ARES, Adding a class and REStructuring Inheritance Hierarchy

H. Dicky, C. Dony, M. Huchard, T. Libourel
LIRMM,
161 rue Ada, 34392 Montpellier Cedex 5, France,
email: name@lirmm.fr

June 9, 1995

Abstract

In object-oriented databases, schema design or evolution [BKKK87] should benefit as much as possible from automatic tools.

In this paper, we focus on the automatic insertion of classes into inheritance hierarchies, while preserving a "maximal factorizing" of class properties.

We describe an incremental algorithm which factors properties, approaches the overloading problem, and can be used to build, reorganize or maintain a hierarchy.

Our algorithm works with a model of hierarchies based on what we call the "Galois SubHierarchy" recently introduced by [GM93] and which is a particular suborder of the Galois lattice.

This algorithm maintains a Galois SubHierarchy without building the whole Galois lattice.

Résumé

Dans les bases de données orientées objet, certaines phases de la conception ou de l'évolution de schéma [BKKK87] pourraient bénéficier d'une automatisation partielle.

Nous nous intéressons plus particulièrement au problème de l'insertion d'une classe dans une hiérarchie existante, en respectant un critère de "factorisation maximale".

Nous proposons un algorithme incrémental qui factorise les propriétés, aborde le problème de la surcharge, et permet de construire, réorganiser ou maintenir une hiérarchie.

Notre méthode s'appuie sur la notion de "sous-hiérarchie de Galois" introduite récemment par [GM93], et qui est un sous-ordre particulier du treillis de Galois.

Notre algorithme permet de maintenir une sous-hiérarchie de Galois sans construire le treillis de Galois tout entier.

Key words : Database schema, Inheritance hierarchy, Schema design, Schema evolution, Galois lattice, Restructuring, Factorizing

1 Introduction

In object-oriented systems, inheritance hierarchies are generally built "by hand" by a designer or a group of designers after an analysis step. The underlying construction methods are mostly informal and empirical.

It seems very difficult to completely automatize the construction of hierarchies because of the number of criteria to take into account and of the difficulty in defining what is a "good" hierarchy independently of a context.

However the structuring process should benefit from a partial automation, especially when the classes to be structured become more intricate and numerous. Some parts of the task are repetitive, (as for example building classes to factorize properties common to several classes) and human work does not ensure that the same situation will systematically be handled in the same way.

Hierarchy design may be done in a global way, given a set of unrelated classes to be organized. *A contrario*, incremental methods can be used during the design step of database schema in order to:

- insert a new class defined by the designer
- insert a class from another database schema ("integration problem").
- insert a virtual class, computed on other classes ("view class")

One of the main interests of inheritance hierarchies is to share descriptions and behavior common to several classes. In this paper, we focus on one essential activity when we build inheritance hierarchies: point out common properties and create classes to store them ("factor common properties"). We propose an incremental algorithm which performs maximal factorizing of properties defined on the classes, and offers a first solution to the problem of property overloading.

We extend different methods proposed in existing systems [Cas92, LBSL91, LBSL90, Ber91] to solve that factorizing problem. We approach property overloading, and produce a simple polynomial algorithm, whose results are well specified.

To describe the algorithm, we use *Galois lattices*, already much used in other domains (*e. g.,* machine learning, information retrieval, knowledge representation), which provide, as noted in [GM93], a nice point of view on the factorizing problem.

Section 2 introduces various definitions and notations. Section 3 explains what Galois lattices can bring to inheritance hierarchy building. Section 4 details the incremental algorithm and

its main properties. The paper ends with a general discussion about the application of the algorithm.

2 Definitions and Notations

We give here the notations used in the rest of the paper.

2.1 Database Schema and Inheritance Hierarchy

In the object-oriented database framework, a database is roughly a pair (S, I) where S is the schema, and I the persistent object instances.

The schema S is an inheritance hierarchy H, that is, a directed acyclic graph $(\mathcal{C}, \Gamma)$ where $\mathcal{C}$ is a set of classes, and Γ a set of inheritance directed edges. H induces a partial order, which we note as $<_H$, and owns a "root" (maximal vertex) Ω. Given a class C, $SuperClasses(C)$ (resp. $SubClasses(C)$) is the set of C ancestors (resp. descendants), considering $<_H$. *ImmediateSuperClasses(C)* (resp. *ImmediateSubClasses(C)*) is the set of the minimal ancestors (resp. descendants) of C.

H may contain *transitivity edges*, *i. e.* edges (C_i, C_j) such that there is a path between C_i and C_j besides the edge (C_i, C_j). In order to simplify the presentation, we do not discuss here how we handle transitivity edges.

2.2 Classes and their Properties

A class is defined by a set of properties. Properties are attributes and methods described, for instance, by a name, a signature, a domain, a value ... We now give notations for two frameworks: a naive approach without overloading and another approach in which overloading is handled. Then we show that the second approach can be reduced to the first one.

- *The Naive Approach*

We consider a set $\mathcal{P}$ of properties ($2^{\mathcal{P}}$ is the set of $\mathcal{P}$ subsets) and a set $\mathcal{C}$ of classes. To each class, the function $Properties : \mathcal{C} \rightarrow 2^{\mathcal{P}}$ associates a set of properties. For a given class C, $Declared(C)$ is the set of properties declared in C, while $Inherited(C)$ is the set of properties declared in C's superclasses [1].
The partial order $<_H$ is included into the inclusion relationship between sets of properties.

In Figure 8, $\mathcal{P}$ is $\{name, university, nb\_students, admin\_unit, laboratory\} \cup \{degree_i, 0 \leq i \leq 4\} \cup \{salary_i, 0 \leq i \leq 3\}$.

