HAL
open science

Schema change propagation in object-oriented databases

G.T. Nguyen, D. Rieu

To cite this version:

G.T. Nguyen, D. Rieu. Schema change propagation in object-oriented databases. [Research Report] RR-1045, INRIA. 1989. inria-00077184

HAL Id: inria-00077184

https://inria.hal.science/inria-00077184v1

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1045

Programme 4

SCHEMA CHANGE PROPAGATION IN OBJECT-ORIENTED DATABASES

Gia Toan NGUYEN
Dominique RIEU

GROUPE DE RECHERCHE
GRENOBLE

Juin 1989

RR_1045

SCHEMA CHANGE PROPAGATION IN OBJECT-ORIENTED DATABASES

NGUYEN G.T[1] & D. RIEU[2]

[1]INRIA & [2]IMAG
Laboratoire de Génie Informatique
BP 53 X
38041 GRENOBLE Cedex
France

nguyen@imag.imag.fr

Abstract

This paper gives an overview of current research efforts directed towards evolving data definitions in object-oriented database systems. The emphasis is on their ability to support two complementary aspects : supporting evolving schemas, and propagating the changes on the object instances.
Several projects are analyzed : *Cadb, Encore, GemStone, Orion,* and *Sherpa.* Current results indicate that if most of them provide schema evolution facilities, they seldom support automatic propagation mechanisms.
A proposal is described that enables *Sherpa* to fully support the propagation of changes and the dynamic classification of the instances whose class definitions are modified. This approach is an extension of techniques used in artificial intelligence for knowledge representation. It extends previous classification mechanisms with a dynamic capability which adequately supports evolving class definitions.
Key-words : classes, objects, evolving schemas, classification, modifications, propagation.

PROPAGATION DES MODIFICATIONS DE SCHEMA DANS LES BASES D'OBJETS

Résumé

On fait une synthèse des recherches en matière de définitions évolutives dans les systèmes de gestion de bases d'objets. L'accent est mis sur leur aptitude à mettre en oeuvre deux aspects complémentaires : des schémas évolutifs et la propagation de leur modification sur les instances.
Plusieurs projets sont étudiés : *Cadb, Encore, GemStone, Orion et Sherpa.* Il apparaît que si la plupart d'entre eux permettent de modifier les schémas, peu nombreux sont ceux qui permettent de propager ces modifications aux données automatiquement.
On présente une méthode qui permet de réaliser ceci dans *Sherpa* pour reclasser automatiquement les objets dont les définitions ont changé. C'est une extension de méthodes utilisées en Intelligence Artificielle pour la représentation des connaissances. Elle étend les mécanismes de classification automatique existant par une approche dynamique qui permet de gérer de manière appropriée des définitions de classes évolutives.
Mots-clés : classes, objets, schéma évolutif, classification, modifications, propagation.

1. INTRODUCTION

Several projects are currently underway for the development of object-oriented database systems, e.g *GemStone, O2* and *Orion* [4, 5, 23]. They are intended to support software engineering, office automation and CAD/CAM applications. As such, they provide facilities to define, store and retrieve shared and persistent composite objects [19]. Most of them support changes in the object definitions i.e., evolving database schemas [11, 13, 21, 22]. However, very few have the ability to propagate the changes on the corresponding instances. This paper proposes an original contribution to this issue. The fundamental assumptions here are : the existence of an object-oriented paradigm for defining and manipulating the data, its implementation and usage in an engineering design environment, and as a consequence, provision for the data definitions (i.e. the database schema or the object class definitions in more specific terms) to interactively evolve during the application lifetime, to cope with the evolution of the data and programs.
Section 2 presents an overview of various interesting functionalities for object-oriented database systems. Several object-oriented systems are analyzed with respect to these functionalities in Section 3. The emphasis is on schema evolution. The ability to handle the changes and propagate them on the object instances is detailed. It results that no system provides a full support for object evolution resulting from changes in the class definitions and the class relationships. A proposal is made in Section 4 to control the modifications performed on the schemas and take automatically into account their impact on the object instances. Section 5 is a conclusion.

