §4.2), a modification does not necessarily generate a new version.

5.4.1. Modification Of The Graph Structure.

• Addition Of A Node.

Following rule R1 (cf. §5.2), if the node exists in the database, it is added to the considered version, otherwise it is created. A node can correspond to a super-class or a sub-class of a specialisation tree. In the first case, only this node is integrated into the version. In the second case, all its ancestors must also be added. To create a node, it must be given a name and be associated with a class, which has to be created if it does not exist. If it is a sub-class, its super-class must be specified together with a specialisation condition.

• Adding An Edge.

The procedure for adding an edge is similar to that for a node. It respects the sharing rule (R1). A role and a pair of nodes must be associated with the edge. The nodes must have been previously defined for the considered version. If the role does not exist, it is created by giving it a name and associating an origin and a domain and specifying its degree of valuation. The two classes of the role must correspond to those figuring on the nodes of the edge.

Consider the case in which the airline company needs to take into account booking goods on flights. A new version (n°2) of the reservation context is derived. To this version are added the following nodes: g: Goods, f: Freight and pb: Pas-Book. They correspond, respectively, to the description of goods, the booking of these on a flight and the booking of passengers on the flight. The classes Goods, Freight and Pas-Book, which did not exist in the schema, are created as in figure 5. The edge defined between the two nodes is made up of a new role. It is created by the attribution of a name (goods_book), an origin class (Freight), a domain (Goods) and a valuation degree of 1.

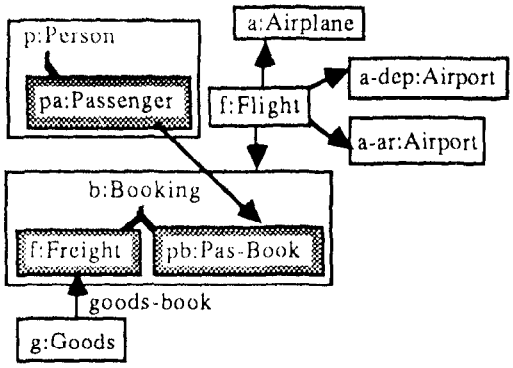

Figure 5: Version 2 Of The Context Reservation.

• Suppression Of A Node.

Suppressing a node consists of taking it out of the version, together with all its associated edges. This operation also removes the class associated with the node, as long as it is not shared by other nodes of the version (rule R2).

If this class is the super-class of a specialisation tree, then all its derived classes are also removed from the version. For example, in figure 5, if the node g: Goods is removed from the version, the edge between g: Goods and f: Freight is also removed, as is the role goods_book. They are, however, effectively deleted only if they are no longer linked to any element of the set of versions of the database.

• Suppression Of An Edge.

Suppressing an edge cuts the link between two nodes. That is it removes the role figuring on the edge. This latter operation only affects other version elements if the role is conditional, that is defines a sub-class C. In this case C and all its derived classes are removed from the version. This operation must not invalidate the connectivity of the graph or the rule R2.

5.4.2. Modification Of Graph Contents.

• Modification Of A Node.

Name changes not being allowed, node modifications are equivalent to those of classes. Amongst these are considered the redefinition of the super-class of a class and of a sub-class.

For reasons connected with objects, redefining the super-class of a class C can only be carried out inside a specialisation tree. In a specialisation tree, all objects are defined at the level of the ancestor class. Thus the transfer of a sub-class C of one tree to another would lead to the suppression of all the objects of C. What is more, this modification must not introduce a loop. The new super-class of C must not be derived from C. For example, consider the specialisation tree of figure 6.

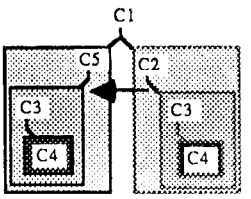

Figure 6: Modification Of A Super-class.

In this example, the super-class of C_3 is redefined. This operation breaks the link between C_2 and C_3 and creates a new link between C_3 and C_5. The new super-class of C_3 cannot be situated outside the tree and must not be a sub-class of C_3.

Following rule R3, if the class to be modified is shared, the modification is done on a copy. Furthermore, copies of all classes down the specialisation hierarchy must be generated. In the preceeding example this comes down to copying classes C_3 and C_4.

In the flight planning context (figure 3, §3.2), widening the concept of student to all members of staff could be required. As is illustrated by figure 7, this comes down to modifying the super-class of the class Student which passes from Radio to Staff. This transformation requires the derivation of a new version (n°2) of the context.

