UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

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

Original

Rapports de Recherche

N° 947

Programme 4

SCHEMA EVOLUTION IN OBJECT-ORIENTED DATABASE SYSTEMS

Gia-toan NGUYEN
Dominique RIEU

GROUPE DE RECHERCHE
GRENOBLE

Décembre 1988

SCHEMA EVOLUTION IN OBJECT-ORIENTED DATABASE SYSTEMS

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

[1] INRIA & [2] IMAG
Laboratoire de Génie Informatique
BP 53
38041 GRENOBLE Cedex
France
nguyen@imag.imag.fr

Abstract

Object-oriented database systems usually exhibit specific advantages over traditional database management systems and programming languages. Among them stand the ease of writing, maintaining and debugging application programs, code modularity, inheritance, persistency and sharability. Of particular interest to software engineering and computer-aided design applications is also the ability to dynamically change the object definitions and the opportunity to define incrementally composite objects. 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 which 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 and instances.
Key-words : object-oriented models, databases, dynamic schemas, inheritance, propagation, classification.

EVOLUTION DE SCHEMAS DANS LES BASES D'OBJETS

Résumé

Les bases de données à "objets" présentent certains avantages par rapport aux bases de données traditionnelles, par exemple une certaine facilité d'écriture, de mise à jour et de débogage des programmes d'application, la modularité et l'héritage de propriétés. La possibilité de modifier dynamiquement les définitions d'objets et de spécifier de façon incrémentalle des objets composés est également un de leurs intérêts pour des applications comme la CAO et le Génie Logiciel.
On passe ici en revue quelques études concernant l'évolution de schémas dans les bases d'objets. L'accent est mis sur l'aptitude à mettre en oeuvre conjointement deux aspects complémentaires : des schémas évolutifs et la propagation de leurs modifications sur les instances d'objets.
Plusieurs projets sont étudiés : *Cadb*, *Encore*, *GemStone*, *Orion et Sherpa*. Il en ressort que la plupart offrent la possibilité de modifier les schémas de données, mais qu'ils mettent rarement en oeuvre la propagation automatique de leurs mises à jour.
On propose une solution permettant dans *Sherpa* de propager les modifications de schémas et d'implanter la classification automatique des instances. Elle est basée sur une extension de techniques utilisées en Intelligence Artificielle pour la représentation des connaissances. Elle étend les mécanismes de classification habituels par une approche dynamique qui permet de gérer simultanément des classes et des instances évolutives.
Mots-clés : modèles orientés-objet, bases de données, schémas dynamiques, héritage, propagation, classification.

1. INTRODUCTION

Object-oriented systems have received considerable attention in the past few years for several reasons. The primary focus of such systems is to provide powerful concepts for application development and programming, e.g modularity, encapsulation, inheritance, overriding, protocols and polymorphism [16, 24]. Conventional database systems have been simultaneously unable to support efficiently advanced applications, e.g CAD/CAM and software engineering, which currently deal with large amounts of evolving and shared composite objects, which are linked by intricate semantic relationships [9, 12].
Consequently, merging object-oriented programming and database concepts was a promising research effort. Several projects are currently underway for the development of object-oriented database systems, e.g *GemStone, O2* and *Orion* [1, 4, 11]. 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 [3]. Most of them support changes in the object definitions, i.e evolving database schemas [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 goal here is to contribute to the characterization of salient features for object-oriented database systems. The fundamental assumptions 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.

The last two assumptions are justified by existing design thoeries, which usually rely on trial and error cycles for the design of complex artifacts. Most of the time, new items are incrementally defined using existing components which are incorporated in the new objects. Further, previous work is often reused by incorporating changes in existing objects, in order to derive new objects that will correspond to the required specifications. This is illustrated throughout the paper by aircraft design examples.
This paper is not intended to be a comprehensive study of existing research efforts and prototypes. It gives only a limited survey of specific capabilities. The choice among various systems and proposals was arbitrary. It is not the will of the authors to provide either a complete comparison or a valuable benchmarking of their functionalities. Further, systems which were not intended to provide database functionalities are also mentioned, because they support other interesting features, e.g *Loops* [16]. Therefore, any statement hereafter should not be interpreted as a judgement or a position of the authors on the overall quality of the research referenced.
Section 2 presents an overview of various interesting functionalities for object-oriented database systems intended to support advanced applications such as engineering design.
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. Restrictions are detailed where appropriate.
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. It uses dynamic inheritance and classification. Section 5 is a conclusion.

2. SCHEMA EVOLUTION IN OBJECT-ORIENTED SYSTEMS

In the following, the reader is supposed to be familiar with the object-oriented paradigm and terminology [16, 18].
An object is defined by a *structure* and an *interface*. The structure includes a collection of properties called *instance variables*, each of which has a value domain which can be either an *atomic value*, sometimes called self-defining object (numbers, character strings, ...), or another object class. The structure corresponds to the notion of *type* in programming languages [14]. The interface defines the *behavioral semantics* of the objects. It includes a set of *methods* or procedures, which are invoked by *messages* to which the objects respond by executing the corresponding code.
Objects are grouped into *classes* which are collections of *instances* corresponding to similar structures and interfaces.

Depending on the system considered, objects may share instance variables and methods by *inheriting* them in a class *inheritance hierarchy* or *lattice*. Multiple inheritance is possible in a class lattice, as in *Encore* and *Orion* [1, 14]. Classes are related in the inheritance lattice by the *super-class/sub-class* relationship.
In the following, a database *schema* is a set of class definitions interconnected by the super-class/ sub-class relationship. It is represented by a class lattice. No restriction is assumed on references from classes to specific instances. Stated otherwise, sub-classes do not necessarily partition the instances of their super- classes. Therefore, the sets of instances in sub-classes are not necessarily disjoint. This provides for multiple inheritance and multiple references. Further, no recursive and no cross-referenced definitions are allowed.
Object and class names are written in uppercase letters, e.g AIRCRAFT, while instance variables are in lowercase, e.g "range". 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 "type" of an aircraft is an atomic value, e.g "jet" or "turboprop", the "mtow" is in tons, the "range" in nautical miles (nm) and the "fuel" capacity in US gallons. 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 "range" between 1,000 and 2,000 nm (this is not described in Figure 1). 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.

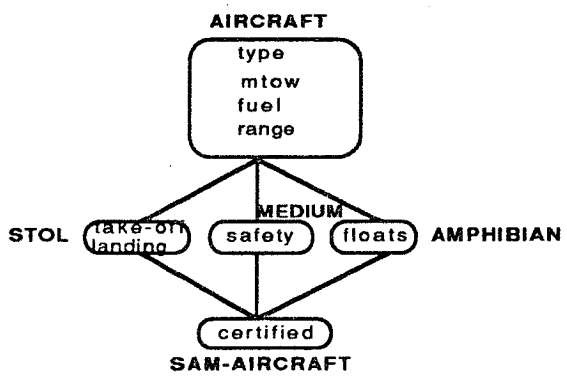

Figure 1. Class lattice for aircraft.