[1] $Inherited(C) = \bigcup_{C_i \in Superclasses(C)} Properties(C_i)$.

- *The Approach with Overloading*

Let us consider again the set of properties $\mathcal{P}$ and the functions *Properties*, *Declared* and *Inherited*.

In the above example, semantics leads to group together $\{degree_i, 0 \leq i \leq 4\}$ (and also $\{salary_i, 0 \leq i \leq 3\}$). We will call a *generic* property a set of properties that we want to group.

Generic property

$\mathcal{P}$ is partitioned into *generic* [2] *properties* and we shall call $\mathcal{G}$ the set of the generic properties.

In Figure 8, $\mathcal{G} = \{N, U, Nb, A, L, D, S\}$ where $N = \{name\}$, $U = \{university\}$, $Nb = \{nb\_students\}$, $A = \{admin\_unit\}$, $L = \{laboratory\}$, $D = \{degree_i, 0 \leq i \leq 4\}$, $S = \{salary_i, 0 \leq i \leq 3\}$.

We denote by p_i an *occurrence* of the generic property P.

For a given class C, we define $GenProperties(C) = \{P \in \mathcal{G}$ s.t. $\exists p_i \in Properties(C)\, and\, p_i \in P\}$. $GenDeclared(C)$ and $GenInherited(C)$ are defined the same way.

Furthermore, semantics partially orders each generic property P [3]; we shall denote by $<_P$ this partial order. Figures 1 and 2 give examples of such given partial orders. For instance, $salary_3 <_S salary_1$ because the code of the method $salary_3$ may reuse the code of the method $salary_1$. In the same way, the type of $degree_4$ may be a subtype of the type of $degree_1$ and $degree_2$.

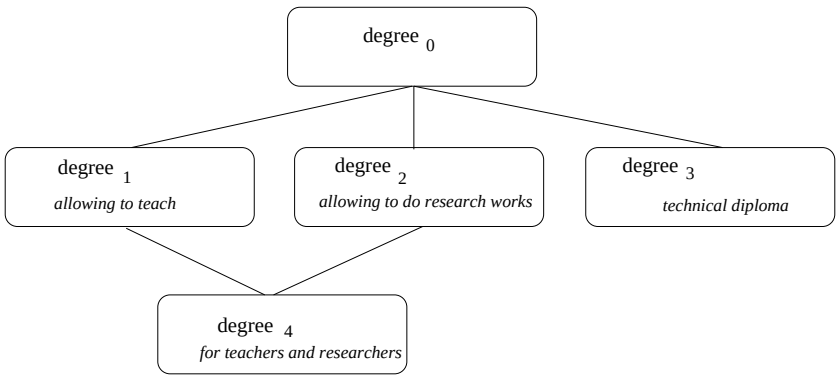

Figure 1: Example of partial order on property "Degree"

Given two different occurrences p_i and p_j of a generic property P, we call $LGB(p_i, p_j)$ (Least Greater Bounds) the set of the smallest elements which are above p_i and p_j in the partial order $<_P$.

[2] This term is inherited from CLOS system, and not from Ada, and C++ !

[3] In Casais's work [Cas92], the height of the partial orders is limited to only one level

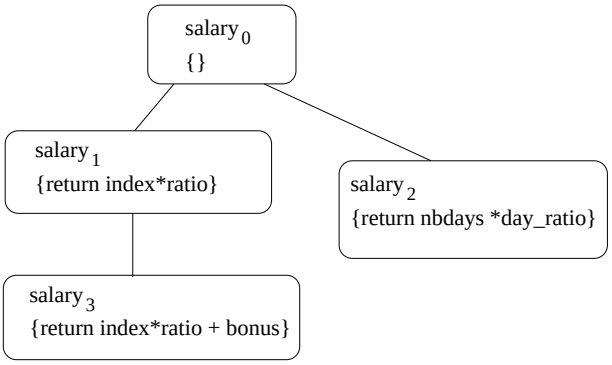

Figure 2: Example of partial order on property "Salary"

Overloading

We talk of *overloading* when several occurrences of a generic property P belong to one or more classes of the inheritance hierarchy.

Overriding

Overriding is a particular case of overloading where two occurrences of a generic property P belong to two distinct comparable classes in the inheritance hierarchy.

- *From the approach with overloading to the naive approach*

In order to reduce the approach with overloading to the naive approach, we will consider that, when a class C has a property p_i, C *de facto* owns all the properties that p_i specializes (and overrides), *i. e.* all p_j, with $p_i <_P p_j$. Let us now consider a class C and $Properties_O(C)$ the set of C properties in the "overloading" approach. One needs only to build $Properties_N(C)$ (the set of C properties in the naive approach) as follows : $Properties_N(C) = \{p_j$ s.t. $\exists p_i \in Properties_O(C)$, $p_i \leq_P p_j$, with $p_j, p_i \in P$, $P \in \mathcal{G}\}$, $Declared_N(C)$ and $Inherited_N(C)$ being defined as previously: $Inherited_N(C) = \bigcup_{C_i \in Superclasses(C)} Properties_N(C_i)$, and $Declared_N(C) = Properties_N(C) \setminus$ [4] $Inherited_N(C)$.

2.3 Meaningful Classes

Among classes, we shall set apart a subset $\mathcal{C}_{Mean}$ of *meaningful* classes. Meaningful classes are the ones the existence of which is demanded by the designer, and that the algorithm will keep. Those are, for instance, the concrete (instantiable) classes. Other classes are factorizing classes. In the OOD framework, if we suppose that all classes having persistent instances are meaningful, the database schema can be modified during the exploitation since the instances need not to

[4] $\setminus$ being the set difference

migrate. We shall use the function $SetMeaningful(C)$ to insert C into $\mathcal{C}_{Mean}$, as well as the predicate $IsMeaningful(C)$.