2. SCHEMA EVOLUTION IN OBJECT SYSTEMS

In the following, the reader is supposed to be familiar with the object-oriented paradigm and terminology [14, 15, 16].
An object is defined by a *structure* and an *interface*. Objects are grouped into *classes* which are collections of *instances* corresponding to similar structures and interfaces. Classes are related in the inheritance lattice by the *super-class/ sub-class* relationship. A database *schema* is a set of class definitions connected by the super-class/ sub-class relationship. It is represented by a class lattice.
Composite object are defined using references to sets of instances of other classes.
In Figure 1, STOL (short take-off and landing), MEDIUM range and AMPHIBIAN aircraft classes inherit the instance variables from the class AIRCRAFT i.e., "type", "mtow" (maximum take-off weight), "fuel" capacity and "range", in a class lattice. The STOL class groups those aircraft with stol capability, giving their "take-off" and "landing" distances. The MEDIUM range class describes the "safety" equipment for those particular aircraft with a specific "range" . The AMPHIBIAN class gives the "floats" characteristics for those particular aircraft with floating equipment [6]. The SAM-AIRCRAFT class groups the instances of the aircraft which are simultaneously STOL, AMPHIBIAN and MEDIUM range. An additional instance variable "certified" gives their particular certification date.

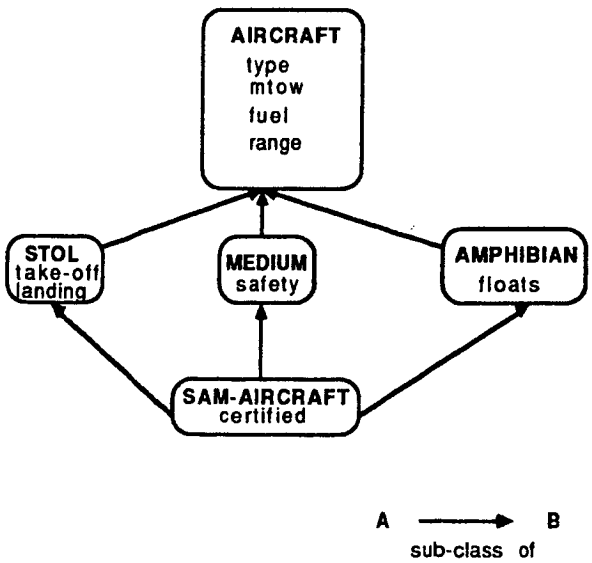

Figure 1. Class lattice for aircraft.

2.1 Schema change operations

Schema change operations fall into three categories : changing class definitions i.e instance variables or methods, modifying the class lattice by changing the relationships between classes, and adding or deleting classes in the lattice.
Changing class definitions includes : adding or deleting new instance variables and methods in a class definition, and modifying existing instance variables and methods, e.g changing their name, their domain or constraints.
Creating new specialized classes from existing classes is a basic constructor of object-oriented schemas. Defining the classes LONG and SHORT range by specialization of the class AIRCRAFT provides the ability to characterize those aircraft with specific range constraints (Figure 2) .
Changing class relationships may be used for the incremental definition of objects or to model a new semantics : for example specific instances of MEDIUM range aircraft (e.g the above A300s) are also instances of the class EXTENDED range (e.g A300-600R). This is modelled by adding a specialization relationship between the corresponding classes in the lattice. Further, EXTENDED range can be made a sub-class of the LONG range class by adding a new relationship, since they must bear similar capabilities and equipment. In the example, the result of these changes is that : the lattice reflects the fact that EXTENDED range aircraft are designed as specific MEDIUM range aircraft, they must comply with the definition of the LONG range class.
Name conflicts may result from changes in the class relationships. When multiple inheritance is supported, ordering the super-classes of a given class may avoid to some extent these conflicts. Overriding allows also locally defined instance variables and methods in a class to redefine inherited instance variables and methods. Further, if instance variables and methods are transitively inherited from the superclasses of a class, constraints and domain conflicts can

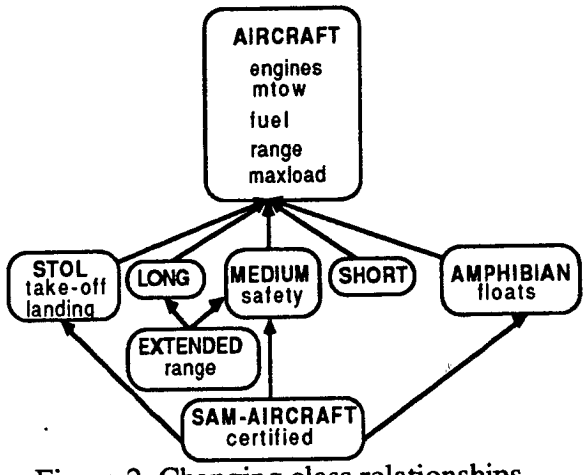

Figure 2. Changing class relationships.

occur. For example, range constraints in the classes STOL, MEDIUM and AMPHIBIAN may overlap. A subclass of SAM-AIRCRAFT should then inherit the most constrained domain for its range instance variable.