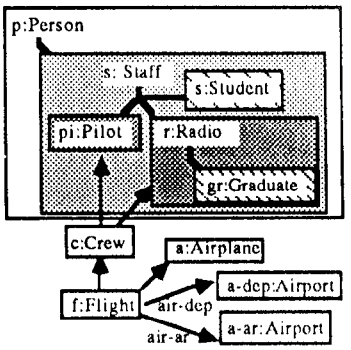

Figure 7: Version 2 of Flight Planning Context.

The modification of a sub-class applies to its specialisation condition, that is to the conditional role, the operator or the value. If a new conditional role is attributed to a sub-class C, the role must have been previously defined in one of the ancestor classes of C. In the preceeding example (figure 7) the classes Person, Staff and Pilot are defined as follows:

Person(Name:string; Age:integer; TypePerson:string)
Staff sub-class of Person
if TypePerson = "employee"
(Salary: integer; Function: string)
Pilot sub-class of Staff **if** Function = "pilot"
(NbFlightHours: integer)

The conditional role Function of the class Pilot can be changed by choosing TypePerson as a new conditional role, since it is defined in the ancestor class Person.

- Modification Of An Edge.

Changes allowed on edges can affect their origin and terminal nodes, as well as the valuation degree of their roles. Because of name invariance, this operation is equivalent to a role modification.

Modification of the origin or terminal nodes of an edge can only be done inside a specialisation tree, for reasons similar to those discussed for classes. The operation must respect rule R3. Consider a role R_1 having as origin and terminal nodes respectively N_3 and N_6, illustrated in figure 8.

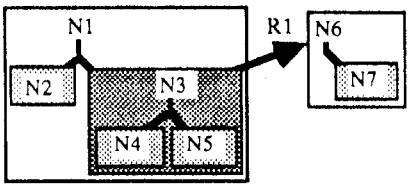

Figure 8: Modification Of Nodes Of A Role.

The only possible new origin nodes of R_1 are N_1, N_2, N_4 or N_5. Replacing N_3 by a node outside the tree comes down to suppressing all the object associations supported by R_1. The same is true for the terminal nodes. From this, the new terminal node of R_1 can only be N_7.

For example, the edge associated with nodes b: booking and pa: Passenger of version 1 of the context reservation (cf. figure 4) goes through the role pass_book, which has the class Booking as origin. After the specialisation of this class in sub-classes Freight and Pas-Book (cf. figure 5), the role pass_book can be modified and a new origin class, Pas-Book, attributed.

While increasing the valuation degree of a role creates no problem, its diminution has consequences on the data. If, in an object, the number of values of a role is greater than the new valuation degree, the object receives an unknown value for the role. Let o_1 be an object to which a role R_1 associates the values $\{v_1, v_2, v_3\}$. If the valuation degree of R_1 is changed to 2, then o_1 receives an unknown value for R_1.

5.5. Rules Associated With Modifications.

We define a set of rules which must always be followed when the database schema is transformed. The operations presented in §§5.3 and 5.4 can thus only be carried out if they do not violate these rules. This guarantees the coherence of a schema after modification.

- **R4**. The set of nodes and edges of a version must form a connected graph. There are thus no isolated nodes in the graph and each of them can be reached from the root node or class.
- **R5**. If a node belongs to a version, then all its ancestors must also belong to the same version.
- **R6**. The role of an edge must necessarily link the two classes defined in the nodes.
- **R7**. The root class of a generic context can neither be suppressed nor modified.

5.6. Schema Of A Database.

With the defined model, a user can consider a schema either as a set of context versions or as a view of the set of versions. In this latter case it is defined by choosing a set of context versions. In the preceeding paragraphs two generic contexts were defined which have each two versions (Flight planning (Version 1, figure 3 (§3.2), Version 2, figure 7 (§5.4.2)) and Reservation (Version 1, figure 4 (§5.3.1), Version 2, figure 5 (§5.4.1))). A schema of the database Airline can be defined by choosing version 1 of Flight planning and version 2 of Reservation.

Note that different versions of the same generic context can figure in the same schema. For example, a schema can be made up from versions 1 and 2 of the context Flight planning.

Version selection is done statically or dynamically. In the first case versions are referenced by their identifiers and those of their respective generic contexts. In the second case, only the identifiers of the generic contexts are specified. Version selection is carried out by choosing default versions (cf. §4.3).