2.4 Maximal factorizing

For an inheritance hierarchy H, let us express what we mean by "maximal factorizing".

- *Naive approach:* let $p \in \mathcal{P}$, we cannot find two classes C_1 and C_2 such that p belongs to both $Declared(C_1)$ and $Declared(C_2)$.

  This means that, whenever two classes C_1 and C_2 own the same property, there always exists in the inheritance hierarchy a single common superclass (which can be C_1 or C_2) that declares this property.

- *"Overloading" approach:* When a class C owns a property p_i, it implicitly owns all the properties p_j that p_i specializes, thus such that $p_i <_P p_j$. If p_j is not inherited by C, we call p_j a "potential property" of C. Such a p_j is nowhere declared in the hierarchy.

  $PotentialProp(C) = \{p_j \text{ s.t. } \exists p_i \in Properties(C),\ p_i <_P p_j\} \setminus Inherited(C)$

  $PotentialDecl(C) = PotentialProp(C) \setminus \bigcup_{C_i \in Superclasses(C)} PotentialProp(C_i)$.

  The notion of maximal factorizing becomes : let $p \in \mathcal{P}$, we cannot find two classes C_1 and C_2 such that p belongs to both $PotentialDecl(C_1) \cup Declared(C_1)$ and $PotentialDecl(C_2) \cup Declared(C_2)$.

  This means that, whenever two classes C_1 and C_2 own, respectively, two different occurrences p_i and p_j of the generic property P, for each property p_k of $LGB(p_i, p_j)$, there always exists in the inheritance hierarchy a common superclass of C_1 and C_2 that declares p_k (see Figure 3).

3 What hierarchy building has to do with Galois lattices

Galois lattices [Aig79] are used in knowledge representation, under the name of "concepts lattice" [Wil89, Wil92] and also by some of the most recent studies on the organization of a hierarchy of classes [GM93, DDHL94a].

Given a set of classes (provided with their properties) to be organized, the "maximal factorizing" property is not sufficient to ensure an unique resulting hierarchy. Among the different possible results, some are more compact (properties are better regrouped) than other. Galois lattice is a structure which shows all non empty intersections between class property sets, thus

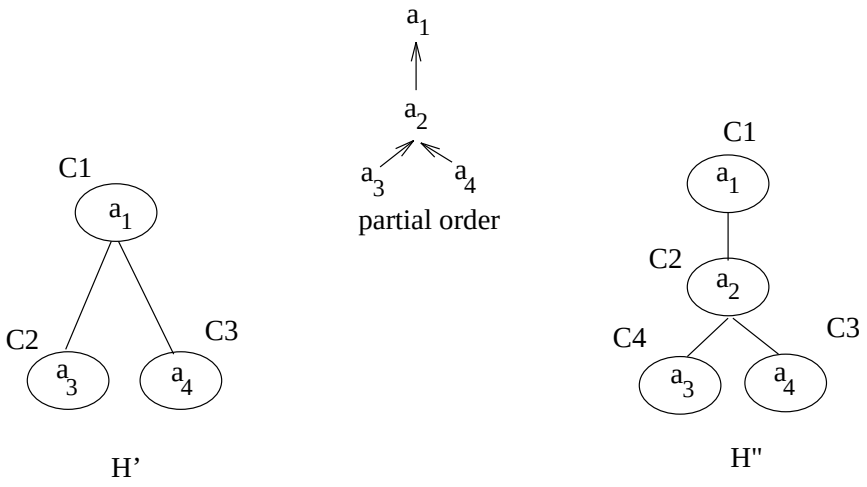

Figure 3: H' is not maximally factorized, H" is.

making explicit all the classes share, as we shall state after the definition of Galois lattice. As a consequence, it is possible to build from Galois lattice the unique more compact hierarchy where properties are maximally factored.

We define these notions in the following.

Galois lattice (from [Bor92]). Let $\mathcal{C}$ and $\mathcal{P}$ be two finite sets and $\mathcal{R}$ a binary relation upon $\mathcal{C} \otimes \mathcal{P}$. Within the inheritance framework, $\mathcal{C}$ will be the set of classes, $\mathcal{P}$ the set of properties, and $\mathcal{R}$ the binary relation "owns as a property". The Galois lattice $GL(\mathcal{R})$ is defined as follows:

- members of $GL(\mathcal{R})$ are Cartesian products $\mathcal{K} \otimes \mathcal{F}$ with
  1) $\mathcal{K} \subseteq \mathcal{C}$, $\mathcal{F} \subseteq \mathcal{P}$, and $\forall C \in \mathcal{K}$, $\forall P \in \mathcal{F}$, we have $\mathcal{R}(C, P)$
  2) $\mathcal{K} \otimes \mathcal{F}$ is maximal under inclusion for property 1 [5].
- the members of $GL(\mathcal{R})$ are totally ordered by the relation denoted by $<$ and defined as follows :
  $\mathcal{K} \otimes \mathcal{F} < \mathcal{K}' \otimes \mathcal{F}' \Leftrightarrow \mathcal{K} \subset \mathcal{K}'$ (which is equivalent to $\mathcal{F} \supset \mathcal{F}'$)

Figures 4 and 5 [6] show a binary relationship and its Galois lattice.

This lattice is isomorphic to the lattice built upon the intersections between sets of properties of $\mathcal{C}$ classes, lattice in which those intersections are ordered by inclusion. We build the isomorphism when we change the label vertices and keep only the right member of the Cartesian

[5] $\mathcal{K} \otimes \mathcal{F}$ satisfying prop. 1 is maximal under inclusion for prop. 1 if there is no $\mathcal{K}' \otimes \mathcal{F}'$ satisfying prop. 1, and such that, $\mathcal{K} \otimes \mathcal{F} \neq \mathcal{K}' \otimes \mathcal{F}'$, $\mathcal{K} \subseteq \mathcal{K}'$ and $\mathcal{F} \subseteq \mathcal{F}'$

[6] Usually, only Hasse diagram (also called transitive reduction) is drawn to represent orders and especially lattices.