2.2 Other relevant issues

Incremental specification of the design artifacts requires changing dynamically the class definitions, hence the database schema. While composite objects are often made available, the notion of *dependent* objects is seldom supported. Composite object definition and manipulation is a major issue in many applications today [3]. The notions of composite objects and dependent objects are differenciated [7, 9, 12]. For example an aircraft is composed of a fuselage, wings, engines and a landing gear. It is indeed a *semantic* relationship between the instances involved. This is shown in bold lines in all subsequent figures.
Object versions are also relevant to object-oriented database design.*Generic instances* may be used to model the version derivation hierarchy for a given class (Figure 3). They may be used to reference objects without specifying in advance the particular version needed.
Specific versions in the derivation hierarchy are called *version instances*. As shown in Figure 3, CFM56 is a version instance of the generic instance "turbo-fan engine".

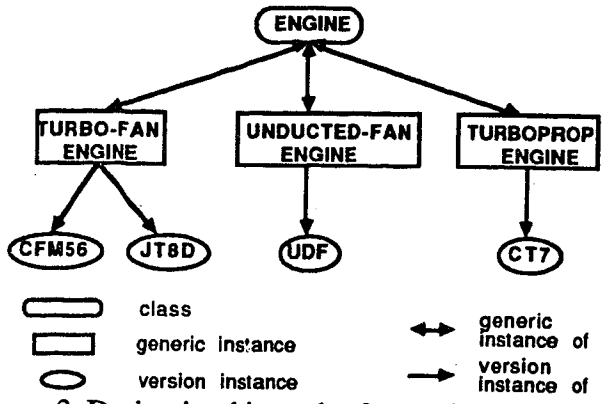

Figure 3. Derivation hierarchy for engine versions.

3. SYSTEMS SURVEY

If most object-oriented systems bear only slight variations in the basic concepts they implement, the major differences are in their support for schema evolution. A limited survey of existing prototypes is given in this section, with emphasis on the issues discussed in Section 2. This includes *Cadb, Encore, GemStone, Orion* and *Sherpa.*

3.1 Schema change operations

Most systems provide a limited set of schema change facilities. Modifying the domain of an instance variable is allowed in *Orion* by generalization only, and within the limits of the inherited instance variables' domains. This avoids values of existing instance attributes to become illegal with respect to an updated domain.
In *GemStone,* modification of constraints on instance variables is limited to the specialization or generalization of their domain [11]. Similar changes are allowed in *Encore.* They are handled in the latter by creating systematically new versions of the classes.
Changes to the schema yield new versions of classes in *Encore.* For example, deleting a class in the lattice provokes the creation of new versions of its sub-classes, which automatically inherit the instance variables of its super- classes.

3.2 Composite objects

As mentionned previously, composite objects are basically structural aggregates of sub-parts involving instances of component classes. A *dependency* relationship between the components and the "owner" object must be provided if the semantics of the composite object is to be applied on the components. For example, composite objects are instantiated as a whole in *Loops,* and deleted as a whole in *Orion.* This dependency is system-defined in *Loops,* but generic in *Orion* i.e., class-defined. In contrast, *Cadb* - an acronym for Computer-Aided Design Data Base - supports the dependency relationship at the instance level. Instances of the components are exclusively owned by one *parent* instance in the composite object class, which defines its *context.* They may be shared by other instances.

3.3 Versions of schema, classes and instances

A major goal in *Encore* is that the modification of object types i.e., class definitions, should remain transparent to the application programs [13]. The emphasis is on the preservation of the behavior of the objects using versions of classes. An error-handling mechanism provides the correct version for a class corresponding to a specific message version, and vice-versa. An object instance belongs to exactly one specific version throughout its lifetime. All versions of classes can be modified and instantiated at any time. Versions of sub-classes are created automatically upon creation of new versions of classes. *Orion* extends the notion of version to that of versions of schema [17]. As described previously, *generic instances* are provided to support versions of instances [3]. Like *Cadb,* it supports versions of composite objects. Neither *GemStone* nor *Loops* seem to support explicitly the notion of version.

3.4 Propagation of changes on class definitions

Schema changes in *GemStone* are controlled using a set of invariants that define the legal configurations of the class hierarchy. For example, changing the name of an instance variable in a class is propagated to its subclasses, provided they do not redefine it locally. In contrast with *Orion*, a class may not be deleted in *GemStone* and *Encore* if there are any existing instances. Since no reference to deleted classes and instances are allowed, classes referencing deleted ones are forced to refer to their immediate superclass. For example, deleting the class AMPHIBIAN aircraft in Figure 2 enforces direct inheritance between the classes AIRCRAFT and SAM-AIRCRAFT.
In *Orion*, one can change the name or add new instance variables in a class definition. This is propagated to the extent that no name conflict and no local redefinition appears in the sub-classes. The domain of an instance variable can only be generalized, thus avoiding any impact on existing instances [1].
In *Encore,* no propagation of schema changes is supported. Rather, a specific notion of *compatible version* is implemented to handle the mismatch between the successive definitions of an object class and the corresponding methods. An error-handling mechanism provides for the correct mapping between the various object versions and the corresponding methods. Associated with each class is a *version set interface* which is the union of the specific version interfaces. It includes only the least constrained instance variables and methods, thus providing a potential interface for all versions. The refinement corresponding to each particular version is dealt with by the error-handling mechanism which is attached to each particular version [13].