It is emphasized here that no classification mechanisms like those used by knowledge representation schemes in Artificial Intelligence are invoked in object-oriented systems. It results that inheritance in the usual object-oriented approach does not imply the automatic propagation of the instances corresponding to the various class definitions in the lattice. For example, the instances created in the class AIRCRAFT do not automatically belong to any sub-class, unless explicitly coerced by the user. This is discussed in Section 4 where the dynamic classification of the instances in an object-oriented database system is proposed.

2.1 Schema change operations

Although different in some aspects, the taxonomy of schema change operations and the related issues discussed in this section grossly follows that of *Orion* [1]. 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.
These operations are briefly surveyed in the remainder of this section (Sections 2.1.1 to 2.1.3). Other relevant issues are discussed in the following subsections, including composite objects and versions (Sections 2.2.1 to 2.2.3). Various prototypes are later analyzed with respect to these issues, in Section 3.

2.1.1 Changing class definitions

Changing class definitions includes :
- adding or deleting new instance variables and methods in a class definition,
- modifying existing instance variables and methods, e.g changing their name, their domain or constraints.

Uniqueness of the names is usually required between classes and among each class's instance variables. Potential name conflicts may be readily solved by rejecting the conflicting instance variable or method [11].

For example, the instance variable "maxload" can be added to the class AIRCRAFT (Figure 2). Its domain is the maximum weight of cargo allowed.

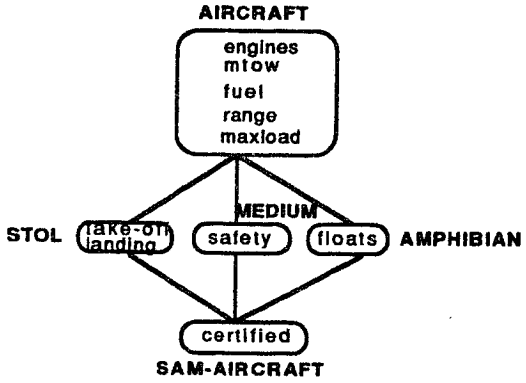

Figure 2. Adding instance variables.

2.1.2 Adding and deleting classes

Adding and deleting classes in the class lattice is a common feature in object-oriented data models. Creating new specialized classes from existing classes is also a basic constructor of object-oriented schemas.

For example, 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 3) .

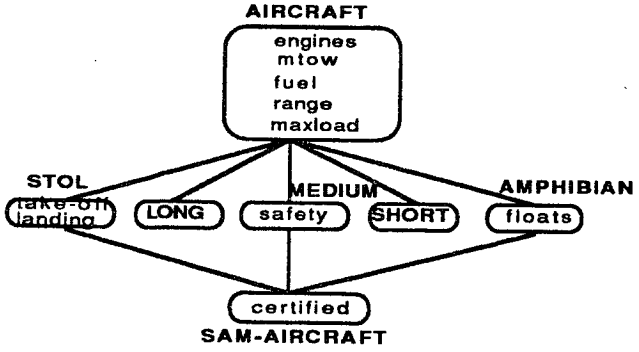

Figure 3. Adding new classes.

Specific equipment, such as increased fuel capacity, allow some aircraft like the Airbus A300 to be certified for long-range intercontinental flights. They have the so-called "extended-range" capability, e.g the Airbus A300-600R. The class EXTENDED range can be defined as a specialization of the class AIRCRAFT (Figure 4).

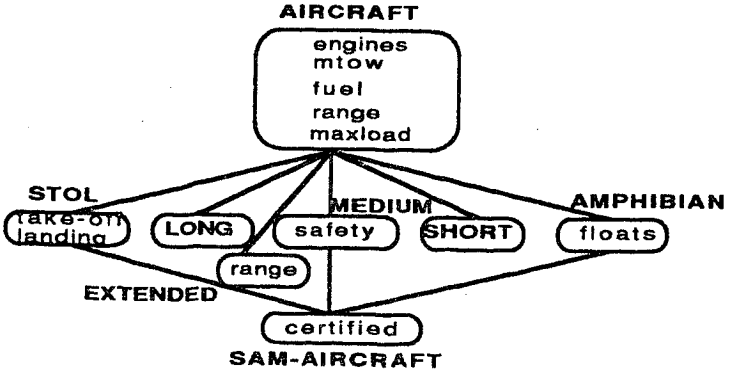

Figure 4. Specializing classes.

2.1.3 Changing class relationships

Changing the class relationships is another facility which allows taking into account schema evolution. It 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 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 (Figure 5). 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. Finally, the existing relationship between the AIRCRAFT and EXTENDED classes may be deleted because it happens that no aircraft is designed primarily as an EXTENDED range, but rather as a refinment of an existing MEDIUM range one.
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, for example by inheriting particular performance characteristics. This is described by the specialization relationship between the LONG and EXTENDED classes (Figure 5).
Name conflicts may result from changes in the class relationships. When multiple inheritance is available, ordering the super-classes of a given class may avoid to some extent these conflicts. For example, the instance variables and methods with the lowest order number in the superclass list are inherited. Overriding allows also locally defined instance variables and methods in a class to redefine inherited instance variables and methods. For example, the instance variable "range" is defined locally in the class EXTENDED range (Figure 4). Specific calculation or constraints can therefore be invoked on the range instance variable for that particular class.

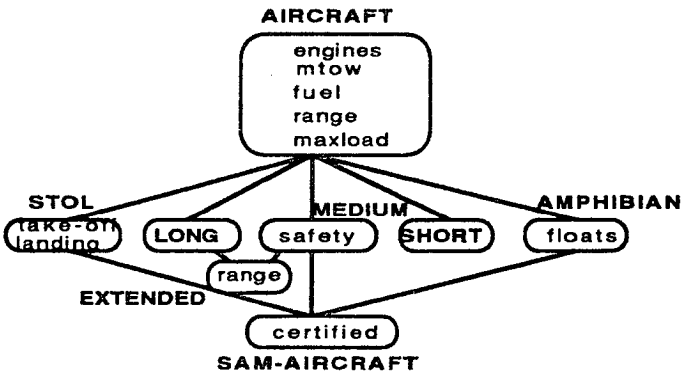

Figure 5. Changing class relationships.

Further, if instance variables and methods are transitively inherited from the superclasses of a class, constraints and domain conflicts can also 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 Relevant issues

The following sections discuss some features currently supported by advanced object-oriented database systems which interact with evolving schemas. This includes versions of objects, composite objects and versions of composite objects, which are fundamental to applications such as engineering design (Sections 2.2.1 to 2.2.3).

2.2.1 Composite objects

Incremental specification of the design artifacts requires changing dynamically the class definitions, hence the database schema. While many prototypes support only the schema change operations described in Section 2.1, more sophisticated ones provide advanced semantics concerning the management of dynamically evolving objects.
While composite objects are often made available, the notion of *dependent* objects is seldom supported. This is described in this section. From a more pragmatic point of view, versions of (composite) objects are sometimes provided. This is detailed in Sections 2.2.2 and 2.2.3.
Composite object definition and manipulation is a major issue in many applications today [3].