$\mathcal{C}$ \ $\mathcal{P}$	a	b	c	d	e
1	x	x			
2			x		
3	x		x	x	
4	x		x		x

Figure 4: a binary relation $\mathcal{R}$ "owns as a property" upon $\mathcal{C} \otimes \mathcal{P}$ with $\mathcal{C} = \{1, 2, 3, 4\}$ and $\mathcal{P} = \{a, b, c, d, e\}$

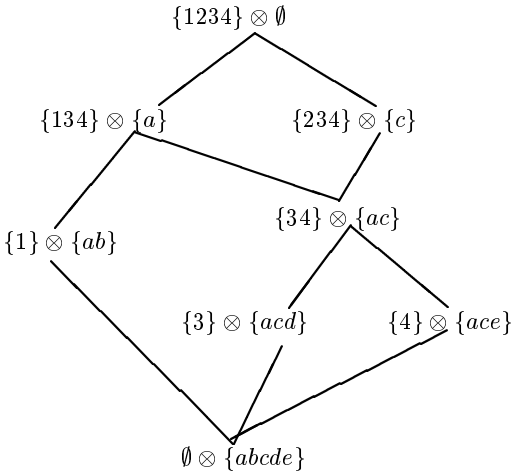

Figure 5: Galois lattice built from Figure 4 example

product.

Another labelling will get rid of redundant information, in order to get something looking more like an inheritance hierarchy. A vertex v, labeled by $\mathcal{K} \otimes \mathcal{F}$ will thus be labeled by $\mathcal{K}_r \otimes \mathcal{F}_r$, where $\mathcal{K}_r$ is the subset of those classes of $\mathcal{K}$ that do not appear below v in the lattice, and $\mathcal{F}_r$ the subset of those properties of $\mathcal{F}$ that do not appear above v (see Figure 6).

Galois inheritance lattice

Galois inheritance lattice $GIL(\mathcal{R})$ is isomorphic with Galois lattice. To $V = \mathcal{K} \otimes \mathcal{F} \in GL(\mathcal{R})$ one associates in $GIL(\mathcal{R})$ the member $V_r = \mathcal{K}_r \otimes \mathcal{F}_r$ with $\mathcal{K}_r = \mathcal{K} - \bigcup_{V' \in SubClasses(V)} \mathcal{K}'$, and $\mathcal{F}_r = \mathcal{F} - \bigcup_{V' \in SuperClasses(V)} \mathcal{F}'$, where we note $V' = \mathcal{K}' \otimes \mathcal{F}'$.

$GL(\mathcal{R})$ and $GIL(\mathcal{R})$ are just two different labellings of the same structure. Such a structure has an important size, exponential in min(number of classes, number of properties); one can

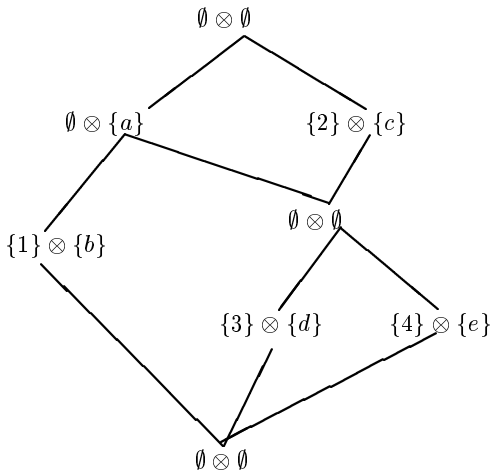

Figure 6: Galois Inheritance Lattice built from Figure 4 example

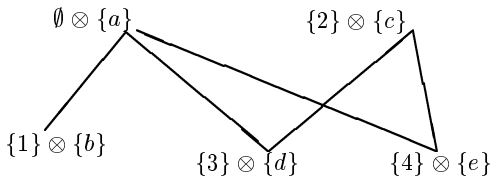

Figure 7: Galois SubHierarchy built from Figure 4 example

store only a part of it in a structure [GM93] which is defined below. This last structure (see Figure 7) seems to be the relevant one to build inheritance hierarchies when property factorizing is emphasized.

Galois SubHierarchy

The Galois SubHierarchy $GSH(\mathcal{R})$ is the order deduced from $GIL(\mathcal{R})$ by removing "empty" vertices, *i. e.* vertices such that $V_r = \emptyset \otimes \emptyset$.

Vertices can be deleted because they do not help to declare properties, —all their properties appear in the labels of higher vertices— and because no class of $\mathcal{C}$ owns exactly their set of properties. Due to the difference of size between Galois lattice $GL(\mathcal{R})$ and Galois SubHierarchy $GSH(\mathcal{R})$, algorithms that build directly $GSH(\mathcal{R})$ seem more suitable than algorithms that build $GL(\mathcal{R})$ and then reduce $GL(\mathcal{R})$ to $GSH(\mathcal{R})$. In adapting the algorithm of [MGG90], one gets a "global" algorithm, that builds the whole $GSH(\mathcal{R})$ from the class descriptions, here the $\mathcal{R}$ matrix. In Section 4, we give the "incremental" [7] algorithm, that inserts a class in an already built $GSH(\mathcal{R})$.

[7] This term may have several senses, it is used here to express the fact that classes are inserted one after the other

4 The *ARES* algorithm

This section describes the *ARES* algorithm that inserts a class in a class hierarchy defined as in Section 2. First, we run *ARES* on an example. After that, we give *ARES* specification, description and complexity. We then show that *ARES* is an incremental insertion algorithm in a Galois subhierarchy. This ensures that, whenever *ARES* is used alone to build up a hierarchy, the result does not depend on the order of insertion.