3.5 Propagation of changes on object instances

Schema modification is not *per se* an issue. Of primary concern is the capability to provide some form of controlled side-effects on the object instances. The spectrum lies between a fully automatic propagation of the changes and a manual one.
The first approach is used in *GemStone* and *Orion*, while the second is that of *Encore*. An explicit *convert* operator has to be invoked by the user in order to modify in *Encore* an instance and conform it to a modified class definition.
Another relevant issue for change propagation is the delay by which the modifications are actually performed on the object instances. Propagation can be immediate or deffered.
Immediate propagation is adopted in *GemStone*. It is called *conversion*. The impact of schema modifications is immediately implemented on the instances involved.
Deffered propagation is used in *Orion*. It is called *screening*. The side-effects are propagated only when the instances are accessed. The first solution emphasizes consistency and information preservation. It also sacrifices performance. One advantage is that it limits the propagation to the execution of epilogs in the execution of methods. The second solution emphasizes performance but requires a permanent propagation mechanism throughout the system's lifetime.
Invariants preserving rules in *Orion* also avoid most propagation problems on object instances. Screening deleted instance variables and adding default or nil values to an instance variables are performed when fetching the modified class' objects.

4. DYNAMIC PROPAGATION OF CHANGES

Propagating the schema changes on the object instances is a capability that systems intended to support engineering applications must provide [21]. It allows the designers to iteratively check the side-effects of the data manipulations and to incrementally design the artifacts. It also helps controlling their consistency with respect to existing objects and to the design rules. While most systems support schema evolution, they seldom support automatic propagation of the changes. A proposal is made in this section to address this issue. It relies on techniques like classification, which are usually not provided in database environments, but rather in artificial intelligence for knowledge representation.

4.1 Incremental design of objects

Dynamic inheritance and classification provide a means to automatically propagate schema changes on the object instances. *Cadb* is one of the first prototype to implement such mechanisms, though no explicit notion of messages is implemented [12]. Rather, calculated properties are made available to define instance variables whose values depend on other instance variables from one or more objects.
Cadb is intended to minimize the restrictions usually burdening propagation of changes in other engineering database systems. Object classes are defined by specification rules i.e., instance variables, derivation rules and integrity constraints. Object instances are grouped into sets that instantiate the classes. *Composite objects* are taken into account together with the notions of *dependent objects* and *context*. In contrast with other proposals, both *top-down* and *bottom-up* design of composite objects are simultaneously supported. This allows the instantiation of *partially known objects* and the incremental design of large objects from existing components. Mixing both approaches is possible i.e., include existing components and later reference new components yet unknown.
In contrast, *Orion* allows only top-down design i.e., components may only be instantiated if their parent exists [3]. For example, engines cannot be created if the corresponding aircraft does not exist. However, for most aircraft and engine manufacturers, items are designed and fabricated independently. This allows aircraft to be equipped with various powerplants from one model to the other (B737, Airbus,...), from one version to another (B737-100, B737-200,...) and from one serial number to another, depending usually on the carrier airline requests. Similarly, older aircraft are periodically refitted with new engines. This implies replacing the engine component for those aircraft involved. While adding the new instances of engines is no problem, keeping the old ones aside is not possible if a composite link exists between the classes AIRCRAFT and ENGINE. Stated otherwise, the notion of dependent object as defined in *Orion* is here too strong.

4.2 Relevant classes

In *Cadb*, the completeness and the consistency of the object instances are systematically taken into account. An extended notion of object class, called *relevant* class, is implemented. Relevant classes represent *partial* and *meaningful* designs, characterized as potential steps towards a complete class definition. Every instance is attached to exactly one relevant class, whatever its completeness and consistency. Relevant classes cannot be related by the subclass/superclass relationship nor the generalization/specialization relationship [18]. Being partial definitions, they are *not* subclasses with respect to the object-oriented paradigm.
Relevant classes are characterized automatically by selecting from the powerset of the instance variables and constraints in a class definition, those corresponding to meaningful combinations, with respect to semantic rules [8]. The semantic rules can depend on the data model only and can be augmented by application dependent rules. A formal definition is given in [10].
The rules depending on the data model state for example that the definition of a relevant class must include all decidable constraints i.e those constraints whose arguments are all instantiated. This provides a means to take the consistency of the objects systematically into account. For example, if the "mtow" of an aircraft is the sum of its "maxload", plus its "fuel" weight, every relevant class, hence each partially instantiated instance that includes those two properties *must* also include its "mtow". The following partial definition is therefore irrelevant because it does not include the "mtow" (Figure 4) :