While traditional bussiness aplications usually rely on simple record-oriented data structures, the management of large engineering projects, such as aircraft design, require the manipulation of thousands of components which participate in the overall design. These pieces of work must be incorporated in the global object definition and their manipulation requires careful attention. While off-the-shelf components, i.e existing objects, can be used in many parts of a design, specific components need always be designed to meet particular requirements. Semantic information has therefore to be taken into account to differentiate between those particular components which are specific to the design under consideration, namely dependent objects, and the replacable units which are previously defined. It results that the notions of composite objects and dependent objects must be differenciated [12].
For example an aircraft is composed of a fuselage, wings, engines and a landing gear. They are objects on their own and designed independently, but a *dependency* relationship exists between an aircraft instance and fuselage, wing or engine instances. It reflects the fact that the existence of the component classes' instances depend on the existence of the "owner" aircraft instances. It is therefore more than a mere *structural* definition, which can be defined straightforwardly, e.g by the list "mtow", "fuel", etc. It is indeed a *semantic* relationship between the instances involved. It can be defined at a generic level, i.e in the corresponding class definitions, or at the specific instance level. This is shown in bold lines in Figure 6 and all subsequent figures.

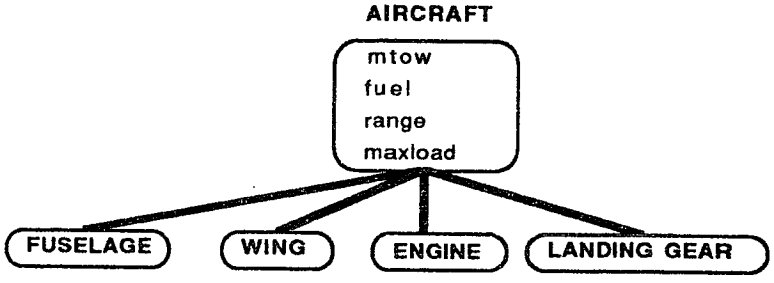

Figure 6. Composite object aircraft.

For example, each AIRCRAFT instance owns specific instances of the classes FUSELAGE, WING, etc. In particular, various aircraft belonging to the same model, say B737, can be fitted with different engines, e.g CFM56 or JT8D from various manufacturers. Each serial number aircraft owns exclusively different instances of the various engines. Should an aircraft instance be deleted, all instances referenced through a dependency relationship are deleted. This means that the deleted aircraft's engines are also deleted. If this is not the wish of the user, the dependency relationship must be relaxed.
Changes to the definition of composite objects have to be propagated on their components, e.g deleting a dependency relationship. Similarly, changes to the definition of a component must be visible to the owner object to enforce the dependency relationship.

2.2.2 Versions

Versioning of objects has been studied for a long time in the database community. Object versions are also relevant to object-oriented database design because increased computer assistance encourages trial and error cycles for application development.
Generic instances may be used to model the version derivation hierarchy for a given class (Figure 7). They may be used to reference objects without specifying in advance the particular version needed.

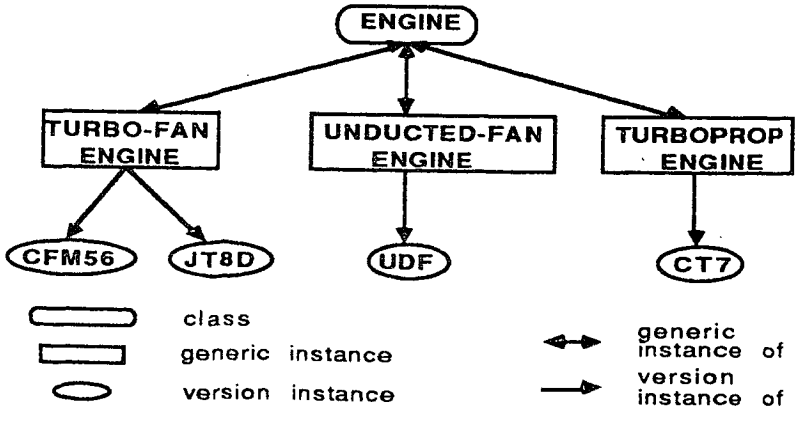

Figure 7. Derivation hierarchy for engine versions.

Specific versions in the derivation hierarchy are called *version instances*. As shown in Figure 7, CFM56 is a version instance of the generic instance "turbo-fan engine". Turbo-fan engines are fitted on all popular commercial aircraft today. Similar comments apply to the version instances UDF and CT7 for the generic instances "unducted-fan engine" - a new generation jet engine - and "turboprop engine" -the most widely used propeller engines nowadays.
Also various options in the B737 model, called 737-100, 737-200 and 737-300 provide for different seat capacity, within a range of 100 to 150 passengers, and various sophistication in cockpit equipment, including CRT instrument displays and automatic landing electronics [15]. In Figure 9, B737 is a generic instance for the class AIRCFRAFT, while B737-300 is a particular version instance for B737.
Changes to the definition of a class have to be propagated to its generic instances, unless a new generic or version instance is explicitly created. Modifications performed on a generic instance should also be propagated to its version instances, unless a new generic instance is created.
Note that the notion of generic instance is specific to the applications. One might argue for example that "turbo-fan engine" can be made a sub-class of ENGINE, since turbo-fans are particular engines. They have specific characteristics known by the engineers. This is fundamentally a schema design issue. It is not discussed here.

2.2.3 Composite object versions

When a dependency relationship exists between a parent class and a component class, for example between the AIRCRAFT class and the ENGINE class (Section 2.2), it applies to their generic instances, if any. For example the dependency between the classes AIRCRAFT and ENGINE in Figure 6 applies to their generic instances "B737" and "turbo-fan engine" (Figure 8).

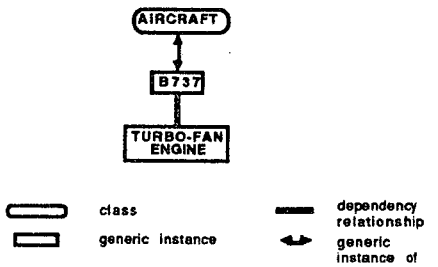

Figure 8. Composite link between generic instances.

This allows generic instances to be referenced dynamically by other objects, i.e without specifying a priori any particular version instance. Conversely, version instances are referenced statically, i.e like any other object instance. Binding a particular version instance to a component generic instance may be allowed. For example the version instance B737-200 can be bound to the generic instance "turbo-fan engine" in Figure 9. This means that the designers intend to fit turbo-fan engines on the B737-200s, without specifying more precisely any such engine by now.