4.1 Through an example

Figure 8 shows a hierarchy H and a class A to be inserted, $A = Invited\_Prof$. The properties *degree* and *salary* are ordered as in Figures 1 and 2. Figure 9 shows the final hierarchy. The algorithm deals with all the classes following a linear extension of $>_H$: thus a class is taken into account after all its superclasses. We shall consider the following linear extension of Figure 8: Ω, *Person, Researcher, Teacher, Teacher_Researcher, Administrator, Teacher_Researcher_Administrator*.

For any class C, the properties of $Declared(C)$ are in roman-face, while italics are used for the properties of $Inherited(C)$ [8] . For the class $Invited\_Prof$, italics are used for the potential properties.

First the classes Ω, *Person* and *Researcher* are explored: their property set is included in the property set of A, thus they are superclasses of A [9] .

While exploring the class *Teacher*, the algorithm creates a class, that we call *"Able to teach"*, in order to factor the properties declared by *Teacher* and that belong to $Properties(A)$, here $\{degree_1\}$. *"Able to teach"* is superclass of *Teacher* and subclass of the superclasses common to *Teacher* and A, *i. e. Person*. The edge (*Teacher*, *Person*) becomes a transitivity edge and disappears.

Such an extraction upon the class *Teacher_Researcher* produces the class *"Able to Teach-in High School"*. The exploration of *Administrator* and *Teacher_Researcher_Administrator* has no effect. A is linked to its immediate superclasses, in the example, to *"Able to Teach in - High School"*. In Figure 9, A has no subclass, since no class of H owns all the properties of A.

[8] We have represented all the inherited properties, including the overriden properties

[9] The superclasses of A appear in Figure 9 in rectangular boxes

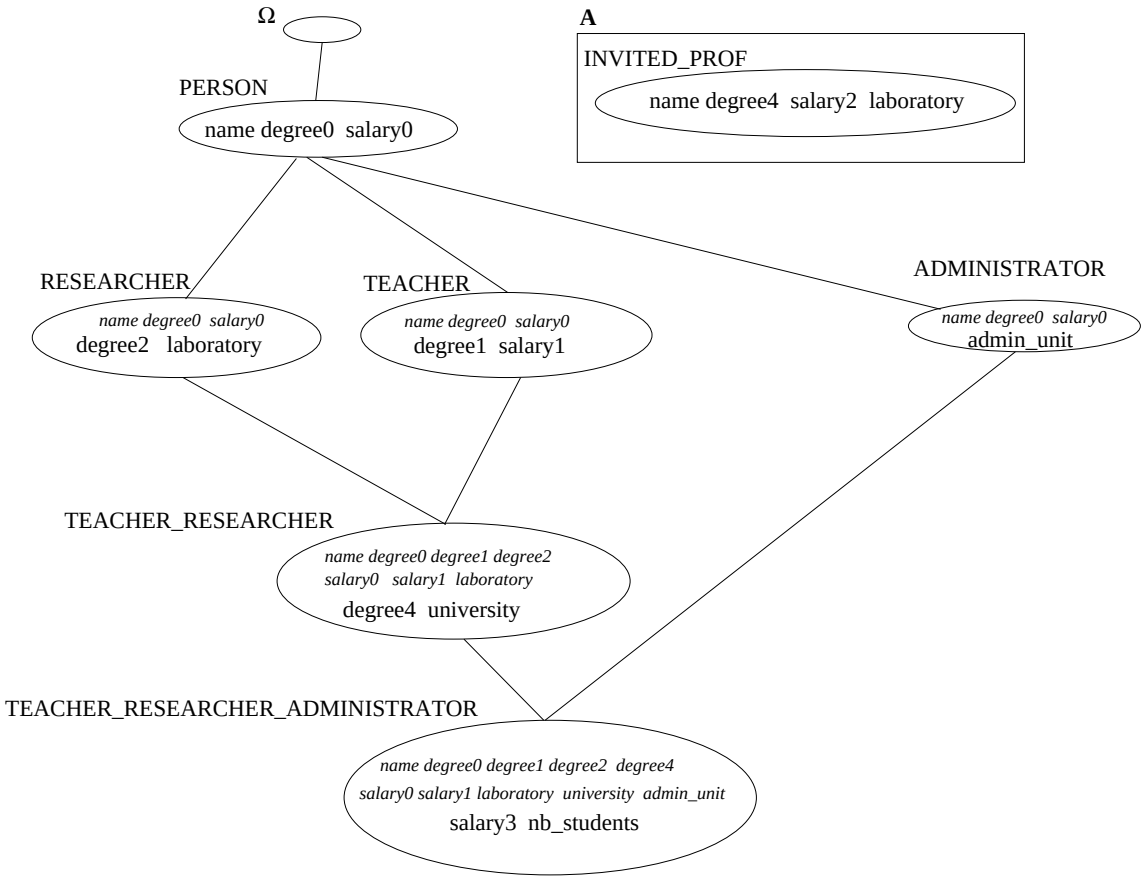

Figure 8: Initial hierarchy and a class A to be inserted

4.2 Specifications

Input :

The algorithm starts with $H_i = (\mathcal{C}_i, \Gamma_i)$ a class hierarchy with root Ω [10] and with a meaningful class A to be inserted. $Properties(A)$ is A's property set. The framework is the naive approach, and in H_i the properties are maximally factored: thus any property is declared only once (the transformation described in Section 2.2 allow to take overloading into account).