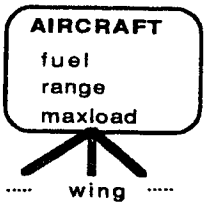

Figure 4. An irrelevant aircraft class.

The application semantics provides the opportunity to reduce further the number of relevant classes. This is defined in *Cadb* using application dependent rules. For example, the design of a new aircraft may involve two projects including the fuselage, the wings and the landing gear for the first one, and leaving the design of the engines to another project - which is actually the way it works. Each project may use a specific part of the class lattice, each of which is represented by a set of relevant classes, for example AIRCRAFT1 and AIRCRAFT2 in Figure 5. Note that both

classes include the mtow, fuel, range and maxload instance variables from AIRCRAFT, because all are required to design engines, wings, etc.

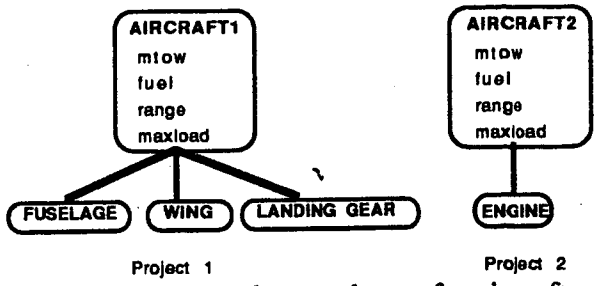

Figure 5. Two relevant classes for aircraft.

Incomplete objects, thus incrementally specified instances, are dynamically attached to the relevant classes. Because they are partial although meaningful definitions of the final design, such instances can always be attached to exactly one relevant class.

4.3 Schema change operations

The modifications allowed on the schema are modelled by finite sequences of four operations. Namely *reduction*, *augmentation*, *connection* and *product*.
The reduction and augmentation are used to drop and add instance variables, constraints and methods to classes [10].
Reduction and augmentation have two arguments. The first one is a class name and the second is a list of instance variables, methods and constraints to be dropped or added to the class definition. The operators are symmetrical : the result of a sequence of both reduction and augmentation operators using the same arguments is to leave the class definitions unchanged. The connection and product are used to combine class definitions together, in order to define composite objects. The connection applied on two argument classes produces a class definition which is an aggregate of the two arguments. Note that it can be implemented by iteratively applying the augmentation operator. The connection operator does *not* reflect any semantic dependency between its arguments. It is a mere structural aggregation. This departs from the *dependency* relationship between an object and its components. The product has three arguments : two classes and a list of instance variables, methods and constraints used to match the argument definitions. It can be implemented using a sequence of augmentation and reduction operations. The result is a class definition which is an aggregate of the two arguments, without repeating the matching elements in the list.
For example the class LANDING GEAR can be defined by connecting the classes MAIN GEAR and NOSE GEAR. It is depicted by double lines in Figure 6. An instance of the class LANDING GEAR is subsequently an aggregate of instances of NOSE GEAR and MAIN GEAR.

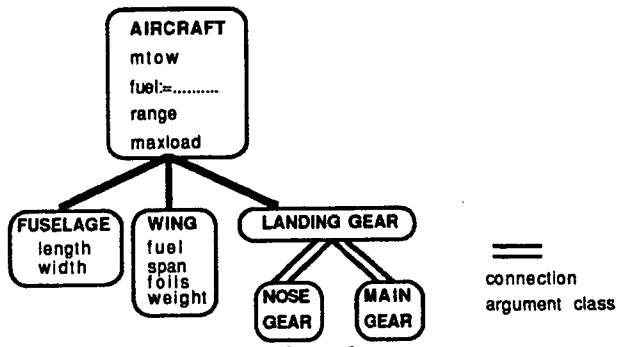

Figure 6. Connecting classes.

The reduction, augmentation, connection and product operators are designed as low level operators which provide a limited but powerful set of schema changes. They can be used to define specializations and generalizations of existing objects (e.g EXTENDED range aircraft), sub-sets of objects depending on their use by other objects, and entirely new objects (e.g LANDING GEAR). They can therefore be used to implement higher level operators, e.g specialization, generalization and aggregation [10]. As such they are expected to support most schema change operations on inheritance lattices.

