On Automatic Class Insertion with Overloading

H. Dicky, C. Dony, M. Huchard, T. Libourel

LIRMM: Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier
161, rue Ada – 34392 Montpellier Cedex 5 – FRANCE
email: dicky,dony,huchard,libourel@lirmm.fr

Abstract

Several algorithms [Cas92, MS89, Run92, DDHL94a, DDHL95, GMM95] have been proposed to automatically insert a class into an inheritance hierarchy. But actual hierarchies all include overriden and overloaded properties that these algorithms handle either very partially or not at all. Partially handled means handled provided there is a separate given function f able to compare overloaded properties [DDHL95, GMM95].

In this paper, we describe a new version of our algorithm (named *Ares*) which handles automatic class insertion more efficiently using such a function f. Although impossible to fully define, this function can be computed for a number of well defined cases of overloading and overriding. We give a classification of such cases and describe the computation process for a well-defined set of nontrivial cases.

The algorithm preserves these important properties:
- preservation of the maximal factorization of properties
- preservation of the underlying structure (Galois lattice) of the input hierarchy
- conservation of relevant classes of the input hierarchy with their properties.

1 Introduction

This paper deals with automatization of the insertion of a class (defined by a set of properties) into an existing inheritance hierarchy, we will refer to this as the "class insertion" problem. It also deals with automatic inheritance hierarchy construction or reorganization which is related to the "class insertion" problem. We propose, via a new algorithm, new advances to fill the gap between what current class insertion algorithms are able to do and what automatic handling of actual inheritance hierarchies really requires.

Why automate hierarchy construction? Class or object (Some programming or knowledge representation object-oriented languages are classless [DMC92]) inheritance hierarchies are at the heart of object-oriented programs, object knowledge-bases and object data-bases, and they are a cornerstone of frameworks *i. e.* of adaptable and reusable object-oriented architectures. Any kind of automated help in building, reorganizing or maintaining hierarchies can thus be of interest and can have applications in several important research areas of object technology:

- organization of object-oriented frameworks [JF88]: automatic reorganization is able to bring to the fore new factorization classes and abstract classes [OJ93].
- adaptation of legacy systems: numerous object-oriented systems, thus numerous hierarchies, have been developed in the past years, automatic reorganization can help to adapt or reuse them,
  - by reorganizing poorly designed systems built either by nonspecialists, or too rapidly, or without any concern for generalization,
  - by reorganizing huge systems built by different designers or programmers at different time periods,

- by merging hierarchies: the final hierarchy could be computed by reclassifying classes from the different hierarchies. This approach should not be confused with hierarchy combination, as proposed in [OH92], where a methodology is proposed to extend existing hierarchies.

- software adaptability: automatic insertion of a class adds flexibility to an object-oriented software system, which becomes for example able to undergo change.

Independently of the application area, the more the classes to be structured multiply and become intricate, the more the structuring process can benefit from partial automatization.

Given these possible applications, the next question that emerges is: what kind of methods can be provided?

Before going further, it should be stated that it would certainly be impossible to find a general algorithm that could completely automate, generally speaking, class insertion and/or hierarchy reorganization; firstly, because of the difficulty in expressing criteria to define a "good" hierarchy independently of the context, and secondly, because the construction rules are often very informal and empirical.

The different works describing algorithms for automatic class insertion or hierarchy reorganization that have been published [GM93, Cas92, LBSL91, LBSL90, Ber91, MS89, Run92, DDHL94a, DDHL95, Moo95, MC96] focus on the most tangible and one of the most important criteria used when organizing hierarchies: to point out common properties and create classes to store them (*i.e.* "factor common properties").

Once this criterion is set, there is room for multiple variations: incrementality, maximal factorization, conditions on inputs and outputs of the algorithm, constraints imposed by a particular application domain.

Finally, a common and fundamental characteristic of object-oriented programs, knowledge representation and database hierarchies is that they include properties whose name's are overloaded. So usable and actual class insertion algorithm has to correctly handle overloading. Most existing algorithms do not handle this issue and, when done, it is only partial [DDHL95, GMM95]. The main issue concerning overloading in our context is to compare properties of the same name using their signatures and codes. Unfortunately, code comparison is undecidable.

This paper describes *Ares*, explains how we achieve property comparisons in a number of well-defined cases, and how we use this procedure to efficiently insert classes in the presence of overloading.

In Section 2, we present the terminology used. In Section 3, some commented examples of algorithm inputs-outputs are proposed that highlight its main properties and give an idea on the way overloading is handled. Section 4 compares our approach with related works. Section 5 gives a detailed description of the algorithm that takes overloading into account.

Then a thorough study of how to compare occurrences of generic properties, the key-problem for handling overloading, is presented.

2 Terminology and context

Before describing examples of class insertion, in the light of the fact that words such as "overloading", "properties", "genericity", "signature" are somehow overloaded in the world of object-oriented languages, let us first introduce the classical terminology, and terms specific to our problem.

The algorithm will be applicable, provided it is correctly interfaced, to inheritance hierarchies for various object-oriented systems. Designing an algorithm interface for a particular language may be complicated. In order to describe the algorithm, we have chosen the global context of a standard class-based object-oriented language with inclusion polymorphism, property overloading and overriding.

2.1 Classes, inheritance, properties

Classes and types are assimilated, and basic types are interfaced and can be considered as classes. Classes are organized into an *inheritance hierarchy* H with a root. The subclass relationship induces a partial order, which we denote by $>_H$.

A class is characterized by a *set of properties*. Class properties can be either instance variables or methods (*Smalltalk* terminology). We will refer to variables and methods under the terms *property* or *class property*.

All properties have a name and other characteristics such as a *signature*, and in the case of methods they may have a *body* or *code* (a set of instructions).

The signature of an instance variable represents its type. The signature of a method is the ordered list of its parameter types and possibly its return type. Traditionally, the first element of a signature is the receiver type. In this presentation, the signature does not include this first element.

For a given class C, $Declared(C)$ denotes the set of properties declared in C, and $Inherited(C)$ is the set of properties declared in C superclasses.

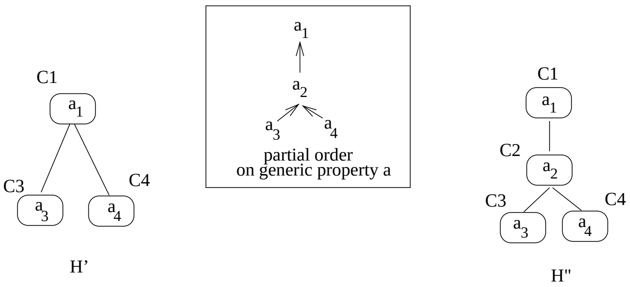

Figure 1: H' is not maximally factorized, H" is.