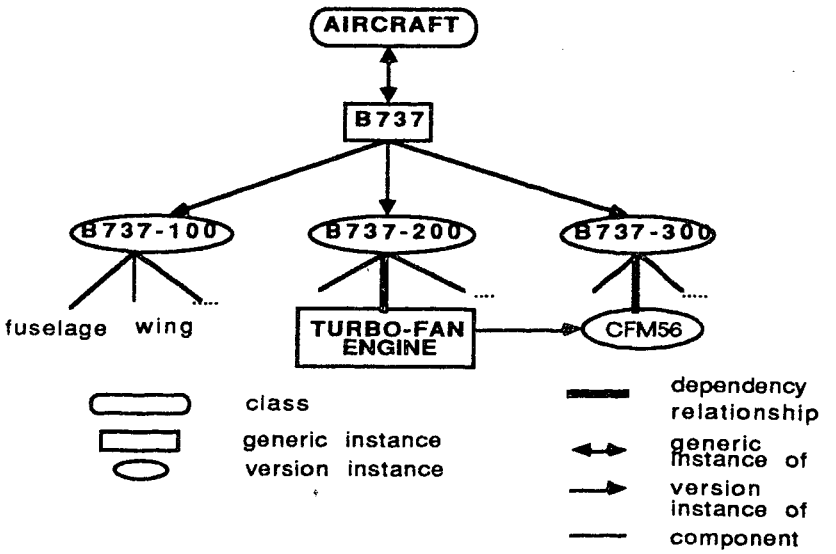

Figure 9. Composite object versions.

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 above. This includes *Cadb, Encore, GemStone, Orion and Sherpa.*
The notion of inheritance lattice is discussed first (Section 3.1). The operations allowed are analyzed next (Section 3.2). Composite objects and versioning are detailed in sections 3.3 and 3.4. Propagating the changes performed on the schemas concerns both the control of their impact on the class definitions and on the existing object instances. This is discussed in sections 3.5 and 3.6. While the first issue is often emphasized, the second is often superficially addressed. It requires indeed classification mechanisms which are usually not made available in the database approach. This will be discussed later in Section 4.

3.1 Inheritance lattice

Consistency of the schema in *GemStone* and *Orion* is specified by a set of *invariants* that are enforced by semantic rules [1, 11]. In contrast with *GemStone,* but like *Loops*, classes are organized in *Orion* in an inheritance lattice [1]. This allows multiple inheritance and complicates somewhat the name conflict problem. As mentionned above (Section 2.1.3), this can be solved by ordering the superclasses for each class (user defined or default ordering).
In the example in Figure 2, the ordering of the super-classes for the class SAM-AIRCRAFT is STOL, MEDIUM and AMPHIBIAN. Should the class EXTENDED range (Figure 4) be made a super-class of SAM-AIRCRAFT, it would be added at the end of its super-class list (i.e STOL, MEDIUM, AMPHIBIAN and EXTENDED range), unless specified otherwise by the user.
This helps avoiding to some extent name conflicts when renaming or adding new instance variables. Should any such conflict arise due to inheritance of instance variables and methods, propagation of the new or modified instance variable does not occur in the conflicting sub-classes (see Section 3.5).

3.2 Schema change operations

Most systems provide a limited set of schema change facilities. Concepts such as default or shared values and composite objects in *Orion* are usually not supported by other systems [3]. Provision for renaming of classes and instance variables, as well as name conflict resolution, are summarized in Table 1.

	Name conflict resolution	Change name of instance variable	Change name of class
Cadb	no	no	no
Encore	no	no	no
Orion	yes	yes+	yes
GemStone	no	yes*	no
Sherpa	yes	yes	yes

\+ Propagated to subclasses
* Except if inherited (propagated to subclasses)

Table 1. Changing names in the schema.

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 instances 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 (see Section 3.4). Similarly, adding a

new class as a leaf in a hierarchy provides in *GemStone* a means to *specialize* an existing class by adding specific instance variables, constraints and methods. The new class inherits all instance variables and methods from its *super-classes*, unless otherwise redefined.
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. This is summarized in Table 2.

	Add a class	Delete a class	Add an instance var	Delete an instance v.	Create superclass	Delete superclass	Change constraint
Cadb	yes	yes	yes	yes	yes	yes	yes
Encore	yes	yes	yes	yes	yes	yes	yes
Orion	yes	yes	yes	yes	yes	yes	yes+
GemStone	yes	yes*	yes	yes§	no	no	yes°
Sherpa	yes	yes	yes	yes	yes	yes	yes

+ generalize only (propagated to subclassses) * if class empty
° generalize or specialize (not propagated) § except if inherited (not propagated to subclasses)

Table 2. Schema change operations.

3.3 Composite objects

As mentionned previously, composite objects are basically structural aggregates of sub-parts involved in the definition of a composite object. 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* 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. This is summarized in Table 3.
One problem in the *Orion* approach is that composite objects and dependent objects are merged in the same concept, by the generic *composite link* property. As described later in Section 4.1, this constraint should sometimes be relaxed. Since it decouples the (instance-specific) dependency relationship from the (generic) composite object definition, this problem is irrelevant in *Cadb*.
Neither *GemStone*, *Encore* nor *Loops* seem to support explicitly a similar notion of dependent objects.

3.4 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 [14]. The emphasis is on the preservation of the behavior of the objects using versions of classes. An error-handling mechanism provides the correct version of 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. Error-handlers are systematically associated with the new versions in order to provide for consistent access later on. So far however, no name conflict resolution is provided. Future extensions should also provide for type merging and splitting.
Orion extends the notion of version to that of versions of schema [22]. As described previously,

generic instances are provided to support versions of instances [3]. Like *Cadb*, it supports versions of composite objects. Composite links between a parent version instance and a component generic instance are supported, e.g B737-200 and turbo-fan engine, and between a parent version instance and a component version instance, e.g B737-300 and CFM56 (Figure 9). Neither *GemStone* nor *Loops* seem to support explicitly the notion of version (Table 3).

	Versions Instances	Versions of classes	Versions of schema	Dependent objects
Cadb	yes	no	no	yes°
Encore	yes	yes*	no	no
Orion	yes	no	yes+	yes§
GemStone	no	no	no	no
Sherpa	yes	yes	no	yes

* all versions always instantiable ° instance-specific level
\+ last version instantiable only § generic (class) level

Table 3. Versioning and dependent objects.

3.5 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. Adding a new instance variable is allowed if it is not already defined in a subclass. Deleting an instance variable in a class definition is allowed if it is not inherited from a superclass. Further, the deletion is not propagated to the subclasses. This must be explicitly performed on each subclass.
Specializing and generalizing the domain of an instance variable is not allowed if it is inherited. In either case, the operation is not propagated to subclasses and has to be explicitly performed as required. This is described in Table 2.
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 LONG range aircraft in Figure 5 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]. For example, the domain of the engines in the class aircraft, e.g the class ENGINE, can be generalized to include all vehicle engines, but it cannot be later restricted to exclude diesel engines (Figure 10) - there exist indeed throughout the world a few exotic prototype aircraft powered by diesel engines.

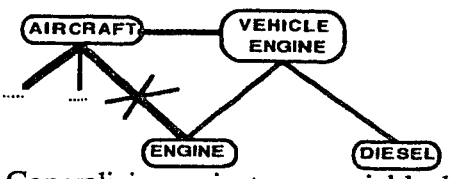

Figure 10. Generalizing an instance variable domain.