4.4 Dynamic classification

An effective classification mechanism is described in this section to propagate the changes performed on the schema. It uses the notion of relevant classes and elaborates on classification mechanisms proposed in artificial intelligence for knowledge representation.
Once schema modifications are performed, their impact on the relevant classes are characterized. For each modified class, the changes are defined and tested on its relevant classes. Should all these changes be correct with respect to the model and the semantic rules, the corresponding sets of instances are modified. This means that the modifications can be propagated on the whole sets of instances belonging to the relevant classes as atomic operations. Because every instance belongs to exactly one relevant class, all instances involved are processed. They need not be scanned and updated individually. Rather, the reduction or augmentation corresponding to each relevant class is systematically applied on all its instances. Since consistency and side-effects have already been checked on the relevant classes' definitions, there is no need to further control the changes on each particular instance. Because relevant classes are defined from a finite set of instance variables and methods, and because no cyclic definition is allowed, the propagation process always terminates.
Recursive application of this heuristic may result in changing the class membership for the modified instances. For simplicity, the following is an example which deals only with the modification of domain constraints. Assume that the range constraints on the various AIRCRAFT sub-classes are as follows (Figure 7) :

- SHORT range class : range ≤ 1,000 nautical miles,
- MEDIUM " : 1,000 < range ≤ 2,000 nm,
- LONG " : range > 2,000 nm,
- EXTENDED " : range > 3,000 nm.

Changing the constraint in the EXTENDED range class from 3,000 to 2,500 nm implies that all the instances of LONG range aircraft having a range between 2,500 and 3,000 nm are now also instances of the EXTENDED range class. This implies moving those instances *downward* in the class lattice. It can be derived automatically by checking the range constraints corresponding to the various classes. Once characterized, those LONG range instances having a range between 2,500 and 3,000 nm can be propagated as a whole set to the EXTENDED range class.

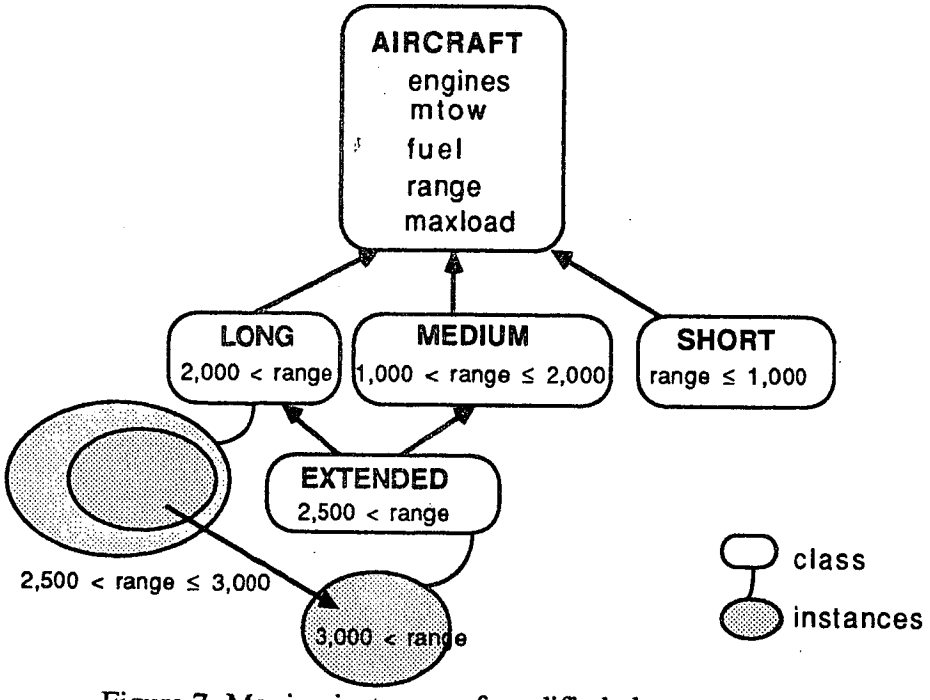

Figure 7. Moving instances of modified classes.

Subsequently, if the range constraint for the class LONG range is changed from 2,000 to 2,200 nm, all the instances of LONG range aircraft having a range between 2,000 and 2,200 nm have to be moved *upward* in the lattice to the AIRCRAFT class.
Next if the range constraint on the MEDIUM range class is relaxed to : 1,000 < range ≤ 2,200, the previous instances are then moved downward from the AIRCRAFT to the MEDIUM class. These two changes imply a *lateral* move of the instances from the LONG to the MEDIUM range class (Figure 8).