2.2 Overloading, overriding and generic properties

Properties can be *overloaded*, *i.e.* it is possible to find properties with the same name and different characteristics (signature, code , *etc.*).

Overriding is a particular case of overloading which makes sense in the presence of inheritance and applies when a redefined property hides, for a certain object, a property of the same name that is otherwise inherited. The rules of conformance that govern signature redefinition are language dependent[1]. The conformance rule for signature redefinition is one point to be specified when the algorithm is to be applied. We present *Ares* using an Eiffel-like covariance policy [Mey92] for variable and method redefinitions.

We have to mention the set of all class properties with the same name and same arity (in case of methods). We call such a set *a generic property* [2].

Each property belongs to a generic property, *i.e.* is an element, or an *occurrence* of the set of properties having the same name, OGP stands for *Occurrence of a Generic Property.*

P denotes a generic property, and p or p_i an occurrence of P, the index is used when necessary, *i.e.* when we want to speak, in the same context, about two distinct occurrences of P.

The different occurrences of P are ordered by a "specialization" order. For variables, this specialization order can be deduced from the specialization order on their types. For methods, this specialization order can be deduced from a specialization order on the signatures and then on a specialization order on method bodies[3]. A ticklish problem arises when we admit "self-reference" in signatures[4].

We call "lowest common generalizations" and use $LCG(p_i, p_j)$ to denote the set of the most specialized common generalizations of two occurrences of the same generic property. In most cases, $LCG(p_i, p_j)$ is a single element set. In the following, we will assimilate this single element with the set. This simplification does not hide difficult problems.

$p(C_1, C_2) : C_3[code]$ denotes a method with signature (C_1,C_2,C_3), where C_3 is the return type, and *code* is the method's body.

We also denote: p_0 or $p()[= 0]$ for a *subclass responsibility* or *pure virtual* method with an empty code. Such a method is automatically the top of the specialization order of P.

[1] For example, concerning methods, the rules are different in Eiffel (multi-covariance, where the type of several or all parameters of a method can be specialized in method redefinitions) and C++ (simple-covariance, only the receiver can be specialized)

[2] same name and same meaning than *Clos* generic functions; note that this notion is reified in *Clos* but is common to all object-oriented languages, for example we can speak of the generic property *printOn:* in Smalltalk, which is the set of all methods named "printOn:" defined in the system

[3] A method that performs a *super* call could be considered as a specialization of the method invoked by this call

[4] a signature is "self-referent" when it contains the type of the method's receiver

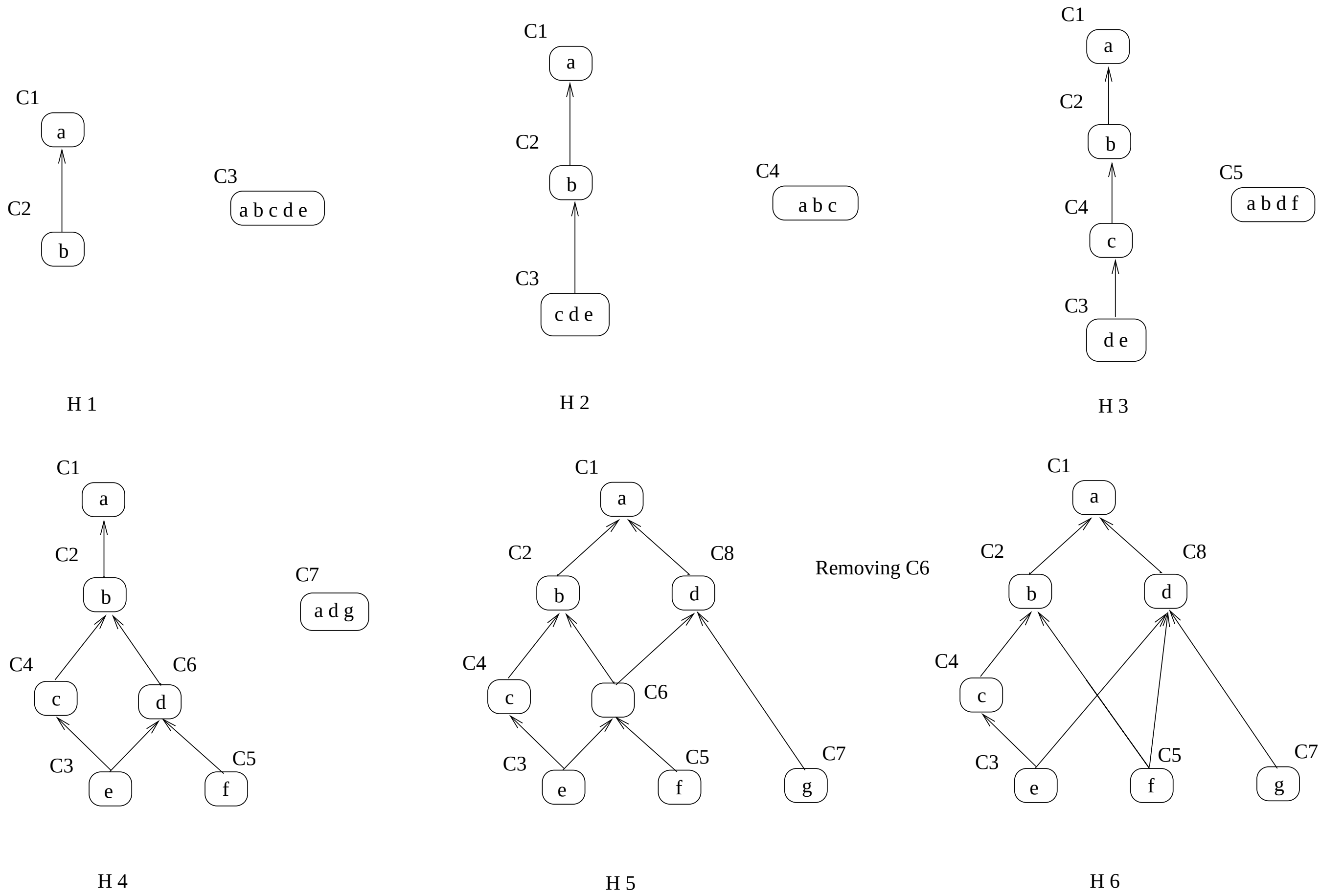

Figure 2: Insertions without overloading

2.3 Meaningful classes

The designer may arbitrarily set apart a subset $\mathcal{C}_{Mean}$ of *meaningful* classes. The algorithm will not be allowed to delete these meaningful classes from the hierarchy. Examples of meaningful classes could be: classes with instances (of great importance in a persistent world) or classes which represent an interesting abstract concept.

2.4 Maximal factorization

An inheritance hierarchy is maximally factorized if and only if, for any two classes C_3 and C_4 with two properties a_3 and a_4 respectively, and for $LCG(a_3, a_4) = a_2$, the hierarchy always includes a common superclass of C_3 and C_4 that declares a_2, such that $a_2 = LCG(a_3, a_4)$ (*cf.* Figure 1).