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 an error-handling mechanism which is attached to each particular version [14].

3.6 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.
Restrictions always limit propagation to specific cases with respect to the operation involved and the arguments of the change (see Section 3.5 for example).
Invariants preserving rules in *Orion* also avoid most propagation problems on object instances. Screening deleted instance variables and adding default or nil values to a instance variables is performed when fetching the modified class' objects.
The notions of composite objects and versions complicate somewhat the propagation of changes performed on the schemas. For example in *Orion*, adding to an existing class a super-class which contains a composite link is propagated to the former's sub-classes. They will inherit the new composite link. For example, replacing the class AIRCRAFT in Figure 5 by the class described in Figure 6 (where a composite link exists with the class ENGINE) is possible. The modification will be visible to the classes LONG, MEDIUM, SHORT and EXTENDED range. However, this implies no automatic inheritance of engine instances.
In contrast with *Encore*, when new version instances of composite objects are derived, no new versions of components are created. Composite links to versions instances are set to nil. Composite links to generic instances, which are dynamically bound, are not modified. For example, deriving a new B737-400 version from the B737-300 model implies that the ENGINE component is set from CFM56 to nil (Figure 11).

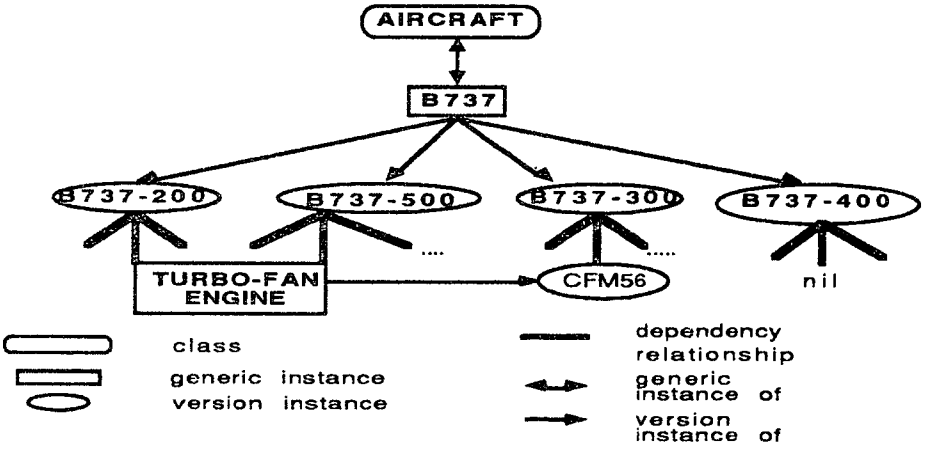

Figure 11. Deriving new versions.

In contrast, if a new version B737-500 is derived from the B737-200 model, the composite link to the turbo-fan engine component (a generic instance) will remain unchanged (Figure 11). The 400 model is indeed presently tested for certification, while the 500 model is already planned by the manufacturer [15].
Because components may usually have only one owner object, it is not possible in *Orion* to change a non-composite link to composite. This avoids maintaining reference counts in the components, which neither have to include any reference to their owner. This is summarized in Table 4.

	Propagation of changes	Immediate propagation	Deffered propagation
Cadb	automatic	yes	no
Encore	manual	no	yes°
Orion	automatic	no	yes+
GemStone	automatic	yes*	no
Sherpa	mixed	yes	optional

* conversion
° convert operator
+ screening

Table 4. Propagation of changes.

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. 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. It therefore provides a new contribution to the domain of data and knowledge bases.
The requirements for supporting the incremental design of artifacts are briefly discussed in Section 4.1. Next, an extended notion of object class is informally defined, which takes into account the completeness of evolving and dependent objects, namely *relevant* classes (Section 4.2 and 4.3). A basic set of schema change operations is then described through examples (Section 4.4). Combined with relevant classes, it allows *grouping* the modifications on the object instances, as a result of changing the schema (Section 4.5 and 4.6).

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 which values depend on other instance variables from one or more objects. This is described in the example in Figure 17, where the total fuel capacity of an aircraft is derived from the fuselage and wings capacities.
Cadb is intended to minimize the restrictions usually burdening propagation of changes in other CAD 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 in the example in Figure 6 if the corresponding aircraft does not exist. It is neither possible to create a CFM56 instance of turbo-fan engine, without knowing to which instance of B737-300 it belongs in Figure 9. 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 request. 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.
One solution is to create a new class for the engines being replaced, say SPARE ENGINE (Figure 12). Note that those engines are in real life reconditioned, then used as spare parts and eventually refitted on other aircraft. This means that they have no owner aircraft for an undefined, but possibly limited, period of time. The old engines have therefore to be deleted in the ENGINE class, and new copies must be created in SPARE ENGINE.

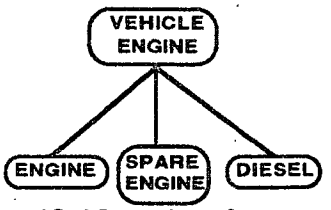

Figure 12. New class for spare engines.

Incremental design can therefore rely in part on the notion of composite object. As shown in the example above, dependency relationships must however be carefully defined to avoid inconsistencies and potential information losses. Crude manipulations of the schema can alleviate to some extent such problems, e.g the creation of the class SPARE ENGINE. It is clear however that controlling incompleteness and inconsistency of the design objects require new concepts and mechanisms . This is discussed in the next Section.

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 [23]. Being partial definitions, they are *not* subclasses with respect to the object-oriented paradigm.They are only used by the system to support the incremental definition of the artifacts. Further, they are not meant to be incorporated in the user-defined inheritance lattice.
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 which 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 13) :

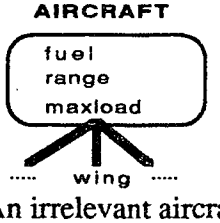

Figure 13. An irrelevant aircraft class.

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 14. Note that both classes include the mtow, fuel, range and maxload instance variables from AIRCRAFT, because all are required to design engines, wings, etc. And again, this is how it really works.

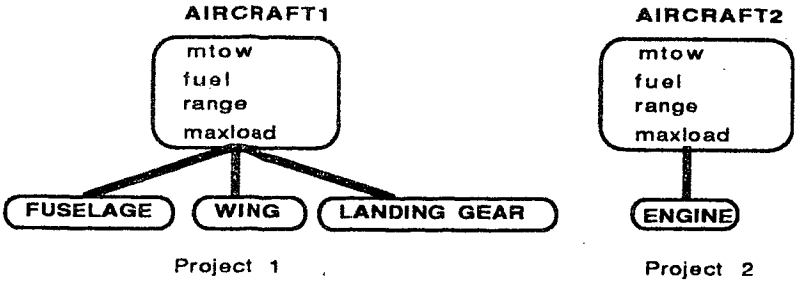

Figure 14. Two relevant classes for aircraft.