6. Object Management In A Version.

As was already underlined in the introduction, an important problem in schema version management is the establishment and maintenance of correct links between different versions of the schema and the objects. In particular, schema modification must not lead to the loss of data. Mechanisms have therefore been defined which allow the association with each context version of the objects pertinent to that version. It should be remembered that the subject covered here is not object version management but only schema version control. For this reason, no account is taken of the evolution of objects in a context version.

6.1. The Root Class Of A Context.

As seen previously (§5.3.1), with each generic context is associated a unique root class. It corresponds to a node of the context. This notion exists in order to determine whether an object belongs to a given version. The class is specified by the user. It is part of the information common to the different versions of a generic context. This means that it is associated with its set of versions and can neither be modified nor suppressed between one version and the next. From a semantic point of view, it is an entry point in the context which corresponds to a semantically dominant node.

In the case where no specialisation tree exists in the context, the root is any class. Otherwise, it must not be a sub-class. For example, for the generic context reservation the root class is chosen to be Booking (cf. §5.3.1, figure 4) and in the context flight planning, the class Flight (cf. §3.2. figure 3).

6.2. Context Versions And Objects.

6.2.1. Objects Of The Root Class.

The notion of root class allows determination of which objects belongs to a context version. With each object of this class is associated the set of versions in which it appears. For example, let C_1 be the root class of a context CT_1 which owns the version set $\{V_1, V_2, V_3, V_4\}$ and let o_1 be an object of C_1. If o_1 appears in versions V_1 and V_2 of CT_1 and not in V_3 or V_4, then the set $\{V_1, V_2\}$ is associated with the object o_1. By default, the context versions in which an object does not participate are those which are not specified for the object.

For example, the root class Booking of the generic context reservation owns, in addition to its roles, a multi-valued role (versions) which associates with each object the list of versions to which it belongs.

Booking

id	versions	booking-nb	flights
34	[1]	678	[sw789,af235]
57	[2]	879	[bc564,ib672]
89	[1,2]	345	[ba234]

Figure 9: The Root Class Of The Context Reservation.

6.2.2. Objects Belonging To A Version.

An object o of a class (different from the root class), which is a component of a version V, belongs to V if an object oc of the context version V can be built from o by applying the connection function and after verifying the integrity rules.

For example, figure 10 shows the objects of version 1 of the context reservation. Thus, Paris airport belongs to this version since it is linked to flight ba234, which is linked in turn to the object of the root class, the booking 345, which is declared as belonging to the version. The airplane DC10 also belongs to this version because even if it is not linked to any object of the root class at present, it will belong to the result obtained by applying the connection function to this version.

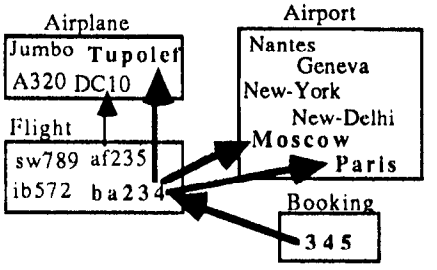

Figure 10: Objects Belonging To A Version.

Now let Flight-Passenger and Military-Flight be two exclusive sub-classes of Flight. Let the new version V' of the context reservation contain the classes Booking, Flight-Passenger, Flight, Airplane and Airport. Then all the objects of Military-Flight are also objects of Flight, but they cannot be part of a V' object obtained by applying the connection function. Thus they do not belong to V'.

6.3. Sharing A Root Class.

In a database, different generic contexts can share the same root class. In this case, it is necessary to indicate for each object of the class and for each generic context

the versions in which they appear. This comes to associating with each object of the root class a pair, whose first member is a context identifier and the second a set of versions.

Let Ct_1, Ct_2, Ct_3 be generic contexts, each owning the set of versions $\{V_1, V_2, ..., V_n\}$ and having the same root class C_1. Consider the set of objects $\{o_1, o_2, ..., o_n\}$. For each object of the set it must be indicated not only in which versions it appears, but also to which generic context it belongs. For example, if o_1 is defined in versions V_1 and V_2 of contexts Ct_1 and Ct_3, the following couples are associated with o_1: $(Ct_1, \{V_1, V_2\}), (Ct_3, \{V_1, V_2\})$.

6.4. Object Creation, Suppression And Updating.

An object is always created, suppressed or updated in a working version V of a context. It can either be created in the root class, or in any other class. In the first case, if the object already exists in the database, in other versions, it is automatically integrated into V and marked as belonging to it. If it does not exist, it must be created and marked as belonging to V.