3 Commented examples of inputs-outputs of the algorithm

Before formally describing the algorithm, we will comment on a few examples of class insertions as they are performed by *Ares*.

3.1 Examples without overloading

Here is a sequence of class insertions (*cf.* Fig. 2) starting from hierarchy H_1 and successively producing hierarchies H_2 to H_6, highlighting decisions taken by *Ares* and showing how the maximal factorization property holds:

- the inserted class is a simple subclass of an existing class.
  The first example shows an initial hierarchy H_1 reduced to classes C_1 and C_2 and a class C_3 to

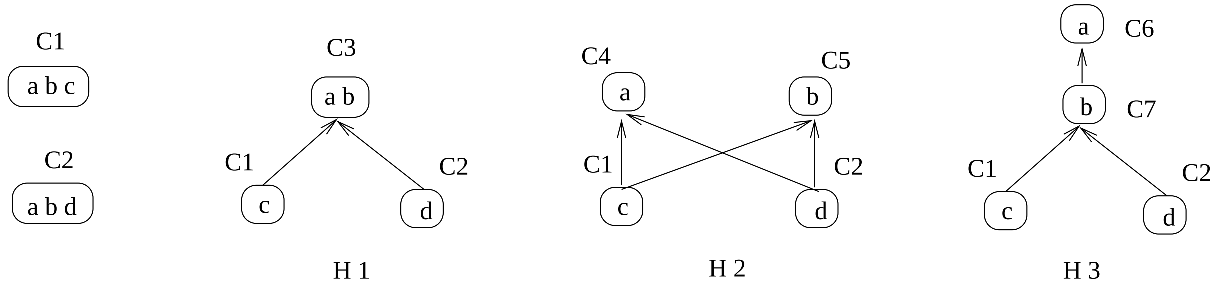

Figure 3: Compactness and maximal factorization

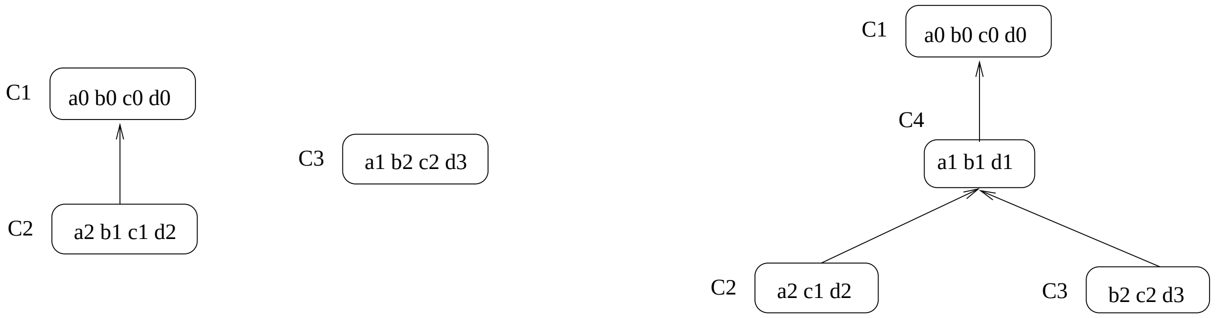

Figure 4: A simple case of overloading

be inserted. C_3's set of properties contains C_2's set of properties, so C_3 is a subclass of C_2. The output hierarchy is H_2.

- the inserted class is not a leaf of the hierarchy. In H_2, the class C_4 is inserted between C_2 and C_3 producing H_3. The declaration of c is transferred from C_3 to C_4.
- a new class is created and factorizes common properties. The next class C_5 is an indirect subclass of C_2. In H_4, class C_6 is created to factorize property d common to C_3 and C_5.
- a class becomes empty. When class C_7 is added, property d is extracted from C_6. The side effect is that C_6 does not declare any more properties in H_5.
- an empty class is removed. The algorithm could be adjusted by deciding whether to keep or delete an empty class. If a deletion policy is chosen, the result of removing C_6 is H_6.

Note that maximal factorization is not always compact. Several maximally factorized hierarchies can be built from the same set of classes.

Consider for example (cf. Fig. 3) a hierarchy built from two classes C_1 and C_2 in which properties a and b have to be factorized.

We may obtain the following different results. Either a and b are grouped together in the same factorization class C_3 (H_1), or a and b are declared in different classes C_4 and C_5 (H_2) (resp. C_6 and C_7 in H_3). All hierarchies are maximally factorized, but H_1 is more compact than the others.

Ares produces compact and maximally factorized hierarchies.

3.2 Handling of overloading in an ideal case

In the presence of overloading, we have divided the problem in two parts. The first problem is to find the lowest common generalization of two occurrences p_1 and p_2 of the same generic property P. The second problem is how to use this generalization in the algorithm, assuming it is available (either computed or given by a human expert). We present here some examples of how *Ares* handles the second subproblem.

In Figure 4, C_3 is to be inserted in the hierarchy made of classes C_1 and C_2; the order for properties is: $a_2 < a_1 < a_0$, $b_2 < b_1 < b_0$, $LCG(c_1, c_2) = c_0$, and $LCG(d_2, d_3) = d_1 < d_0$.

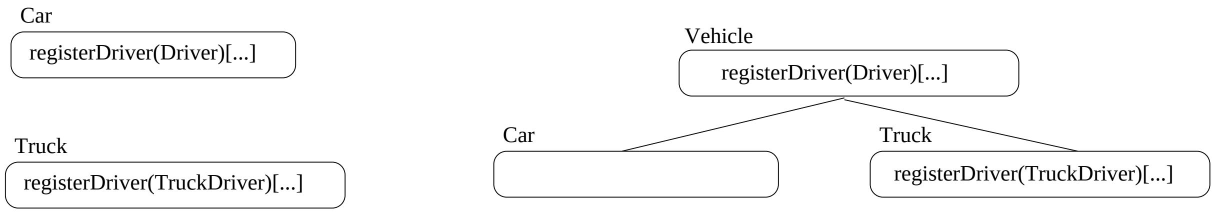

Figure 5: Signatures give the LCG

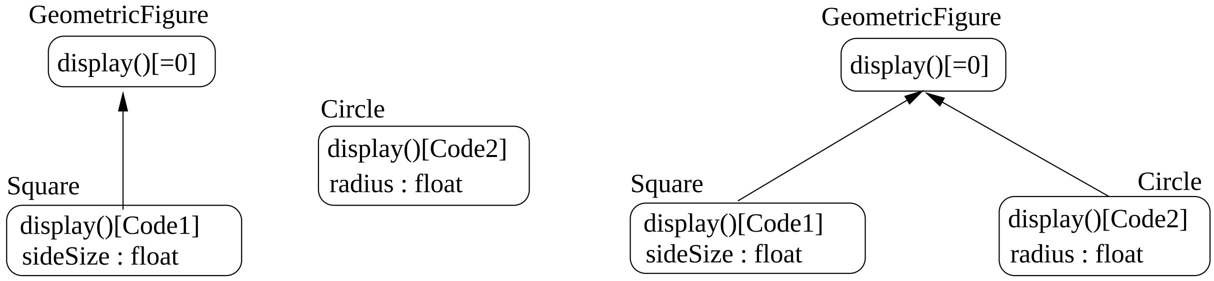

Figure 6: Code gives the LCG

$Ares$ determines that C_3 is a subclass of C_1 simply because each property of C_1 is specialized in C_3. Combining C_2 and C_3 is more complicated, since they are not comparable. For any two occurrences in C_2 and C_3 of a same generic property p, we take the common lowest generalization p_m. If p_m does not appear in the classes above C_2 (here in C_1), we declare p_m in the factorization class C_4.