Output :

The final hierarchy $H_f = (\mathcal{C}_f, \Gamma_f)$ "integrates" H_i and A and respects the following properties, whose proofs are given in [DDHL94b].

- **Maximal factorizing of properties**
- **Transitive closure preservation of the hierarchy**: For all the classes of H_i still belonging to H_f, the inheritance paths remain.

[10] Ω has no properties

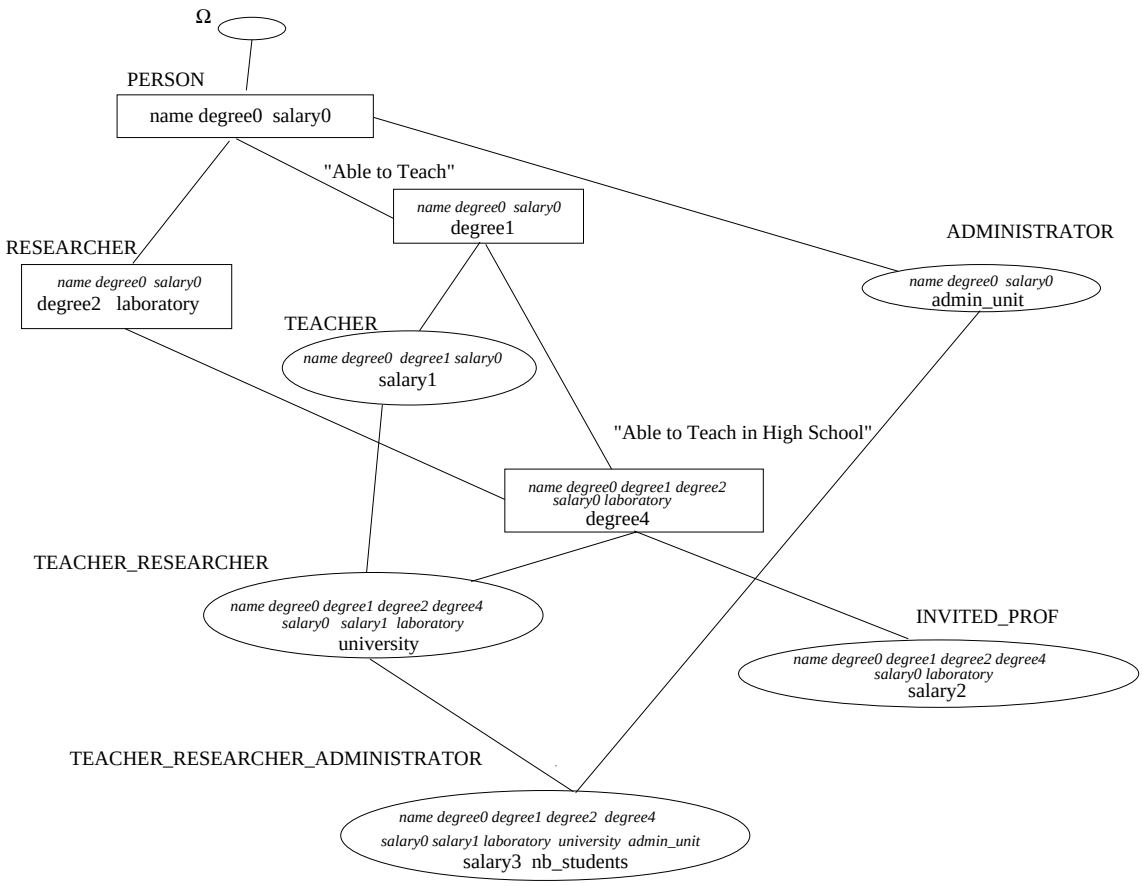

Figure 9: Final hierarchy

- **Conservation of the properties of input classes**: Classes which belong to both hierarchies H_i and H_f keep the same set of properties.
- **Meaningful class conservation**: The set of meaningful classes of H_f is $\mathcal{C}_{Mean} \bigcup \{A\}$

4.3 Description

For presentation reasons, we give the algorithm in two steps, and only detail the first one:

- $LookFor\&BindSuperClasses$ recognizes A's superclasses and binds A to its immediate superclasses.
- $BindSubClasses$ binds A to its immediate subclasses.

We do not speak about transitivity edges. A fully detailed algorithm can be found in [DDHL94b]. The algorithm also uses some global variables:

- $AalreadyCreated$, a Boolean which is true if and only if a class whose set of properties equals A's property set is found or built as a factorizing class. $AalreadyCreated$ initial

value is false.

- SH, the current set of A's superclasses. SH initial value is empty.
- $EmptyClasses$, the set of the non-meaningful classes which set of declared properties has been cleared out. $EmptyClasses$ initial value is empty.

```
Algorithm ARES(H_i,A)
  begin
    LookFor&BindSuperClasses
    BindSubClasses
  end
```

The algorithm $LookFor\&BindSuperClasses$ visits H_i going down from Ω, following a linear extension of $>_{H_i}$. During this visit, A's property set is compared with the set of properties of the visited class. The whole exploration builds SH, the set of A's superclasses in H_f. Then A is —if needed— created and bound to its direct superclasses ($Create\&BindSH$). We end and delete ($DeleteEmptyClasses$) each class of the set $EmptyClasses$, since these classes are non-meaningful, and at this point do not declare any more properties.

```
Algorithm LookFor&BindSuperClasses
  begin
      For every vertex C following LEH_i do
          // LEH_i is an arbitrary linear extension of H_i starting from root Ω
          Visit(C, A)
      endFor
      Create&BindSH
      DeleteEmptyClasses
  end
```

When a class C is visited, remember that all its superclasses have already been visited. $Visit$ deals with the easy cases when either C is A or C is a superclass of A, and calls $Extract$ whenever a factorizing class is needed.

```
Algorithm Visit(C, A)
  begin
      if Properties(A) = Properties(C) then SetMeaningful(C); AalreadyCreated ← true
      else if Properties(A) ⊃ Properties(C) then SH ← SH ⋃ {C}
          else if Properties(A) ∩ Declared(C) ≠ ∅ then Extract(C) endif endif endif
  end
```

The $Extract$ algorithm creates a factorizing class C' and inserts C' into the hierarchy. This factorizing class either is A or is stored inside SH. SH holds already visited H_i classes which are A's superclasses, as well as the factorizing classes (obviously A's superclasses) built up in the previous steps. Given a class C, we consider the set S of classes which are C's superclasses while belonging to SH, and we call $Sups(C, SH)$ the minimal elements of S.

```
Algorithm Extract(C)
  begin
      Create(C'); Declared(C') ← Properties(A) ⋂ Declared(C);
      ImmediateSuperclasses(C') ← Sups(C, SH);
      Inherited(C') = ⋃_{C"∈Sups(C,SH)} Properties(C");
      Properties(C') ← Declared(C') ⋃ Inherited(C');
      ImmediateSuperClasses(C) ← (ImmediateSuperClasses(C) ⋃{C'}) \ Sups(C, SH)
      Declared(C) ← Declared(C) \ Declared(C')
      Inherited(C) ← Inherited(C) ⋃ Declared(C')
      if Properties(C') = Properties(A)
          then SetMeaningful(C'); AalreadyCreated ← true;
          else SH ← SH ⋃{C'} endif
      if Declared(C) = ∅ and not IsMeaningful(C)
          then EmptyClasses ← EmptyClasses ⋃{C} endif
  end
```

When $Create\&BindSH$ is called by $LookFor\&BindSuperClasses$, SH contains all A's superclasses in the current graph. If A has not already been found, it must be created and bound together with its immediate superclasses. We use the function $Min(E)$ which returns the minimal (for $<_H$) classes of set E.

```
Algorithm Create&BindSH
  begin
      if not AalreadyCreated then
          Create(A); SetMeaningful(A);
          ImmediateSuperClasses(A) ← Min(SH)
          Inherited(A) = ⋃_{C"∈Min(SH)} Properties(C")
          Declared(A) ← Properties(A) − Inherited(A)
      endif
  end
```

The algorithm $DeleteEmptyClasses$ is not described here. $ARES$ ends with $BindSubClasses$ that binds A to its immediate subclasses. A class T is a subclass of A when $Properties(A)$ is included in $Properties(T)$. T is an immediate subclass if none of T's superclasses is itself a subclass of A.

4.4 Complexity

With very rough approximations, we find an algorithm complexity that belongs to $\mathcal{O}(n*(\omega*p+m))$, where n is the number of H_i classes, m the number of H_i edges, ω its width, (i.e. the biggest number of incomparable classes in the graph, which is for instance 3 in Figure 8), and p is the maximal number of properties of a class (12 on the same figure). A detailed study of complexity can be found in [DDHL94b]. In the approach with overloading, to keep an interesting cost, some modifications will have to be brought, especially to avoid storing the potential properties of each class.

4.5 *ARES* is an incremental algorithm for maintaining *Galois SubHierarchies*

Galois lattice model allows to specify *ARES* whenever it is used alone to build up a hierarchy. The result below (for a proof see [DDHL94b]) shows that *ARES* is indeed *an incremental insertion algorithm in a Galois SubHierarchy.*

Property

Let $H = (\mathcal{C}, \Gamma)$ be a hierarchy, $\mathcal{C}_{Mean}$ the set of meaningful classes in H, and A the class to be inserted. If H is the Galois subhierarchy of $\mathcal{C}_{Mean}$, then *ARES* builds up the Galois subhierarchy of $\mathcal{C}_{Mean} \bigcup \{A\}$ (which is the set of meaningful classes of the resulting hierarchy).

5 Conclusion and discussion

The Galois lattices provide a nice general method to build inheritance hierarchies. The algorithm *ARES* allows an "exact" insertion of a class in a Galois subhierarchy. The above property ensures that, whenever a hierarchy is created from scratch, the result does not depend on the order in which the classes are inserted.

A prototype of the algorithm has been implemented and tested in ObjVlisp [Coi87].

We now give a short overview of the problems on which we are currently working, which arise in concrete situations.

• An inheritance hierarchy, even with maximal property factorizing, is not always the Galois subhierarchy of the set of its classes, nor even the Galois subhierarchy of the set of its meaningful classes [DDHL94a].

• Partial orders upon properties are often implicit for the designer. They may be made explicit from a preexisting type hierarchy: the partial order on an attribute could be deduced from its type; the partial order on a method could be, in easy cases, deduced from its signature. In other cases the partial order could come from code dependencies between the different occurrences of a method (Figure 2). Many other criteria may be used.

• But even when these orders are known, the factorizing produced by *ARES* may have to be transformed in several ways either to match specific languages capabilities or to "interpret" the semantics of the domain.

Firstly, the resulting hierarchy cannot be encoded in those languages that do not support multiple inheritance, except through further graph modifications as proposed in [GM93].

Secondly, let us look at an example which illustrates problems occurring in concrete overriding situations (Figure 10). Let X and Y be two types. Let p_1 (resp. p_2) be an attribute of

name p and type X (resp. Y), p_1 and p_2 both belong to the generic property P. In that case, it can easily be admitted that the order between p_1 and p_2 is the same as the one between X and Y. Let C_1 (resp. C_2) be the class composed of property p_1 (resp. p_2). We now work on a class hierarchy containing only C_1 and in which we want to include C_2. p_1 and p_2 are incomparable but $LGB(p_1, p_2) = p : S$. When inserting C_2, *ARES* creates a new class CS (superclass of C_1 and C_2) which defines a property p of type S in order to show that factoring is possible, and it leaves properties p_1 and p_2 with their correct types on C_1 and C_2.