Similarly, European cooperation makes it possible to design and manufacture parts of an Airbus by dozens of companies throughout Europe, for example wings in Great-Britain and Spain, fore fuselage in France, rear fuselage in West-Germany and so on. Each company involved must use a set of relevant classes from the whole AIRCRAFT class for its own purpose, for example AIRCRAFT3 and AIRCRAFT4 in Figure 15. Note that the range of an aircraft is irrelevant for the design of the fuselage and wings and is omitted in both definitions.

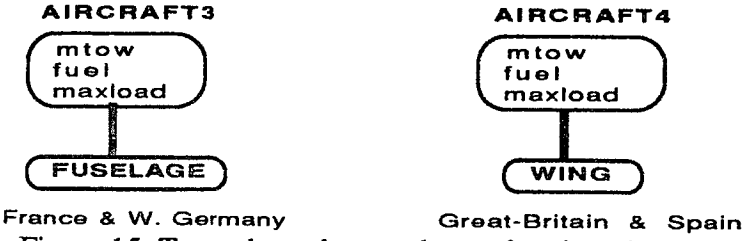

Figure 15. Two other relevant classes for aircraft.

Relevant classes are characterized automatically by selecting all meaningful combinations of instance variables and methods in the AIRCRAFT class. Their purpose is only to evaluate the side-effects and propagating the changes made on the schema. They can be made transparent to the user. The following AIRCRAFT5 and AIRCRAFT6 classes are also relevant (Figure 16).

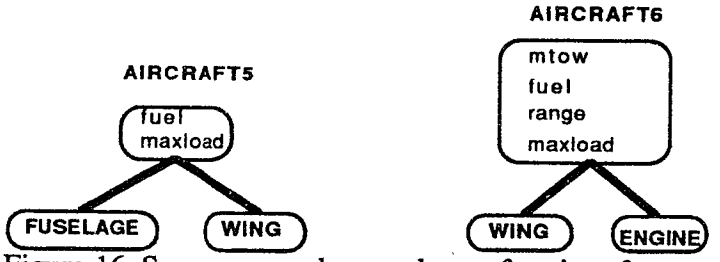

Figure 16. Some more relevant classes for aircraft.

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 Dependent objects

Dependent objects are created when top-down design is used to define components which existence depends on the existence of a parent object. This is similar to the notion of composite objects and composite link in *Orion*. The authors are not aware of other systems implementing such concepts. Contexts are used in *Cadb* to propagate the changes on the dependent instances. Modifications are allowed only in the context where an instance is created. Further, derivation rules specifying the calculation of instance variables which values depend on other instance variables (in the same or other objects) automatically take into account the changes performed on the objects. For example, aircraft usually carry fuel tanks in the fuselage and the wings. This requires the redefinition of the fuel capacity in the example figures above. First a fuel instance variable is added in the classes FUSELAGE and WING. Next the aircraft fuel capacity is redefined as the sum of the fuel capacity in the fuselage, plus two times the fuel capacity in the wings (Figure 17). Modifying any of the fuel values in the classes FUSELAGE or WING is automatically propagated, should an AIRCRAFT's fuel be fetched later.

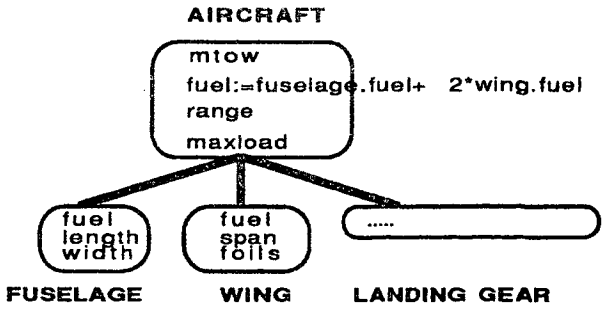

Figure 17. Computation of the aircraft fuel capacity.

4.4 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. A formal definition is given in [10].

For example, augmenting the class WING with an instance variable "weight" allows further information to be taken into account. Similarly, reducing the class FUSELAGE to avoid specifying the fuel capacity of internal tanks is possible by reducing its definition, omitting the "fuel" instance variable (Figure18).
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 definitions. 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.

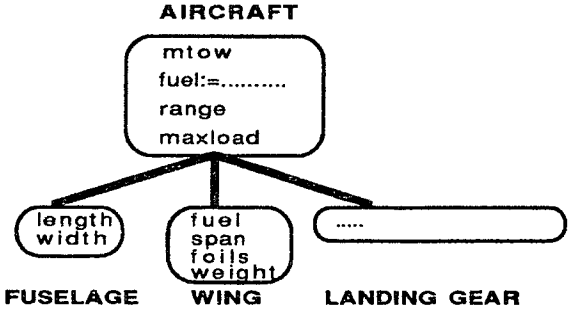

Figure 18. Augmenting and reducing classes.

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, as defined in Section 2.2.1. 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 19. An instance of the class LANDING GEAR is subsequently an aggregate of instances of NOSE GEAR and MAIN GEAR.

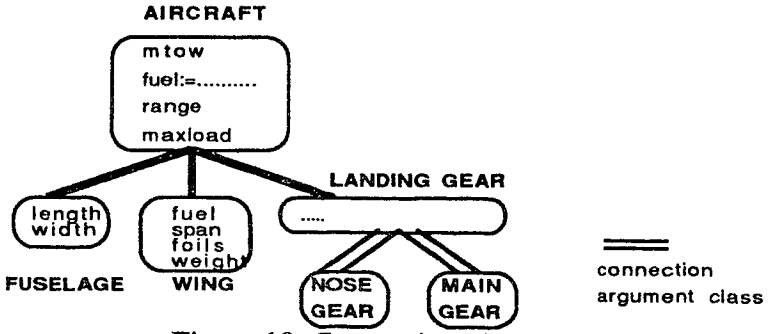

Figure 19. Connecting classes.

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 (e.g B737-CFM56 engines below), 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]. For example, it is straightforward to define a new class for those CFM56 engines fitted on B737s, say "B737-CFM56", and make it a sub-class of ENGINE using the augmentation operator. Applying the augmentation operator allows for example to add servicing information : last service date, mecanics, hours since last serviced, parts serviced, parts replaced, etc (Figure 20).

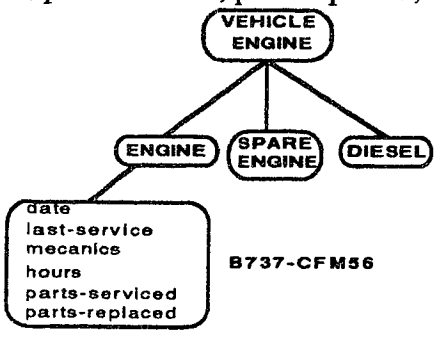

Figure 20. The class of CFM56 engines on B737s.