3.3 Examples of class insertion with overloading and automatic determination of LCG

It is generally impossible to automatically compute the lowest common generalization of two OGP, but it is possible in many situations that we have started studying. The detailed results are presented in section 5.3. We give here some concrete examples of overloading where we know how to compute LCG and how $Ares$ exploits it.

- Signatures give the LCG

The first example (*cf.* Figure 5) comes from [Mey92]. The existing hierarchy is made of a single class Car and the class to be inserted is $Truck$. The two properties to be compared are $p_1 = registerDriver(TruckDriver)$ and $p2 = registerDriver(Driver)$. Given that $TruckDriver < Driver$, we deduce that $p_1 < p_2$ regardless of their bodies and that $LCG(p1, p2) = p2$.

Given this result, $Ares$ knows that p_2 is the method to be stored in the factorization class (which we, not $Ares$, name $Vehicle$) made from Car and $Truck$.

- Code gives the LCG

In the second example (*cf.* Figure 6) two occurrences of the generic property $display$ exist in the hierarchy: $d_0 = display()[= 0]$ and $d_1 = display()[Code1]$ and a new one, ($d_2 = display()[Code2]$) comes with the class Circle to be inserted. Their code being different, d_1 and d_2 can be considered as incomparable. However, another more precise code examination shows that $d_0 = LCG(d_1, d_2)$. These results allow $Ares$ to correctly produce the final hierarchy. It should be noted that the class $GeometricFigure$ is created (except for the name) by the algorithm, if not initially present.

- Using codes and signatures

The last example (*cf.* Figure 7) is taken from Smalltalk-80 [GR83] and adapted to a typed world. Given the class $Date$, inserting the class $Time$

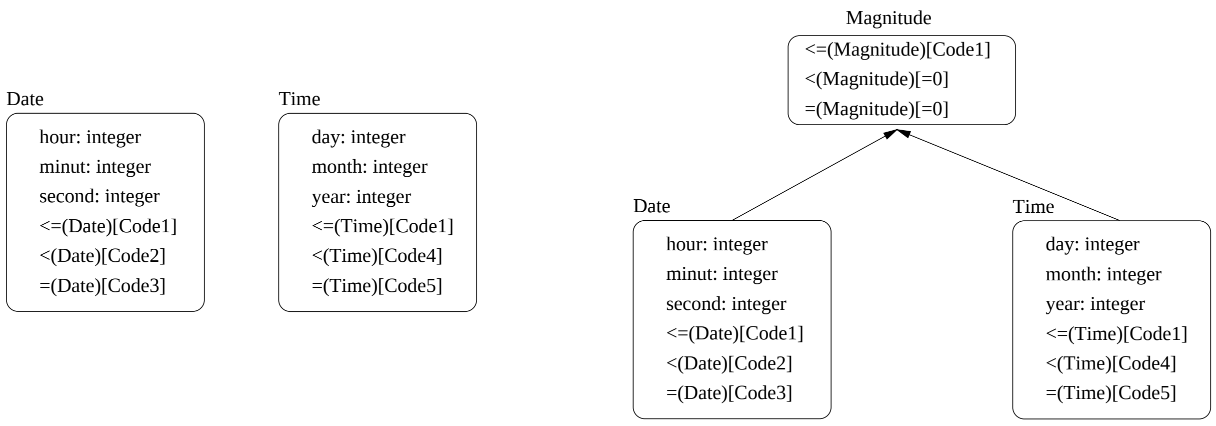

Figure 7: Using codes and signatures

should produce a factorization class ($Magnitude$) with the method $<=$ common to $Date$ and $Time$ and *"subclass responsibility"* versions of methods $<$ and $=$. The occurrences of the generic properties $<=, <, =$ have to be compared and their LCG computed. Let us see how an automatic comparison is possible using the method codes and signatures.

The first issue here is to enrich the language describing our signatures and to recode the signatures of the methods $<=, <, =$ by replacing $Date$ or $Time$ by an **anchored** type as defined in Eiffel [Mey92]. Indeed, the parameter's type of self-referent methods $<=, <, =$ is the type of the method's receiver, thus the possible type of the argument will be determined by the place in the hierarchy where the class will be inserted. In other words, we need to know where the class will be inserted in order to correctly compute LCG and thus to correctly insert it. Thus, to compute LCG, we build new versions of signatures in which anchored types are replaced by the pattern "LC" (which stands for "like current" in reference to Eiffel anchored type declaration).

We then define two signatures that match (in our terminology signatures "potentially equal") as two signatures having at a given position either both the same type, or both the pattern LC.

In the example of Figure 7:

- The two methods $<$ (of $Date$ and $Time$) now have signatures that match, and have different bodies. This is enough to confirm that they are incompatible and that their LCG will be a method having an empty body, defined in the common superclass —say $Magnitude$— of $Date$ and $Time$, and of signature ($Magnitude$).
- The two methods $<=$ (of $Date$ and $Time$) have signatures that match and the same code p. This is enough to confirm that they can be factored via a LCG, which is a method of code p defined on the same common superclass —$Magnitude$ —of $Date$ and $Time$ and of signature ($Magnitude$).

This example represents a typical situation showing how $Ares$ is able to highlight interesting factorization classes.

4 Comparison with related works

Related works [GM93, Cas92, LBSL91, LBSL90, Ber91, MS89, Run92, DDHL94a, DDHL95, Moo95, MC96] may be studied from three viewpoints: the strategy used to reorganize hierarchies, the features of the hierarchy and the handling of overloading.

4.1 Global, incremental and toolbox strategies

To build a hierarchy, different strategies can be considered:

- The *Toolbox* approach, proposed by [Ber91], is based on a set of local operations allowing users to modify a hierarchy.