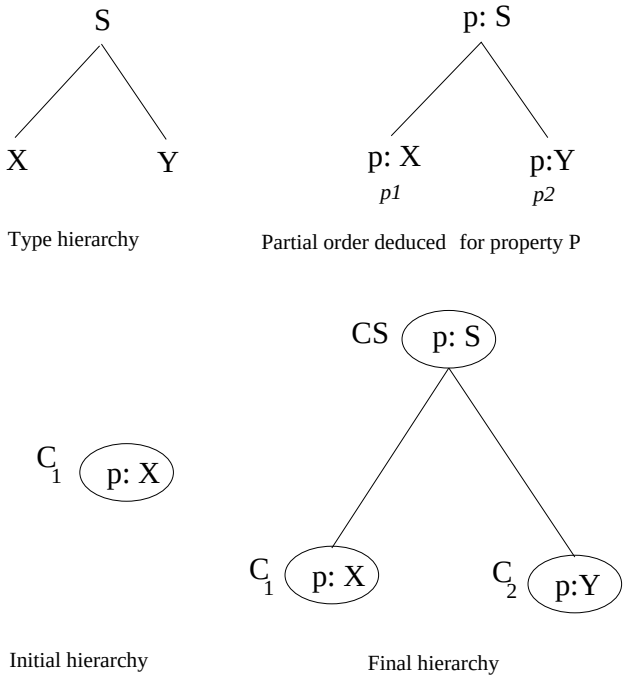

Figure 10: Attribute overriding example

Here are some possible adaptations of *ARES*'s result:

1. the result is not modified: this is possible in *Eiffel* and in O_2C where an attribute can be specialized in a subclass with a compatible more specific type. The presence of p on CS makes sense, for example, when a method that uses p is shared by C_1 and C_2 and is factored on CS.

2. p is removed from C_1 and C_2 and factored on CS with name p and type S, p is inherited by C_1 and C_2, the type constraint on p_1 and p_2 is relaxed. This is the only way to implement the previous example in *C++*. Indeed, an attempt to override an attribute in a subclass C fails: it leads to a situation in which C owns two different attributes.

3. p_1 and p_2 should not be factored on CS. This situation can occur at least in the following configuration: in languages in which all classes have a common superclass ω, the situation

where $S = \omega$ can mean that C_1 and C_2 own by accident a property with the same name p but with no common semantics. It seems difficult to take the "no-factorizing" decision without an external human operator checking the algorithm's result.

Thirdly, a factorizing class declaring two properties could be split into two different factorizing classes (see Figure 11), without losing the maximal factorizing property. The choice between the two depends on the semantics of the domain.

Furthermore, the pursuit of the maximal factorizing criterion may multiply the number of factorizing classes. Sometimes it is a good thing, when it creates "good" reusable classes.

Figure 11: Two different maximal factorizings

But sometimes, when the partial order holds more information than needed for building the hierarchy, some factorizing classes are useless. In hierarchy H'' of Figure 3, for instance, the designer could see no meaning in the creation of class C_2. The partial order relevant to the building of the hierarchy would in fact be the order induced by $\{a_1, a_3, a_4\}$.

Our current work consists in applying the algorithm to concrete hierarchies written in different languages. We have already identified how to build partial orders for attributes of classes (a simple example was given in this section, with Figure 10). The case of methods is more complex since code and domain, codomain result types have to be taken into account. It is already clear that some cases will require an external human help but also that many cases are simple enough to be automated.

Acknowledgement : With sincere thanks to anonymous referees for their helpful comments.

"Where do right ideas come from ? From class consciousness" Chairman Mao

References

[Aig79] M. Aigner. *Combinatorial Theory.* Springer-Verlag, 1979.

[Ber91] P. Bergstein. Object Preserving Class Transformations. *Proceedings of OOPSLA'91*, 1991.

[BKKK87] J. Banerjee, W. Kim, K.J. Kim, and H. Korth. Semantics and implementation of schema evolution object-oriented databases. *Proc. ACM SIGMOD Conf.*, 1987.

[Bor92] J. P. Bordat. *Sur l'algorithmique combinatoire d'ordres finis. Thèse d'état.* Université Montpellier 2, 1992.

[Cas92] E. Casais. An incremental class reorganization approach. *ECOOP'92 Proceedings*, 1992.

[Coi87] P. Cointe. Metaclasses are First Class : The ObjVlisp Model. *OOPSLA'87 Proceedings*, 1987.

[DDHL94a] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, un algorithme d'Ajout avec REStructuration dans les hiérarchies de classes. *Actes de Langages et Modèles à Objets 94*, 1994.

[DDHL94b] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, un algorithme d'Ajout avec REStructuration dans les hiérarchies de classes. Technical report, LIRMM, 1994.

[GM93] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *OOPSLA 93 Proceedings*, 1993.

[LBSL90] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. Abstraction of object-oriented data models. *Proceedings of International Conference on Entity-Relationship*, 1990.

[LBSL91] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. From objects to classes: Algorithms for optimal object-oriented design. *Journal of Software Engineering*, 1991.

[MGG90] Guy Mineau, Jan Gecsei, and Robert Godin. Structuring Knowledge Bases Using Automatic Learning. *Proceedings of the sixth International Conference on Data Engineering*, 1990.

[Wil89] R. Wille. Knowledge acquisition by methods of formal concept analysis. *Data Analysis, Learning Symbolic nd Numeric Knowledge*, 23, 1989.

[Wil92] R. Wille. Concept lattices and conceptual knowledge systems. *Computers Math. Applic*, 23, 1992.