4.5 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 modifications can be propagated on whole sets of instances belonging to 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 formula 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. Since 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 example, assume that the range constraints on the various AIRCRAFT sub-classes are as follows (Figure 21) :

- SHORT range class : range $\leq$ 1,000 nautical miles (nm),
- MEDIUM " : 1,000 < range $\leq$ 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 instances of LONG range aircraft having a range between 2,500 and 3,000 nm are now also instances of the EXTENDED range class (Figure 21). 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.
Subsequently, if the range constraint for the class LONG range is changed from 2,000 to 2,200 nm, all 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 $\leq$ 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 22).

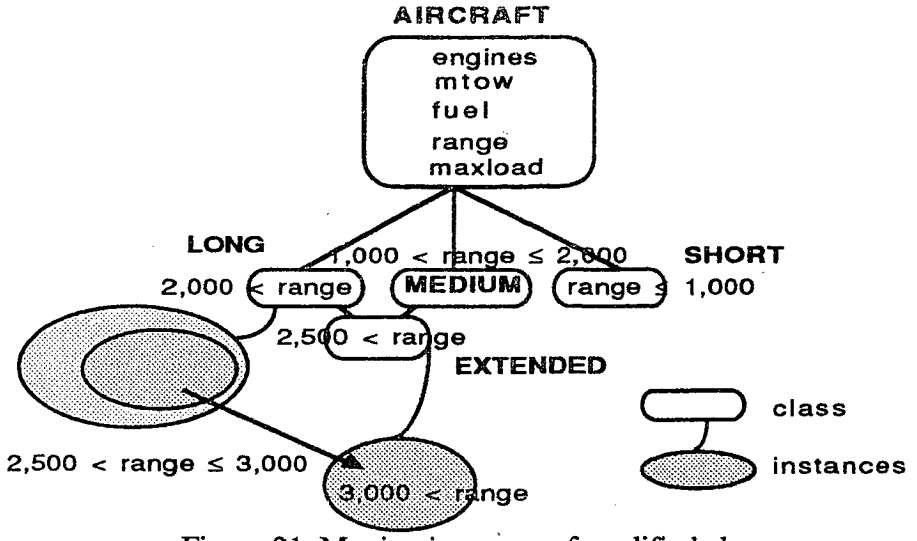

Figure 21. Moving instances of modified classes.

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.

4.6 Grouping changes

The example above deals only with the modification of domain constraints. Dynamic classification applies also to the incremental design of objects. In particular, going back and forth through various design alternatives subsumes the ability to dynamically change the object definitions. It follows that controlling the completeness and the consistency of the objects is a crucial issue.
For example, augmenting and reducing class definitions is propagated on the corresponding relevant classes. Existing instances can therefore be modified dynamically, as a result of changing 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. 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.

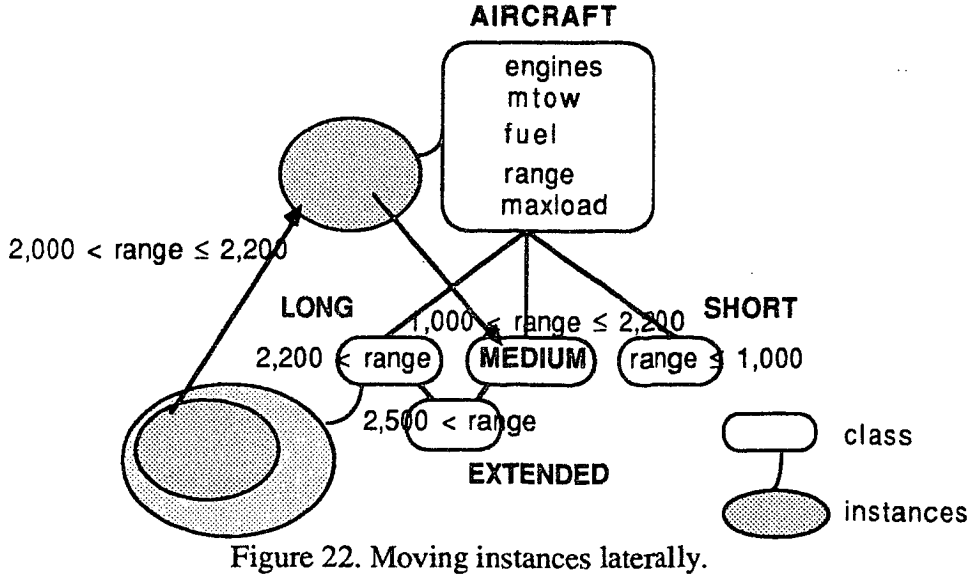

Figure 22. Moving instances laterally.

For example reducing the AIRCRAFT4 definition in Figure 15 with the mtow and range instance variables produces a move of all the corresponding instances to the relevant class AIRCRAFT7 (Figure 23). Also reducing the relevant class AIRCRAFT6 with the ENGINE component (Figure 18) implies moving all its instances to the class AIRCRAFT4. Further, augmenting AIRCRAFT7 with the mtow and the range instance variables will move back the corresponding instances to the reduced class AIRCRAFT6.

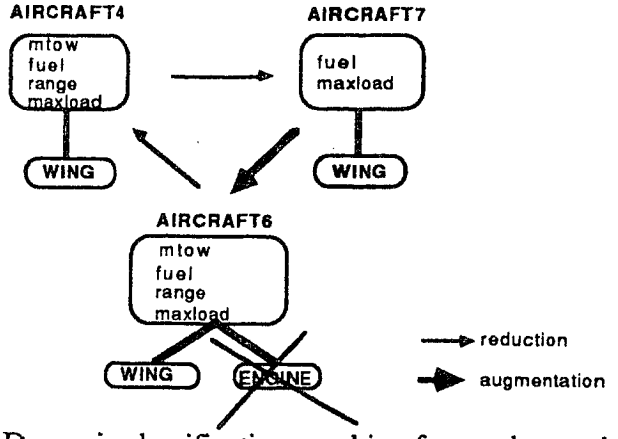

Figure 23. Dynamic classification resulting from schema changes.

Since modifications are characterized in terms of the relevant classes, there is no need to search for particular changes on the instances individually. This allows *grouping* the changes on all the instances belonging to the relevant classes, thus improving the effectiveness of the approach. This issue was previously recognized as a major requirement for class modification in *GemStone* [11].
As described in Figure 24 given in Appendix, the relevant classes form a connected graph which edges are labelled by the schema modifications. The AIRCRAFT class is the result of connecting the classes AIRCRAFT1 and AIRCRAFT2 together (Section 4.2). The relevant class AIRCRAFT3 to AIRCRAFT7 can be obtained by reducing, augmenting, or connecting together the others. Figure 24 does not detail all the relevant classes.
A prototype *Cadb* implementing these features has been implemented in Prolog on Apollo workstations [12]. It includes approximately 25,000 lines of code. Current extensions directed toward the full support of dynamic information are underway in the *Sherpa* project. It is a joint INRIA-IMAG project.
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 will borrow from previous work on *Cadb*. Provision for incompleteness of the instances will be 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 will be supported by an assumption-based truth maintenance system [2].