- *Global* algorithms [MGG90, LBSL91, LBSL90, Cas91, Moo95, MC96] build in a single step the whole hierarchy from the binary relation $Class - property$.
- *Incremental* algorithms insert a new class into an already existing hierarchy. Such a technique is proposed by [Cas92, MS89, Run92, DDHL94a, DDHL95, GMM95].

All of these strategies may lead to the same results, no one can be considered better than another. For instance, given a set of classes, a whole hierarchy can be built by successive applications of an incremental algorithm. Conversely, a global algorithm can always be used to insert a class A in a hierarchy whose class set is $\mathcal{E}$ —starting from A and $\mathcal{E}$, forgetting the structure of the hierarchy. However, depending on the utilization context, one strategy or another will be more suitably adapted. A global algorithm is obviously more suitable in the first case above, while an incremental algorithm should be used in the second. Besides, global algorithms are more adapted when the given data is the relation *Class-Property* —for instance, when reorganizing an unsatisfactory hierarchy from scratch—, while incremental algorithms and toolboxes fit evolution better.

4.2 Underlying hierarchy models

The underlying model used to represent hierarchies is more or less restrictive.

[Cas92] does not impose any constraints on the inheritance graph; this seems powerful at first sight but there is no formal characterization of the results produced by the algorithm. In [LBSL91, LBSL90], there is a strong constraint on hierarchies in which only leaves can represent instanciables classes.

Moreover, a second set of algorithms use implicitly ([Run92, MS89, MC96]), or explicitly ([GM93, GMMM95, DDHL94a, DDHL95]) with further adaptations, the Galois lattice of the *Class-Property* relation to encode hierarchies. The Galois lattice is a mathematical construction ensuring the most compact maximal factorization (more details can be found in [Aig79, Wil89, Wil92, GM93, DDHL94a, GMMM95]).

[Run92, MS89] use the whole lattice (precisely a sup-semi-lattice) and this raises some problems. Firstly because of space consumption: in the worst case, the space complexity is exponential in $max(number\ of\ classes, number\ of\ properties)$. Secondly, because this structure imposes some constraints on the hierarchy; for example, Figure 2 shows how this structure can forbid the deletion of a class (class C_6 in H_5); indeed, if the deletion is achieved (as in H_6) then C_3 and C_5 have the two lowest common superclasses C_2 and C_8 and the hierarchy is no longer a lattice.

More cleverly, [GM93] proposed use of a structure which is a sub-order of the Galois lattice, that we call a Galois subhierarchy, which improves space complexity. However, in the same example of Figure 2, the Galois subhierarchy imposes a contrary constraint to the hierarchy: class C_6 *must* be deleted even if it is a *meaningful* class (cf. section 2.3). [MC96] seems to produce the same structure of a Galois subhierarchy.

Our algorithm is based on the Galois subhierarchy. It uses and preserves an underlying Galois lattice and thus produces formally well characterized results. We have added a slight modification to avoid the deletion of meaningful classes.

4.3 Taking overloading into account

Initial studies [LBSL90, LBSL91] did not take overloading into account. A first advance is described in [Cas92], which simply allows an abstract (pure virtual) method to be overriden by an implemented one which itself cannot be overriden. A second step is described in [MS89, Run92, DDHL95, GMMM95, GMM95], proposing systems able to take overloading into account, provided there is an "oracle" able to compare two occurrences of the same generic property, and give their lowest common generalization(s).

We improve algorithms concerning this topic in two ways: the main one concerns partial automation of the oracle for a set of well-defined cases, including cases of "self-referent" signatures, the second one is a significant improvement of the space complexity.

5 The *Ares* Algorithm

We present below the specifications, the algorithm, and then detail our approach of overloading with self-referent signatures.

5.1 Specifications

Input: The algorithm starts with $H_i = (\mathcal{C}_i, \Gamma_i)$ a class hierarchy with a root [5] and with a meaningful class A to be inserted. $\mathcal{C}_{Mean}$ is the set of meaningful classes of H_i. $Properties(A)$ is A's property set.

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

- **Preservation of the maximal factorization of properties**
  When H_i is maximally factorized, so is H_f
- **Preservation of the underlying model**
  When H_i is a Galois sub-hierarchy of $\mathcal{C}_{Mean}$, H_f is a Galois sub-hierarchy of $\mathcal{C}_{Mean} \bigcup \{A\}$.
- **Inheritance path preservation of the hierarchy**
  For all classes of H_i still belonging to H_f, the inheritance paths remain.
- **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 $\mathcal{C}_{Mean} \bigcup \{A\}$

Note that with any input hierarchy, *i. e.* not necessarily a Galois subhierarchy and/or a maximally factorized one, everything common to the hierarchy and the class to be inserted is factorized by *Ares*.

5.2 The algorithm

The algorithm (*cf.* Figure 8) can be split into three parts : (1) search of A superclasses, (2) deletion of non-meaningful empty classes and (3) search of A subclasses. We focus on part 1, in which A superclasses are found or built and where A is bounded to its immediate superclasses. This part raises the main issues. A complete description of parts 2 and 3 could be adapted from previous work [DDHL94b].

The algorithm uses some global variables:

- $AalreadyCreated$: a Boolean which is true if, while visiting the hierarchy and creating factorization classes, a class with the same properties as A is found.
- SH: the current set of A superclasses. At any time, SH holds already visited classes in H_i which are A superclasses, as well as the factorizing classes (obviously A superclasses) already created by the algorithm.
- $EmptyClasses$: the set of the non-meaningful classes which, after a factorization, do not declare any more properties.

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

The algorithm visits all classes in the input hierarchy H_i going down from the root and following a linear extension LEH_i of $>_{H_i}$, *i.e.* a class is visited after all its superclasses. The goal of these visits is to build SH, the set of A superclasses in H_f. Then, if needed, A is created and bounded to its direct superclasses.

When a class C is visited, its set of declared properties $Declared(C)$ is compared to the set $Properties(A)$. We compute $ExtractedProperties(C, A)$ which can be seen as the set of the properties "common" to C and A. For any pair of occurrences (p_C, p_A) of the same generic property, we call a function which in first approximation returns $LCG(p_C, p_A)$ —for further details, see section 5.3. We keep $LCG(p_C, p_A)$ if it is not declared in a (strict) superclass of C. More formally, the set $ExtractedProperties(C, A)$ is:
$ExtractedProperties(C, A) =$
$\{p_m = LCG(p_C, p_A)\ s.t.\ \exists P,\ p_C \in P,\ p_A \in P,\ p_C \in Declared(C),\ p_A \in Properties(A),$
$and\ p_m\ is\ not\ declared\ in\ a\ strict\ superclass\ of\ C\}$

The set of remaining properties of C (resp. A) is:
$Remainder(C) = Declared(C) \backslash$

[5] The root has no properties, and does not appear in the figures

$ExtractedProperties(C, A)$
$Remainder(A) = Properties(A) \backslash$
$[ExtractedProperties(C, A) \cup Inherited(C)])$.