When an object of the root class is suppressed, if it appears in no other context version than V, it is effectively removed from the database. The links between the object and objects of other classes are broken. If the object is shared between several versions, it is marked as no longer belonging to V. Following the possession rules between an object and a version defined for the root class (cf. §6.2.1), all objects which reference it no longer appear in V.

The creation and the suppression of an object in a class other than a root class are the usual operations of creation and suppression.

Updating an object consists of modifying the value of a role. If an object of a root class is shared by several generic context versions, it can not have different values for these versions. This means that object updating is reflected on all the versions in which it appears. One of the primary principles announced is that this work applies not to object versions but uniquely to the evolution of the database schema.

7. An Evolutionary Model.

We will show briefly how this version management model can be considered as an evolutionary database model. Indeed it will undoubtedly improve flexibility in the use and transformation of a database. It also opens new perspectives in the database design process.

7.1. Independance Between Manipulation Programs And Schema.

An originality of this model resides in the improvement of the invariance of manipulation programs. Most manipulation programs which are applied to a context version V do not require any rewriting before being applied to another version V' of the same context. Of course rewriting is necessary if a class belonging to V and missing in V' is needed. This property comes from the invariance of names (§5.1) and the use of contexts [Falquet,89]. Indeed, schematically a context can be considered as a large object and the logical data access can be written without knowledge of all the classes and roles which compose it.

7.2. Object Life Cycle.

Another originality of this model is its facilities for designing and easily implementing object life cycles. Roughly speaking we consider that an object life cycle can be divided into several periods [Guyot,86]. Each period defines an object environment in terms of data, integrity rules and processes. So a period provides the set of data which may be linked to the object, the set of integrity rules which are defined, the set of processes which may be executed. The object environment changes when an object leaves a period and enters a new one. This version management system allows an object life cycle to be designed and easily implemented. The various periods correspond to context versions and the root class of a context is the class of objects the life cycle of which has to be implemented.

7.3. A New Solution To An Old Problem.

The version management system can help to solve a concrete problem which we introduce with an example. A student, a faculty and a diploma have respectively a number as identifier and a name. A diploma is delivered by only one faculty. A student can be inscribed at only one diploma d and in only one faculty, which must be the faculty delivering the diploma d. In order to avoid any redundancy the relational schema will be:

St(St# St-Name) Dpl(Dpl# Dpl-Name)
Fac(Fac# Fac-Name)
R(St# Dpl#) S(Dpl# Fac#).

In fact there are two periods. Firstly every student is allowed to choose a faculty, without choosing a diploma. After three months every student has to choose a diploma among the diplomas of the previously chosen faculty. In order to store the facts concerning the first period has the relation T(St# Fac#) to be implemented? If so, it will be redundant in the second period.

With our approach a context Registration is built: it is composed of St and Fac. St is its root class. Then two versions of this context are built : the first one is composed of St, Fac and T and corresponds to the first period. The second is composed of St, Dpl, Fac, R and S and corresponds to the second period. This solution does not contain any redundancy.

8. Conclusion.

In this paper, a version model is proposed which allows following the evolution of the database schema. The version management mechanism is based on the notion of context, which can be considered as an extension of the concept of view. Transformations are carried out on parts of the schema. Any number of contexts can be defined on the database, each one corresponding to part of the schema. Several versions can be derived from a context. The method adopted is thus close to that which consists of managing versions of views. We can compare it with the "schema version" and "class version" approaches. Roughly speaking the

granularity of the former seems to us to be too wide: any schema transformation, even if it concerns only one class, needs a version of the whole schema. In the other hand, the granularity of the latter seems to us to be too narrow: each class is allowed to have several versions and so associations in a schema between classes must follow the various versions. The problem becomes complex. The granularity of context seems to us to be more appropriate.

As we showed in the last paragraph, the version model introduces a new approach for designing a database. The concept of object life cycle can be used, the database design process may be evolutionary. Furthermore the model improves flexibility in the use of databases: there is real independence between data schema and data programs.

The DBMS Farandole 2 is in fact a laboratory written in ADA which includes a classical DBMS (built from ECRINS [Junet,86]) and a process environment. The concept of context is implemented. The elementary data schema transformations are implemented. Version mechanism implementation is in progress.

9. Acknowledgements.

The authors thank Michael Griffiths for his help in translating this paper.

10. References.

J. Autran, C. Palisser: Vues et Versions d'Objets Complexes–Une Application à la CAO, en Architecture; *3èmes Journées Bases de Données Avancées, INRIA*, Port Camargue, May 1987.

D. Batory, W. Kim: Modeling Concepts for VLSI CAD Objects; *ACM TODS, Vol. 10, N°3*, September 1985.

H.T. Chou, W. Kim: A Unifying Framework for Version Control in a CAD Environment; *12th VLDB Conference*, Kyoto, August 1986.

D.J. Ecklund, E.F. Ecklund, R.O. Eifrig, F.M. Tonge: DVSS: A Distributed Version Storage Server for CAD Applications; *13th VLDB Conference*, Brighton 1987.

J. Estublier, S. Ghoul, S. Krakowiak: Preliminary experience with a configuration control system for modular programs; *ACM SIGSOFT-SIGPLAN*, April 1984.

T. Estier: Le Modèle Farandole 2 et le Dictionnaire du SGOOD; *CUI Research Report*, Geneva, September 1989.

T. Estier, G. Falquet: QFE: un générateur d'interfaces pour l'interrogation des bases de données à l'aide de contextes sémantiques; *Inforsid, ed. Eyrolles*, Biarritz, May 1990.

G. Falquet: Interrogation de bases de données à l'aide d'un modèle sémantique; *Thesis, Geneva University*, May 1989.

G. Falquet: F2 an object-oriented database model with semantic contexts; *CUI Research Report*, Geneva, January 1990.

M.C. Fauvet: ETIC, un SGBD pour la CAO dans un environnement partagé; *Thesis, Grenoble University*, September 1988.

J. Guyot: Un modèle de traitements pour les bases de données: un formalisme pour la conception, la validation et l'exécution de la spécification d'une application; *Thesis, Geneva University*, June 1986.

M.Junet, G. Falquet, M. Léonard: ECRINS/86: an extended entity-relationship data base management system and its semantic query language; *12th VLDB Conference*, Kyoto, Japan, August 1986.

G.E Kaiser, A.N. Habermann: An Environment for System Version Control; *in Digest of papers COMPCOM Spring 83, IEEE Computer Society*, San Francisco, 1983.

R.H. Katz, T.J. Lehmann: Database Support for Versions and Alternatives of Large Design Files; *IEEE Transactions on Software Engineering Conference*, Vol. SE 10, N°2, March 1984.

R.H. Katz, E. Chang, R. Bhateja: Version Modelling Concepts for Computer-Aided Design Databases; *ACM SIGMOD Conference on Managment of Data*, May 1986.

R.H. Katz, E. Chang: Managing Change in a Computer-Aided Design Database; *13th VLDB Conference*, Brighton 1987.

W. Kim, D. Batory: A Model and Storage Technique for Versions of VLSI CAD Objects; *Conference on Foundations of Data Organization*, Kyoto, May 1985.

W. Kim, H.T. Chou: Versions of Schema for Object-Oriented Databases; *14th VLDB Conference*, Los Angeles, August 1988.

W. Kim, N. Ballou, H.T. Chou, J. F. Garza, D. Woelk: Features of the ORION Object-Oriented Database System; in Object-Oriented Concepts, Databases and Applications, ed. W. KIM and F. M. Lochovsky, *ACM Press Frontier Series*, New York, 1989.

P. Klahold, G. Schlageter, W. Wilkes: A General Model for Version Management in Databases; *12th VLDB Conference*, Kyoto, August 1986.

D. Maier, J.D. Ullman, M.Y. Vardi: On the Foundation of the Universal Relation Model; *ACM TODS*, Vol.9, N°2, 1984.

C. Palisser: Charly, un Gestionnaire de Versions pour la CAO en Architecture; *Thesis, Aix-Marseilles University*, Marseilles, November 1989.

C. Palisser: Le Modèle de Versions du Système Charly; *6èmes Journées Bases de Données Avancées, INRIA*, Montpellier, September 1990.

C. Palisser, J. Andany, M. Léonard: Un Modèle de Versions de Schémas de Bases de Données; *CUI Research Report*, Geneva, July 1990.

M.J. Rochkind: The Source Code Control System; *IEEE Transactions on Software Engineering*, Vol. SE-1, N°4, December 1975.

A.H. Skarra, S.B. Zdonik: The Management of Changing Types in an Object-Oriented Database; *OOPSLA Conference*, Portland, September 1986.

W.F. Tichy: RCS-A System for Version Control; *Software Pratice and experience*, Vol. 15(7), July 1985.

S.B. Zdonik: Version Management in an Object-Oriented Database; International Workshop. Trondheim, June 1986. Ed Reidar Conradi et al. *Lecture Notes in Computer Science* N°244.