4.7 Research issues

Large efforts remain to be done in order to integrate schema evolution and the propagation of changes in some acceptable prototype. Where previous proposals emphasize the persistency and sharability of data, our primary focus is to support dynamically evolving schemas and instances. However sharing and concurrency interfere dramatically with the management of dynamic schemas and the propagation of changes [3, 11].
One approach is to rely on versions of schemas, in much the same way that engineering applications call for versions of objects [22]. Various levels of granularity and versatile locking schemes have also to be designed to take into account changes in the schemas when concurrently accessing the objects [20].
Further, introducing non-monotonic reasoning and using an assumption-based truth maintenance system require the support for dynamically evolving *contexts* [21]. They characterize the set of instances that allow values to be derived from each other. This clearly imposes constraints on object sharing and class evolution. These issues are being explored at present in *Sherpa*.

5. CONCLUSION

When comparing database evolution with recent efforts dedicated to object-oriented systems, it is clear that important steps have been made towards modelling and manipulating dynamic and composite objects, as well as managing and maintaining semantically related information. Even though existing prototypes propose significantly different functionalities, it is now widely recognized that schema evolution is an important requirement for many applications.
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 Pr. Van de Riet for his comments and suggestions, which significantly helped to improve an earlier version of this paper.

This work is supported in part by the French Department of Research, Program PRC-BD3.

REFERENCES

[1] BANERJEE J., KIM W., KIM K.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. & al.
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.
Altaïr Technical Report 10-87. September 1987.

[5] MAIER D. & al.
Development of an object-oriented DBMS.
Proc. OOPSLA '86 Conf. 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 (AI EDAM). (1)2. C.L Dym ed. Academic Press. 1987.

[10] NGUYEN G.T, RIEU D.
Heuristic control on dynamic database objects.
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] RIEU D., NGUYEN G.T
Dynamic schemas for engineering databases.
Proc. 4th International Conference on Systems Research, Informatics and Cybernetics. Baden-Baden (FRG). August 1988.

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

[15] SLOCUM J.
Baby boomer : Boeing 737.
Flying. Vol. 113, no. 3. Diamandis Communications Inc. March 1988.

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

[17] ZDONIK S.B
Object management systems for design environments.
IEEE Database Engineering. Vol. 8, No. 4. December 1985.

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

[19] NEBEL B.
How well does a vanilla loop fit into a frame ?
Data & Knowledge Engineering 1. North-Holland. 1985.
[20] DITTRICH K.R
Object oriented database systems : the notions and the issues.
Proc. Workshop on Object Oriented Database Systems.Pacific Grove (Ca). September 1986.
[21] FILMAN R.E.
Reasoning with worlds and truth maintenance in a knowledge-based programming environment. Comm. ACM. 31, 4. April 1988.
[22] KIM W. & CHOU H.
Versions of schema for object-oriented databases.
Proc. 14th International Conference on Very Large Data Bases. Los Angeles (Ca). August 1988.
[23] 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.
[24] BANCILHON F.
Object-oriented database systems.
Proc. 7th ACM Symp. on Principles of Database Systems. Austin (Texas). March 1988.

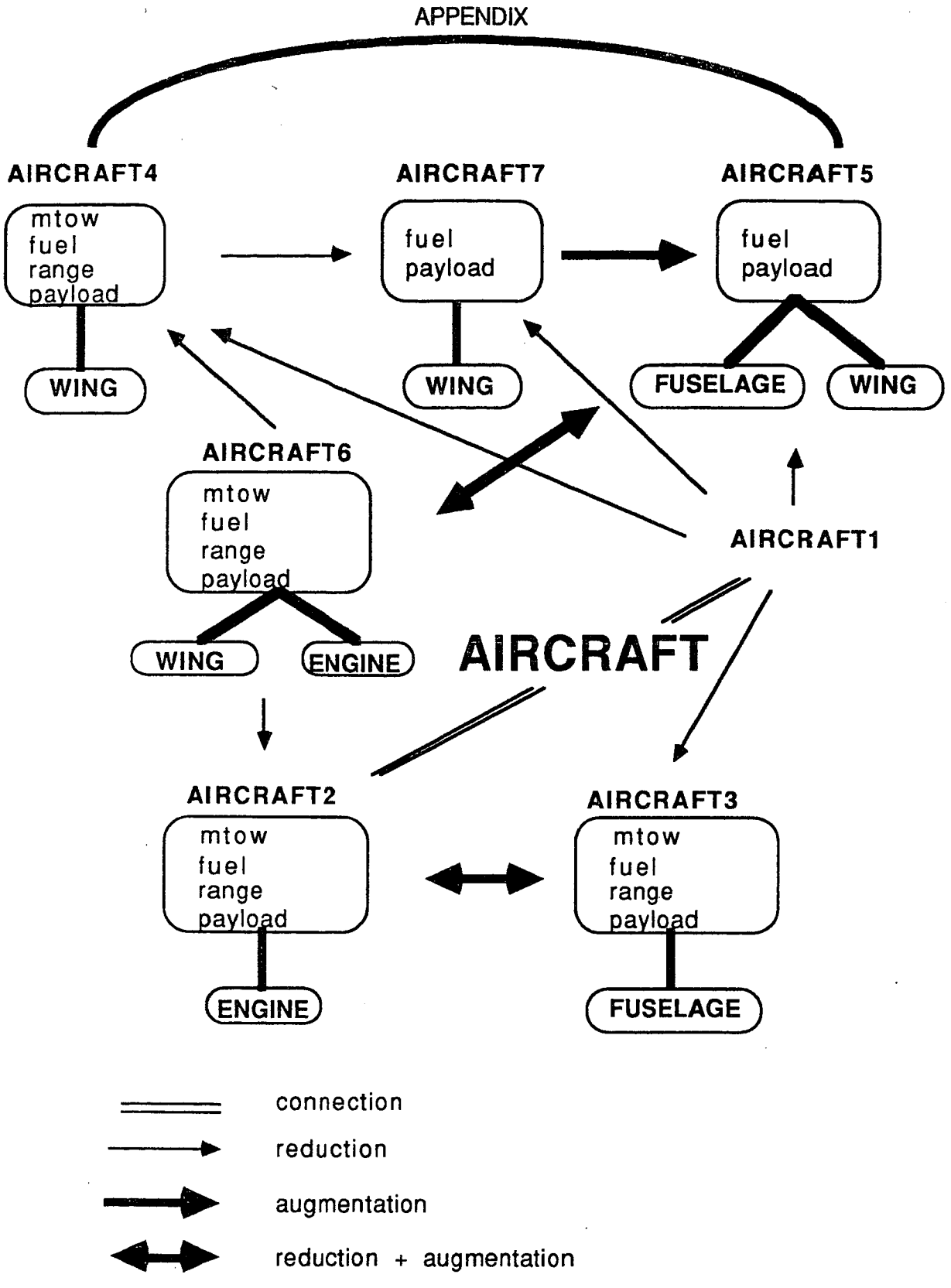

Figure 24. Some relevant classes for the class AIRCRAFT.

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