The operator $\backslash$ is in first approximation the set difference, and will be better specified in Section 5.3.

Now, when $ExtractedProperties(C, A)$ is empty, nothing has to be factorized, and in the other cases:

- $Remainder(C)$ is not empty

  - $Remainder(A)$ is not empty
  Example: $A = C_5$ and $C = C_3$ for H_3 in Figure 2. $ExtractedProperties(C, A) = \{d\}$, $Remainder(C) = \{e\}$ and $Remainder(A) = \{f\}$. The two classes are incomparable and the properties of $ExtractedProperties(C, A)$ are factorized[6] in a common superclass C' of C and A. C' is added in the hierarchy and is stored in SH (the set of A superclasses). C' is defined as a superclass of C, and as a subclass of the classes of $Sups(C, SH)$. We call $Sups(C, SH)$ the minimal elements of the set of classes which are C superclasses while belonging to SH.

  - $Remainder(A)$ is empty
  Example: $A = C_4$ and $C = C_3$ for H_2 in Figure 2. $ExtractedProperties(C, A) = \{c\}$, $Remainder(C) = \{d, e\}$ and $Remainder(A) = \{\}$. A is a superclass of C, the properties of A declared in C are extracted from C and we insert A in the hierarchy as a superclass of C.

- $Remainder(C)$ is empty

  - $Remainder(A)$ is not empty
  Example: $A = C_3$ and $C = C_2$ for Hierarchy 1 in Figure 2. $ExtractedProperties(C, A) = \{b\}$, $Remainder(C) = \{\}$ and $Remainder(A) = \{c, d, e\}$. C is a superclass of A, C is stored in SH.

  - $Remainder(A)$ is empty
  It means that A and C are the same class.

When the whole hierarchy has been visited, SH contains all A superclasses. If A has not already been found, it must be created and connected to its immediate superclasses — we use the function $Min(E)$, which returns the minimal (for $<_H$) classes of set E.

[6] and possibly adapted as shown later

5.3 Automatic comparison of occurrences of generic properties

Handling overloading in *Ares* requires being able to compare an occurrence of a generic property (coming with the class to be inserted) with elements already present in the hierarchy.

A problem arises when the set of properties of the class A to be inserted and the set of properties of an existing class C being visited (cf. section 5.2) contain p_A and p_C respectively, two OGP of the same generic property P.

In such cases, *Ares* needs to compute:

- $LCG(p_A, p_C)$ *i. e.* the property to be factorized in a common superclass of C and A. $LCG(p_A, p_C)$ can be either p_A or p_C or the lowest property that both p_A and p_C specialize.
- $Remainder(C)$ and $Remainder(A)$, allowing *Ares* to state what is the common superclass of A and C[7].

In the above section, describing the algorithm, discussions related to (1) the computation of the LCG of two OGP and (2) the precise description of the computation of $Remainder(C)$ and $Remainder(A)$ have been delegated and are presented here.

We first present additional information and definitions of OGP and their signatures necessary for the comparison of properties. Secondly, we explain how to compute remainders of C and A in the general case, and thirdly we deal with automatic computation of $LCG(p_a, p_c)$.

[7] note that with our working hypothesis, A and C owning an occurrence of the same generic property will have a common superclass, that can be either A or C or a factorization class

```
Algorithm ARES(H_i, A)
  begin
    // Initializations
    AalreadyCreated ← false
    SH ← ∅
    EmptyClasses ← ∅
    // Looking for and binding superclasses
    For every vertex C following LEH_i do
        //  LEH_i is an arbitrary linear extension of H_i starting from root Ω
        // Visiting C
        if ExtractedProperties(C, A) ≠ ∅ then
              if Remainder(C) ≠ ∅ then
                     Create(C')
                     Declared(C') ← ExtractedProperties(C, A)
                     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
                     //C'=A is a superclass of C
                            SetMeaningful(C')
                            AalreadyCreated ← true
                     else SH ← SH ⋃ {C'} endif
                     if Declared(C) = ∅ and not IsMeaningful(C) then
                            EmptyClasses ← EmptyClasses ⋃ {C} endif
              else //Remainder(C) = ∅
                     if Remainder(A) ≠ ∅ then
                            //C is a superclass of A
                            SH ← SH ⋃ {C}
                     else //Remainder(A) = ∅ :  C and A are the same class
                            AalreadyCreated ← true endif
              endif
        endif
    endfor    // Creating class A and binding it to SH
    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
    DeleteEmptyClasses
    BindSubClasses
  end
```

Figure 8: **The ARES Algorithm**

5.3.1 Keys for property comparisons

Methods are compared by mixing code comparison and signature comparison. Instance variables are compared using their types.

Code comparison. At this stage of the work, two cases have been considered, the codes of the methods to be compared are either identical or different.

Signature comparison. Signature comparison is based on type comparison. Two types T_1 and T_2 are either equal, or one is a subtype of the other, or they are incomparable and thus have a common supertype $sup(T1, T2)$.

Moreover, as explained in the examples (cf. Section 3.3), we need to separately consider self-referent signatures, *i. e.* signatures including the class in which the property is defined. Such a class has been characterized in the signature as an anchored type and recoded with the pattern "LC".

Comparison relationships for signatures. Definitions of some comparison relationships between signatures used in the algorithm are presented. Let us consider two signatures $S_A = (A_1, A_2, ..., A_n)$ and $S_C = (B_1, B_2, ..., B_n)$, where A_i and B_i are known types.

- S_A and S_C are **equal** if $\forall i\ (Ai = Bi)$
- S_A and S_C are **potentially equal** if $\forall i\ (Ai = Bi)$ or $(Ai = LC$ and $Bi = LC)$
- S_A and S_C are **comparable** if one is a specialization of the other, for example $S_A < S_C$ if $\forall i\ (A_i \leq B_i)$,
- S_A and S_C are **potentially comparable** if one is a potential specialization of the other. For instance S_A is a potential specialization of S_C if $\forall i\ (Ai <= Bi)$ or $(Ai = LC$ and $Bi = LC)$,
- S_A and S_C are **incomparable**, if ($\exists i$ s.t. A_i and B_i are incomparable) or ($\exists i, j$ s.t. $A_i < B_i$ and $B_j < A_j$).

5.3.2 *Remainder* computation

Let us define **equal** (resp. **potentially equal**) properties as properties with the same code and equal (resp. potentially equal) signatures.

It is now possible to more precisely compute $Remainder(C)$ and $Remainder(A)$.

- $Remainder(C)$ is obtained by removing, from $Declared(C)$, properties equal or potentially equal to a property of $ExtractedProperties(C, A)$.

- $Remainder(A)$ is obtained by removing, from $Properties(A)$, properties equal or potentially equal to a property of $ExtractedProperties(C, A)$.