Clearly, propagating the instances in the appropriate classes can yield a large overhead. It requires checking recursively the constraints in all super-classes and sub-classes of the modified class. As noted elsewhere, the balance is between information preservation and performance [11]. The opportunity to deffer the modifications on the instances leaves this responsibility to the user.

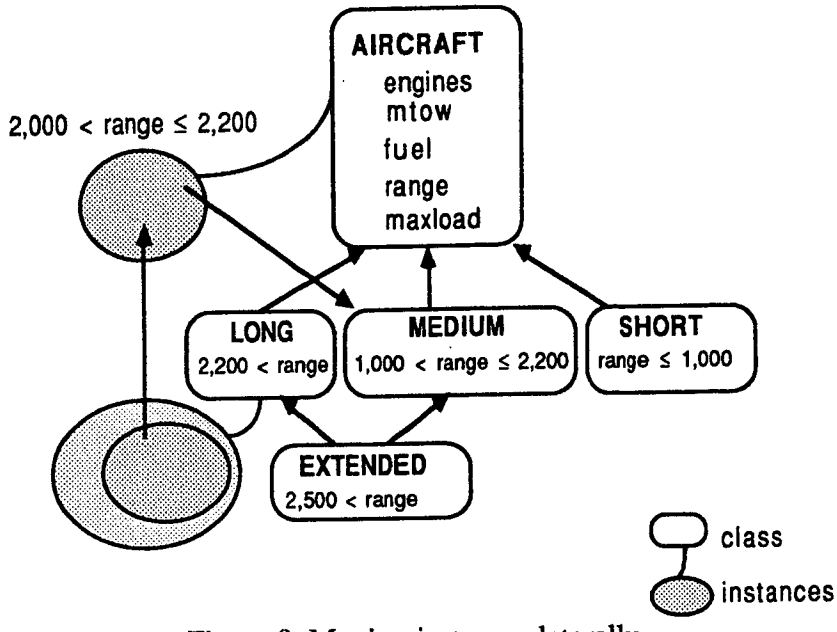

Figure 8. Moving instances laterally.

Augmenting and reducing the class definitions is propagated on the corresponding relevant classes. Existing instances can therefore be modified dynamically, as a result of changing the class definitions in the schema. Accordingly, they can be moved among the corresponding relevant classes. The process is as follows. First, the schema modifications are characterized in terms of relevant classes. New or modified class definitions exhibit specific relevant classes. These relevant classes are compared with existing ones to check if additional ones have been produced. The unchanged ones need no further processing. Pairwise differences between the new and the existing relevant classes are characterized in terms of new and deleted instances variables or methods, and modified constraints. The differences are systematically applied on all the instances belonging to the old relevant classes. The instances are then propagated to the new relevant classes. This is applied until all the relevant classes for all the classes involved in the schema change have been processed. Since relevant classes are attached to classes and because they are in finite number, the propagation process always terminates.

Current extensions directed toward the full support of dynamic information are underway in the *Sherpa* project. It is a joint INRIA and IMAG project which started in 1988. The goals in *Sherpa* are to design and implement a prototype knowledge base management system supporting :

- dynamic instances with automatic propagation of changes on semantically related objects, e.g dependent objects,
- dynamic schemas with automatic propagation of changes on the class lattice and on the instances involved,
- dynamic inference i.e., support for non-monotonic reasoning.

The first two points borrow from previous work on *Cadb*. Provision for incompleteness of the instances is a major step towards an effective system intended to support engineering design applications. The last point calls upon artificial intelligence techniques. In particular, non-monotonic reasoning is supported by an assumption-based truth maintenance system [2].

5. CONCLUSION

An analysis of various schema change operations is given, as well as an overview of other relevant issues. Prototypes systems are then compared with respect to these issues. Some systems provide extensive functionalities, including the management and versioning of composite objects, while other sometimes just ignore some of the capabilities described here. Further, results indicate that if most systems provide some form of schema evolution, very few support adequate propagation mechanisms to make the changes effective on the object instances.
A proposal is made that enables a prototype object-oriented knowledge base system called *Sherpa* to fully support schema change operations and control their impact on the object instances. It is based on a dynamic inheritance and classification scheme that propagates any schema change operation on all the classes involved and on the corresponding instances. The method provides for both immediate or deffered update. It also fully supports top-down as well as bottom-up design of composite objects. As such, *Sherpa* should support a wide variety of engineering applications.
The approach implemented is based on an extended notion of object class, called relevant classes, which takes systematically into account the partial completeness of the objects.
Propagation is performed by characterizing the modifications in terms of the relevant classes only, and grouping the changes to simultaneously update the instances belonging to the same relevant classes. It therefore avoids characterizing and propagating the changes by enumerating all instances individually.
This approach is an extension of techniques used in artificial intelligence for knowledge representation. It extends classification mechanisms with a dynamic capability which adequately supports evolving class definitions. It is expected that this approach will provide an effective methodology for managing dynamically evolving schemas in object-oriented database and knowledge base systems.