5.3.3 Computing LCG of two properties

We deal in this section with the issue of computer-aided determination of $LCG(p_A, p_C)$ in the working context defined in section 2. Recall that we generally distinguish between three kinds of cases in the determination of $LCG(p_A, p_C)$:

- Cases where such a determination requires a human expert, for example when comparing two methods with different codes doing the same thing.
- Cases in which an automatic computation is possible that we do not yet handle. For example, it is possible to perform much more clever code comparisons than those we have already done.
- Cases that we have studied and that we now describe. We consider that the rather simple rules that we have established allow *Ares* to deal with numerous and nontrivial cases.

The cases we have considered are given by mixing code and signature comparisons as summarized in Figure 9.

Let us consider again two OGP: p_A with signature S_A in the class A to be inserted, and p_C with signature S_C declared in the class C that *Ares* is visiting. For each case in the array, we give the LCG and when needed explanations and examples.

1. p_A and p_C have the same signature and the same code : p_A and p_C are the same property, $LCG(p_A, p_C) = p_A = p_C$.

2. p_A **and** p_C **have the same code, and their signatures are *potentially equal*** : both signatures have at least one anchored type at the same position.

For instance, if p_A is $p_A(T_1, ..., T_i, A, ..., T_n)[code1]$, and p_C is $p_C(T_1, ..., T_i, C, ..., T_n)[code1]$, then $LCG(p_A, p_C) = p_m = p(T_1, ..., T_i, sup(A, C), ..., T_n)[code1]$, where $sup(A, C)$ is the lowest common superclass [8] of C and A in which the algorithm will store p_m, if p_m is not already "declared" in a superclass of C *i. e.* if there is no superclass X of C containing $p_X(T_1, ..., T_i, X, ..., T_n)[code1]$.

An example of such a situation can be found in the *Magnitude* example (cf. Figure 7), where A is $Time$, C is $Date$, and the considered property is $\leq$. The following provides a snapshot of things computed by the algorithm:
$LCG(\leq (LC)[code1], \leq (LC)[code1])$
$= \leq (sup(Time, Date))[code1])$
$ExtractedProperties(Time, Date)$
$= \{\leq (sup(Time, Date))[code1], ...\}$
$Remainder(Date)$
$= \{hour : integer, minut : integer, second : integer, ...\}$
$Remainder(Time)$
$= \{day : integer, month : integer, year : integer, ...\}$
Knowing that neither $Remainder(Date)$ nor $Remainder(Time)$ are empty, $Ares$ deduces that a factorization class $C' = Magnitude$[9] has to be created. In $Magnitude$ properties stored in $ExtractedProperties(Time, Date)$ will be declared, in particular $\leq (Magnitude)[code1]$. Note that the signature for $\leq$ on $Magnitude$ has been rebuilt.

3. p_A **and** p_C **have the same code, and their signatures are comparable**. In whole generality, one of the properties is a specialization of the other, if for example $S_A < S_C$, then $LCG(p_A, p_C) = p_C$.

[8] superclass in a broad sense, which can be C or A

[9] We will use the name Magnitude for clarity but of course, *Ares* does not find the name

In the "car-truck" example (*cf.* Section 3.3, Figure 5), no hypothesis have been put forward concerning the code of the two properties *registerDriver*. If we consider that they have the same code, this is an example of our current case 3, and we compute:
$LCG(registerDriver(Driver)[code1],$
$registerDriver(TruckDriver)[code1])$
$= registerDriver(Driver)[code1] = p_m$

p_m will be declared on the superclass of the two classes Car and $Truck$, whatever it is. Knowing whether or not the other property (here $registerDriver(TruckDriver)[code1]$) should be considered the same and subsequently be removed from the other class, is an optimization of the algorithm and is language and application dependent.

4. p_A **and** p_C **have the same code, and their signatures are potentially comparable** : both signatures have at least one anchored type at the same position.

For instance, if
p_A is $p_A(T_1, ..., T_i, X, ..., T_j, A, ..., T_n)[code1]$,
p_C is $p_C(T_1, ..., T_i, Y, ..., T_j, C, ..., T_n)[code1]$,
with $Y < X$. Then $LCG(p_A, p_C)$
$= p(T_1, ..., T_i, X, ..., T_j, sup(A, C), ..., T_n)[code1]$.
This case is very similar to Case 2, but $sup(A, C)$ is more constrained, it cannot be C.

5. p_A **and** p_C **have the same code, and their signatures are incomparable.**

For instance, if
p_A is $p_A(T_1, ..., T_i, ..., T_n)[code1]$, and
p_C is $p_C(T'_1, ..., T'_i, ..., T'_n)[code1]$, then
$LCG(p_A, p_C)$ is
$p(sup(T_1, T'_1), ..., sup(T_i, T'_i), ..., sup(T_n, T'_n))[code1]$.

6. p_A **and** p_C **have the same signatures, and their codes are different.**

If codes are different, at least a deferred property can be declared for a superclass. For example, if
p_A is of the form: $p_A(T_1, ..., T_n)[code1]$, and
p_C is of the form: $p_C(T_1, ..., T_n)[code2]$, then
$LCG(p_A, p_C) = p(T_1, ..., T_n)[= 0]$

Signatures \ Codes	Equal	Potentially equal	Comparable	Potentially comparable	Incomparable
Equal	1	2	3	4	5
Different	6	7	8	9	10

Figure 9: Mixing code and signature comparison

This situation was encountered in the second example of section 3.3, (cf. Figure 6) when comparing the methods $display$ of class $Square$ and $display$ of class $Circle$. The LCG to factorize is $display()[= 0]$. Since this property is already declared in the hierarchy, $Ares$ correctly inserts the class $Circle$ as a subclass of $GeometricFigure$.

This formula for LCG is again an acceptable result, but p_A could also be a specialization of p_C (or the opposite). Determining this requires either a human expert or more sophisticated techniques for code comparison (is p_a's code a specialization of p_c's code?) or an optimization of the $Ares$ result on which we are currently working.

7. p_A and p_C have different codes, and their signatures are potentially equal.

For instance, if
p_A is $p_A(T_1, ..., T_i, A, ..., T_n)[code1]$, and
p_C is $p_C(T_1, ..., T_i, C, ..., T_n)[code2]$, then
$LCG(p_A, p_C) = p(T_1, ..., T_i, sup(A, C), ..., T_n)[= 0]$

The Magnitude hierarchy (*cf.* Section 3.3, Figure 7) includes an example of such a case, where A is $Time$, C is $Date$, and the considered properties are $<$ of $Date$ and $Time$. The computed LCG to be stored in the factorization class is $< (sup(Date, Time))[= 0]$. This factorization class being determined (*cf.* the discussion on Case 2), the final property to factorize is $< (Magnitude)[= 0]$

8. p_A and p_C have different codes, and their signatures are comparable.

One of the properties is a specialization of the other, if for instance $S_A < S_C$, then $LCG(p_A, p_C) = p_C$.

This case occurs in the "car-truck" example (cf. Figure 5) if we consider that the two methods $registerDriver$ have different codes. The computed LCG is $registerDriver(Driver)[code1]$ that will be declared in the common superclass of Car and $Truck$. The difference with Case 3 is that here $registerDriver(TruckDriver)$ clearly overrides $registerDriver(Driver)$.

9. p_A and p_C have different codes, and their signatures are potentially comparable —both signatures have at least one anchored type at the same position.

For instance, if
p_A is $p_A(T_1, ..., T_i, X, ..., T_j, A, ..., T_n)[code1]$, and
p_C is $p_C(T_1, ..., T_i, Y, ..., T_j, C, ..., T_n)[code2]$,
with $Y < X$, then $LCG(p_A, p_C)$
$= p(T_1, ..., T_i, X, ..., T_j, sup(A, C), ..., T_n)[= 0]$.

10. p_A and p_C have different codes, and their signatures are incomparable.

For instance, if
p_A is $p_A(T_1, ..., T_i, ..., T_n)[code1]$, and

$p_C(T'_1, ..., T'_i, ..., T'_n)[code2]$, then $LCG(p_A, p_C) = p(sup(T_1, T'_1), ..., sup(T_i, T'_i), ..., sup(T_n, T'_n))[=0]$.

This is a case where further researches are necessary, indeed such a rule may lead, in certain cases, to the creation of uninteresting (only containing deferred[10] properties) factorization classes. The issues here are (1) how to obtain a more precise rule and (2) how to optimize the hierarchy thereafter.

6 Conclusion

We have presented an incremental algorithm able to automatically insert a class, defined by the set of its properties, into an existing class inheritance hierarchy. The algorithm takes an input hierarchy and a class and produces a well characterized output hierarchy: it preserves the input hierarchy features such as its structure, maximal factorization of properties, inheritance paths and the set of meaningful classes.

Furthermore, handling of overloading in the algorithm has been studied and partially achieved. The problem has been split into two subproblems: (1) the comparison of occurrences of generic properties and (2) the use of the results of these comparisons in the algorithm. Provided that the first subproblem is solved, the algorithm works with overloading according to the above descriptions. Concerning the first subproblem, we have recalled the limits of automatization, *i. e.* we explained why it will never be able to completely deal with the comparison of generic properties without the assistance of a human expert. These limits being defined, we have given a first categorization of properties and some rules to compare them automatically in a certain number of well defined cases, notably in self-referent signatures cases.

The algorithm has been implemented and tested on nontrivial but pre-compiled cases. One of our main current concerns is to apply it to large scale hierarchies produced in foreign applications. This requires interfacing the algorithm, and secondly implementing post-processors that will optimize its results — many optimizations are possible but there was no room to describe them there.

Many further studies can be foreseen: We first plan to extend the number of handled cases of automatic comparison of generic properties; this is possible: (1) by studying in further detail the cases of properties having self-referent signatures, and (2) by analyzing more precisely method bodies via syntactic and even semantic analysis. Concerning method refactoring through syntactic analysis, the reader should refer to [OJ93] and [Moo96]. Another difficult issue would be to combine this work with linearization algorithms [DHHM94] used to solve conflicts in hierarchies with multiple inheritance.

Ackowledgments
We would like to thank Nicolas Prade for its contribution to the automatic comparison of occurrences of generic properties.

References

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

[Ber91] P. Bergstein. Object Preserving Class Transformations. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'91*, 26(11):299–313, 1991.

[Cas91] E. Casais. *Managing Evolution in Object Oriented Environments : An Algorithmic Approach.* PhD thesis, Université de Genève, 1991.

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

[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*, pages 125–136, 1994.

[DDHL94b] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, un algorithme

[10] subclass responsibility

d'Ajout avec REStructuration dans les hiérarchies de classes. Technical report, LIRMM, 1994.

[DDHL95] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, Adding a class and REStructuring Inheritance Hierarchies. *11 ièmes journées Bases de Données Avancées, Nancy*, 1995.

[DHHM94] R. Ducournau, M. Habib, M. Huchard, and ML. Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'94*, 29(10):164–175, 1994.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92.*, 27(10):201–217, 1992.

[GM93] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10):394–410, 1993.

[GMM95] R. Godin, G. Mineau, and R. Missaoui. Incremental structuring of knowledge bases. *Proceedings of International KRUSE symposium: Knowledge Retrieval, Use, and Storage for Efficiency Springer-Verlag's Lecture Notes in Artificial Intelligence*, 9(2):179–198, 1995.

[GMMM95] R. Godin, H. Mili, G. Mineau, and R. Missaoui. Conceptual Clustering methods based on Galois lattices and applications. *Revue d'intelligence artificielle*, 9(2), 1995.

[GR83] A. Golberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison Wesley, Reading, Massachusetts, 1983.

[JF88] Ralph E. Johnson and Brian Foot. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[LBSL90] K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe. Abstraction of object-oriented data models. *Proceedings of International Conference on Entity-Relationship*, pages 81–94, 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*, pages 205–228, 1991.

[MC96] Ivan Moore and Tim Clement. A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies. *TOOLS Europe 1996 Proceedings, Prentice-Hall*, 1996.

[Mey92] B. Meyer. *Eiffel, The Language.* Prentice Hall - Object-Oriented Series, 1992.

[MGG90] Guy Mineau, Jan Gecsei, and Robert Godin. Structuring Knowledge Bases Using Automatic Learning. *Proceedings of the sixth International Conference on Data Engineering*, pages 274–280, 1990.

[Moo95] Ivan Moore. Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies. *TOOLS USA 1995 Proceedings, Prentice-Hall*, 1995.

[Moo96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 1996.

[MS89] M. Missikoff and M. Scholl. An Algorithm for Insertion into a Lattice: Application to Type Classification. *Proc.*

3rd Int. Conf. FODD'89, pages 64–82, 1989.

[OH92] Harold Ossher and William Harrison. Combination of Inheritance Hierarchies. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92*, 27(10):25–40, 1992.

[OJ93] William F. Opdyke and Ralph E. Jonhson. Creating Abstract Superclasses by Refactoring. *Proceedings of the 21st Annual Conference on Computer Science*, pages 66–72, February 1993.

[Run92] E. A. Rundensteiner. A Class Classification Algorithm For Supporting Consistent Object Views. Technical report, University of Michigan, 1992.

[Wil89] R. Wille. Knowledge acquisition by methods of formal concept analysis. *Data Analysis, Learning Symbolic nd Numeric Knowledge*, 23:365–380, 1989.

[Wil92] R. Wille. Concept lattices and conceptual knowledge systems. *Computers Math. Applic*, 23:493–513, 1992.