Acknowledgements.

The authors are grateful to their colleagues from the *Sherpa* project : L. Buisson, J. Escamilla, J. Euzenat, V. Favier, P. Fontanille, P. Jean, F. Rechenmann, A. Sales-Simonet and P. Uvietta for many invaluable discussions.

REFERENCES

[1] BANERJEE J., KIM W., KIM H.J, KORTH H. *Semantics and implementation of schema evolution in object-oriented databases.*
Proc. ACM SIGMOD Conference. San Francisco (Ca). May 1987.

[2] DE KLEER J. *An assumption-based truth maintenance system.*
Artificial Intelligence. 28-II. 1986.

[3] KIM W., BANERJEE J., CHOU H.T. *Composite object support in an object-oriented database system.* Proc. OOPSLA '87 Conference. Orlando (Florida). October 1987.

[4] LECLUSE C., RICHARD P., VELEZ F. *O2, an object-oriented data model.*
Proc. ACM SIGMOD Conference. Chicago. May 1988.

[5] MAIER D., STEIN J., OTIS A., PURDY A. *Development of an object-oriented DBMS.*
Proc. OOPSLA '86 Conference. Portland (Oregon). September 1986.

[6] MOLL N. *Aqua Van : Cessna Caravan amphibian.*
Flying. Vol. 113, no. 3. Diamandis Communications Inc. March 1988.

[7] NGUYEN G.T. *Semantic data engineering for generalized databases.* Proc. 2nd International Conference on Data Engineering. Los Angeles (Ca). February 1986.

[8] NGUYEN G.T, RIEU D. *Expert database support for consistent dynamic objects.*
Proc.13th International Conference on Very Large Data Bases. Brighton (England). September 1987.

[9] NGUYEN G.T, RIEU D. *Expert database concepts for engineering design.*
Artificial Intelligence for Engineering Design, Analysis and Manufacturing. 1(2). Academic Press. 1987.

[10] NGUYEN G.T, RIEU D. *Heuristic control on dynamic database objects.*
Proc. IFIP Conference "The Role of Artificial Intelligence in Databases and Information Systems". Guangzhou (P.R of China). July 1988.

[11] PENNEY D.J, STEIN J. *Class modification in the GemStone object-oriented DBMS.*
Proc. OOPSLA '87 Conference. Orlando (Florida). October 1987.

[12] RIEU D., NGUYEN G.T. *Semantics of CAD objects for generalized databases.*
Proc. 23rd Design Automation Conference. Las Vegas (Nevada). June 1986.

[13] SKARRA A.H, ZDONIK S.B. *The management of changing types in an object-oriented database.* Proc. OOPSLA '86 Conference. Portland (Oregon). September 1986.

[14] STEFIK M., BOBROW D.G. *Object-oriented programming : themes and variations.*
The AI magazine. January 1986.

[15] GOLDBERG A., ROBSON D. *Smalltalk 80 : the language and its implementation.*
Addison-Wesley. 1983.

[16] NEBEL B. *How well does a vanilla loop fit into a frame?*
Data & Knowledge Engineering 1. North-Holland. 1985.

[17] KIM W., CHOU H.T. *Versions of schema for object-oriented databases.*
Proc. 14th International Conference on Very Large Data Bases. Los Angeles (Ca). August 1988.

[18] SCHREFL M., NEUHOLD E.J. *Object class definition by generalization using upward inheritance.* Proc. 4th International Conference on Data Engineering. Los Angeles (Ca). February 1988.

[19] BANCILHON F. *Object-oriented database systems.*
Proc. 7th ACM Symposium on Principles of Database Systems. Austin (Texas). March 1988.

[20] MOON D.A. *Object-oriented programming with Flavors.*
Proc. OOPSLA '86 Conference. Portland (Oregon). September 1986.

[21] KIM H.J. *Issues in object-oriented database schemas.*
PhD Dissertation. The University of Texas at Austin. May 1988.

[22] NGUYEN G.T, RIEU D. *Schema evolution in object-oriented database systems.*
Data & Knowledge Engineering. Elsevier Science Publishers B.V. North-Holland. 1989.

[23] KIM W., BALLOU N., CHOU H.T, GARZA J.F, WOELK D., BANERJEE J.
Integrating an object-oriented programming system with a database system.
Proc. OOPSLA '88 Conference. San Diego (Ca). September 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique