Automatic hierarchies reorganization: an algorithm and case studies with overloading*

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: dony,huchard,libourel@lirmm.fr

Abstract

The automatic construction or reorganization of classes inheritance hierarchies is one of the facets of the management of object-oriented software evolution that has received a lot of attention in the last years.

Several algorithms have been proposed to automatically insert a class into an inheritance hierarchy or to reorganize them. One of the major remaining issue for all these algorithms is a complete handling of overloaded and overriden properties.

In this paper, we describe a new version of our algorithm (named *Ares*) which makes new advances on the automatic handling of overlaoded properties. Besides, the algorithm has the following 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. We also present numerous examples and two case-studies highlighting both the possibilities and the current limits of the algorithm.

Key words : Class management and class evolution, Reuse, Inheritance hierarchies, Overloading, Galois lattice, Class libraries, Restructuring

1 Introduction

The automatic construction or reorganization of classes inheritance hierarchies is one the facets of the management of object-oriented software evolution that has received a lot of attention in the last years. Automatic reorganization takes its place in the process of re-engineering of object-oriented systems initially build without concern for generalization, and of large and long-lived applications. It can bring to the fore factorization and abstract classes [OJ93] or help to merge hierarchies representing different applications[1]. The more the classes to be structured multiply and become intricate, the more the structuring process can benefit from automation. This paper deals with automation 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. 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. Before going further, it should be stated that it is certainly impossible to find a general algorithm that could completely automate, class insertion and/or hierarchy reorganization; firstly because there exists no criteria defining what is a "good" hierarchy independently of a context, and secondly, because the construction rules are often very empirical.

A lot of different works describe algorithms for automatic class insertion or hierarchy reorganization [MS89, LBSL90, Ber91, LBSL91, Cas92, Run92, GM93, Cas94, DDHL94a, Cas95, DDHL95, Moo95, CL96, DDHL96, Moo96]. Many of them focus on the most tangible criteria used when organizing hierarchies: to point out common properties and create classes to store them (*i.e.* "factor common properties"). Beyond that common criterion, there is room for multiple variations: global handling of a hierarchy or incremental classes insertions, maximal or partial factorization, absence or presence of conditions on inputs and outputs, taking into account or not the characteristics of classes properties (signatures, codes, ...). A common characteristic of object-oriented programs, knowledge representation and database hierarchies is that they include properties whose name's are

*Version longue de [DDHL96].

[1] Merging using automatic class insertion should not be confused with hierarchy combination, as proposed in [OH92].

overloaded. Advances have been made in handling this issue[Cas94, DDHL95, GMM95, DDHL96], which requires that classes properties be compared using their codes and signatures. A better handling of that last subproblem remains the most important barrier to the full reorganization of actual hierarchies.

This paper deals with property comparison, automatic class insertion and hierarchy reorganization in presence of overloading. It completes, clarifies and extends the description of the *Ares* algorithm given in [DDHL96]. Section 2 defines our terminology. Section 3 presents some commented examples of algorithm inputs-outputs that highlight its main properties and give an idea on how overloading is handled. Section 4 focuses on the classes comparison and properties comparison subtasks of the algorithm. Section 5 gives a detailed description of the algorithm. Section 6 proposes two case studies in which the algorithm has been applied to actual hierarchies. Section 7 presents the algorithm's underlying theoretical model. Section 8 makes a synthesis of related works and compares our results to those of existing equivalent systems. Section 9 concludes this paper with some perspectives. Appendix A summarizes the 10 possible cases of properties comparison that *Ares* is able to handle.

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. We denote this partial order by $<_H$: for two classes C_1 and C_2, $C_2 <_H C_1$ stands for C_2 is a subclass of C_1. 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 is 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.

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[2]. The conformance rule for signature redefinition is one point to be specified when the algorithm is to be applied. We have adopted an Eiffel-like covariance policy [Mey92] for variable and method redefinitions.

We call *a generic property* the set of all properties having the same name and same arity (in case of methods)[3]. Each property belongs to a generic property, *i.e.* is an element, or an *occurrence* of the set of properties having the same name[4]. 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, of two distinct occurrences of P.

[2] 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)

[3] 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.

[4] *OGP* stands for *Occurrence of a Generic Property.*

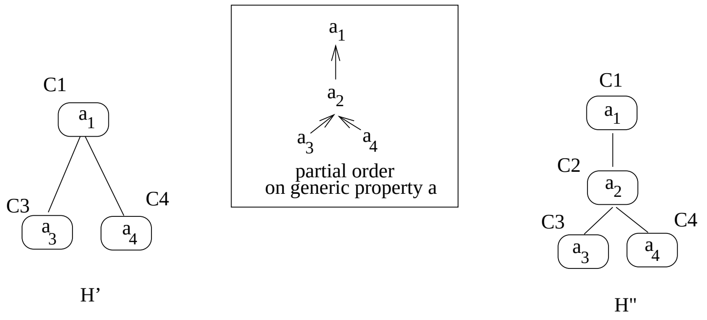


Figure 1: H' is not maximally factorized, H" is.

The different occurrences of P are ordered by a "specialization" order (denoted by $<$). For instance variables, this specialization order can be deduced from the specialization order on their types, *i. e* under our hypotheses from $<_H$.
For methods, this specialization order can be deduced from a specialization order on the signatures and then on a specialization order on method bodies[5]. A ticklish problem arises when we admit "self-reference" in signatures[6]. "$p(C_1, C_2) : C_3[code]$" 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 note: p_0 or $p()[=0]$ a *subclass responsibility* or *pure virtual* method with an empty code. Such a method is automatically the top of the specialization order of P.
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.

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_i and C_j with two properties p_i and p_j respectively, the hierarchy always includes a common superclass of C_i and C_j which is the only class that declares $LCG(p_i, p_j)$ (*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 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.

[5] A method that performs a *super* call could be considered as a specialization of the method invoked by this call.
[6] A signature is "self-referent" when it contains the type of the method's receiver.

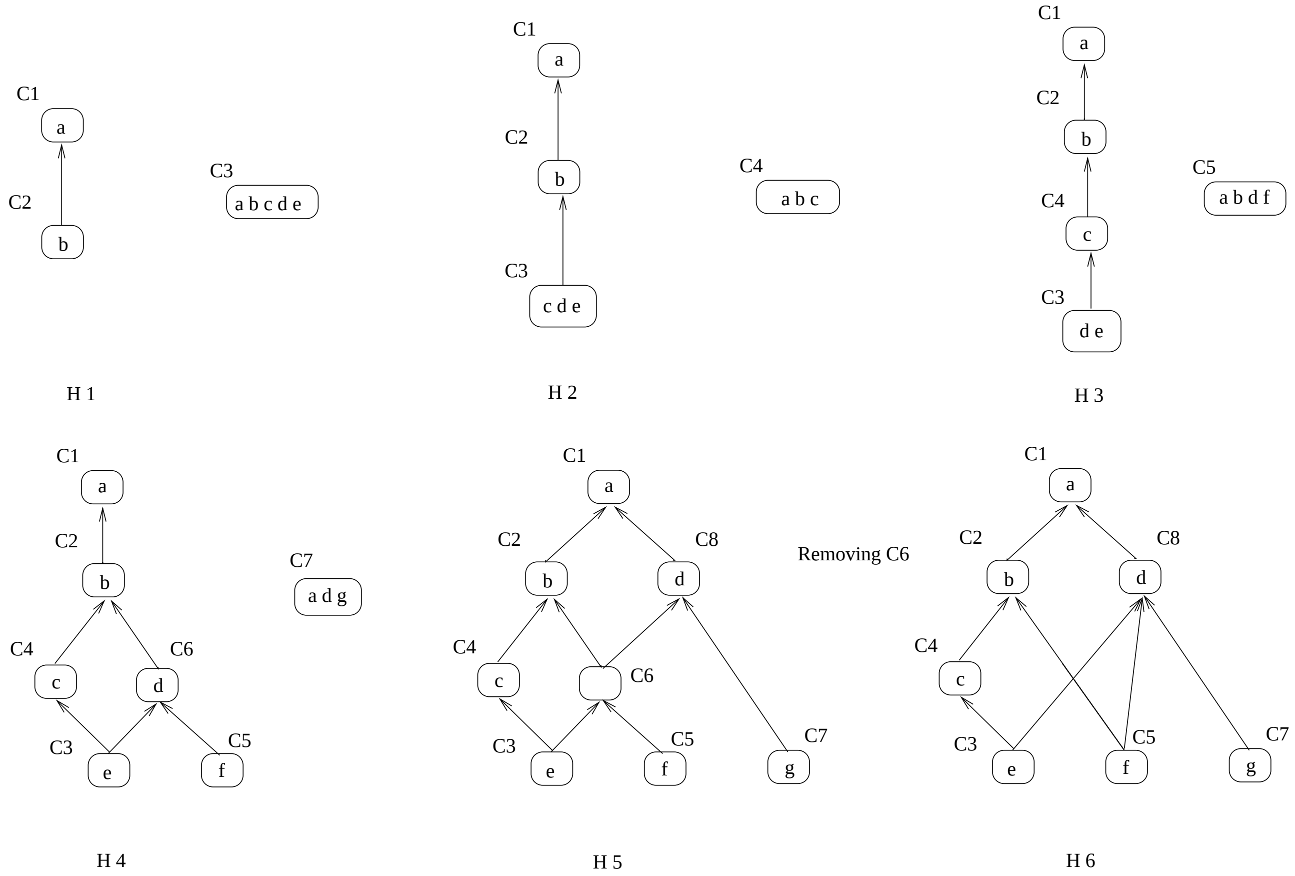


Figure 2: Insertions without overloading

- 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 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 Example with overloading

In the presence of overloading, the problem can be split in two parts. The first issue is to find the lowest common generalization of two occurrences p_1 and p_2 of the same generic property P. The second issue is to use correctly this generalization in the algorithm, assuming it is available (either computed or given by a human expert). We present here an example of how *Ares* handles the second subproblem, in an ideal case where the LCG are known.

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 $LCG(d_2, d_3) = d_1$ with $d_1 < d_0$.

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.

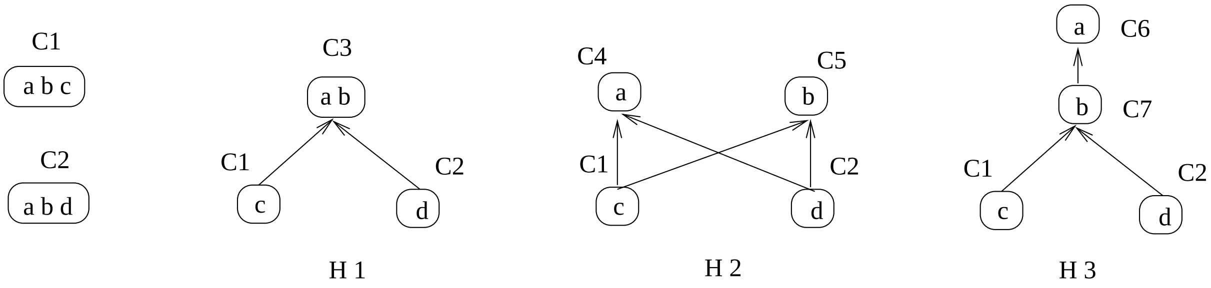


Figure 3: Compactness and maximal factorization

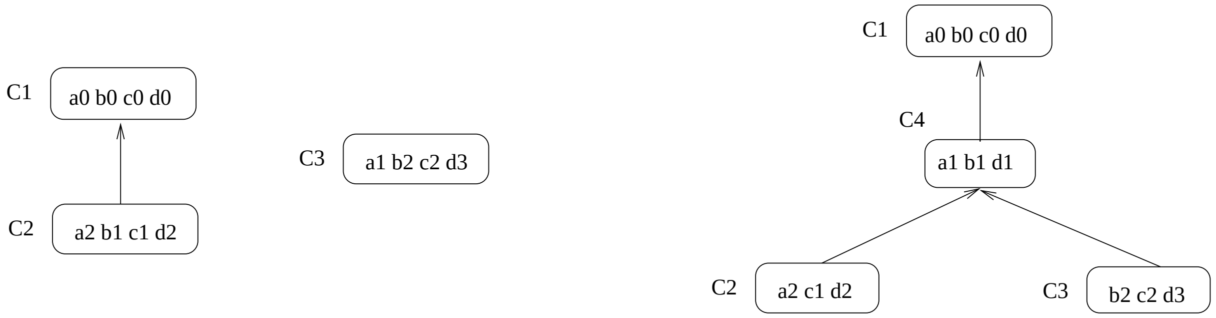


Figure 4: A simple case of overloading

3.3 Examples with overloading and automatic determination of *LCG*

The automatically computation of a *LCG* is not possible in whole generality, but possible for particular cases. The complete set of cases that we are able to handle is presented in section section A and the detail of the computation in section 4.1. We now introduce concrete examples where an *LCG* can be computed and exploited by *Ares*.

- Signatures give the *LCG*

The first example (*cf.* Figure 5) comes from [Mey92]. The existing hierarchy is made of a single class Car. A first class to be inserted is $Truck$. The two properties to be compared are $p_1 = registerDriver(TruckDriver)$ and $p2 = registerDriver(Driver)$. If it is known that $TruckDriver < Driver$ then it can be deduced, regardless of their codes, that $p_1 < p_2$ and that $LCG(p1, p2) = p2$. Then *Ares* stores p_2 in a new factorization class (which we, not *Ares*, name $Vehicle$) made from Car and $Truck$. The same criteria are used to insert the class *AmbulanceTruck* and produce the final hierarchy. *Ares* is able in such a case to produce a hierarchy with several level of overriding.

- 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, it is simple to compute that $d_0 = LCG(d_1, d_2)$. These results allow *Ares* to create the class *GeometricFigure* (except for the name) and to produce the resulting hierarchy.

- Codes and signatures give the LCG

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 $Time$ should produce a factorization class ($Magnitude$)[7] with the method $<=$ common to *Date* and $Time$ and deferred[8] versions of methods $<$ and $=$. The occurrences of the generic properties $<=, <, =$ have to be compared and their *LCG* computed.

The first issue here is to enrich the language describing our signatures in order to note that in the signatures of the methods $<=, <, =$, the types *Date* or *Time* are **anchored** type as defined in Eiffel [Mey92]. Indeed, the

[7] The set of classes whose instances can be ordered.
[8] Pure virtual or subclass responsibility methods.

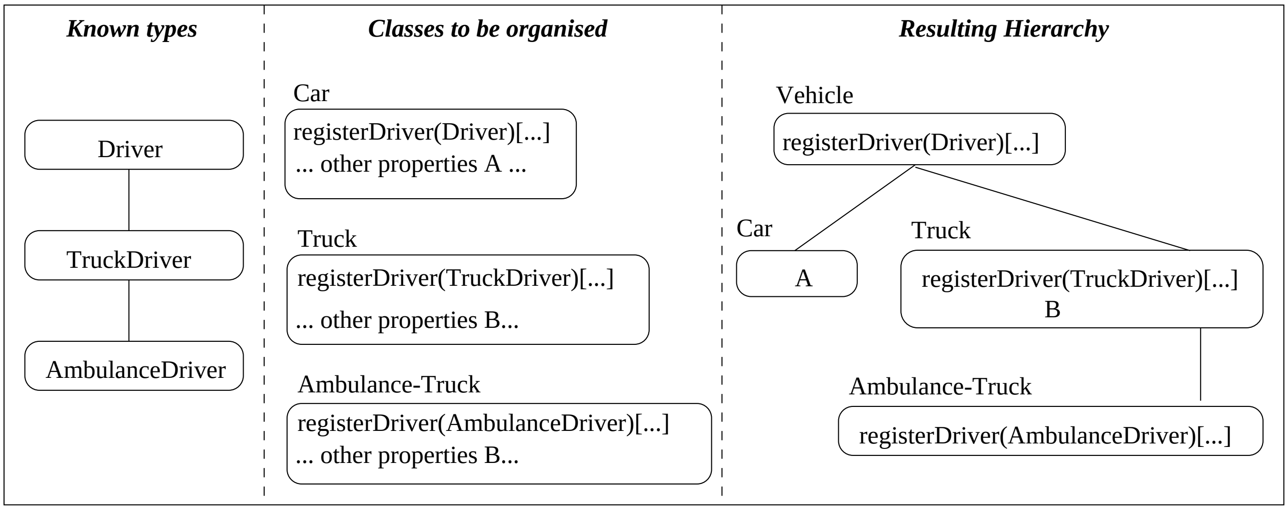


Figure 5: Signatures give the LCG

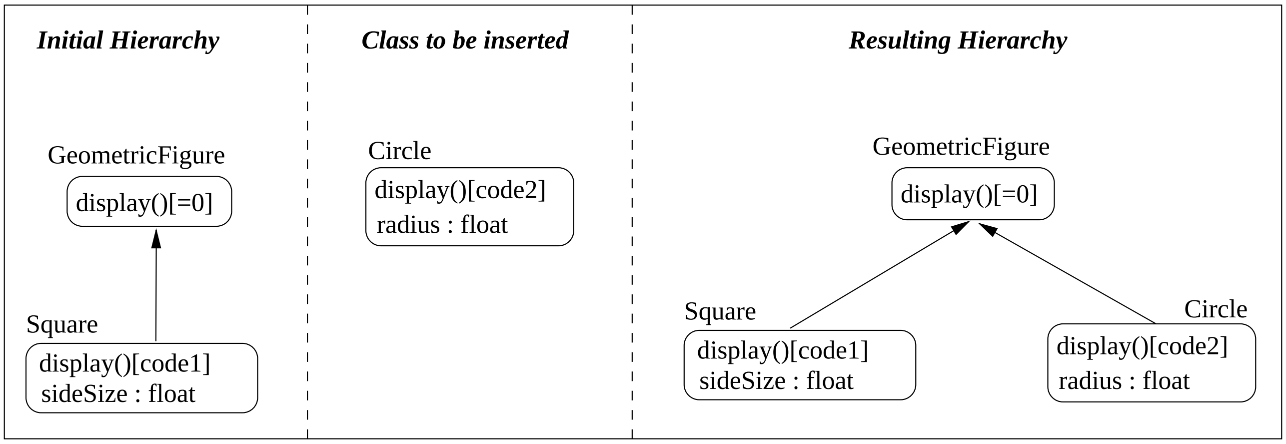


Figure 6: Code gives the LCG

parameter's type of the 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. We then define two signatures that match (in our terminology signatures "potentially equal") as two signatures having at a given position either the same type, or anchored types.
- 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 $code1$. This is enough to confirm that they can be factored via a LCG, which is a method of code $code1$ defined on the same common superclass —$Magnitude$ —of $Date$ and $Time$ and of signature ($Magnitude$).
- The next issue in this example is to compute whether a property $<=$ should remain on classes $Date$ and $Time$. The answer to that question is language dependent. For example, the solution depicted in figure 7 allows to maintain a control on parameter's types of $<=$, but requires that covariant redefinitions be allowed. In Smalltalk, the methods $<=$ do not remain on classes $Date$ and $Time$, and runtime errors will occur when comparing a date and a time. $Ares$, by maintaining precisely what can remain on those two classes allows implementors to choose any language dependent solution.

4 Automatic classes comparisons

The algorithm compares the class to be inserted (that we call A) to all other classes in the hierarchy. This section describes the comparison of A with one of these classes (that we call C) and focuses more particularly on the comparison of properties of these two classes.

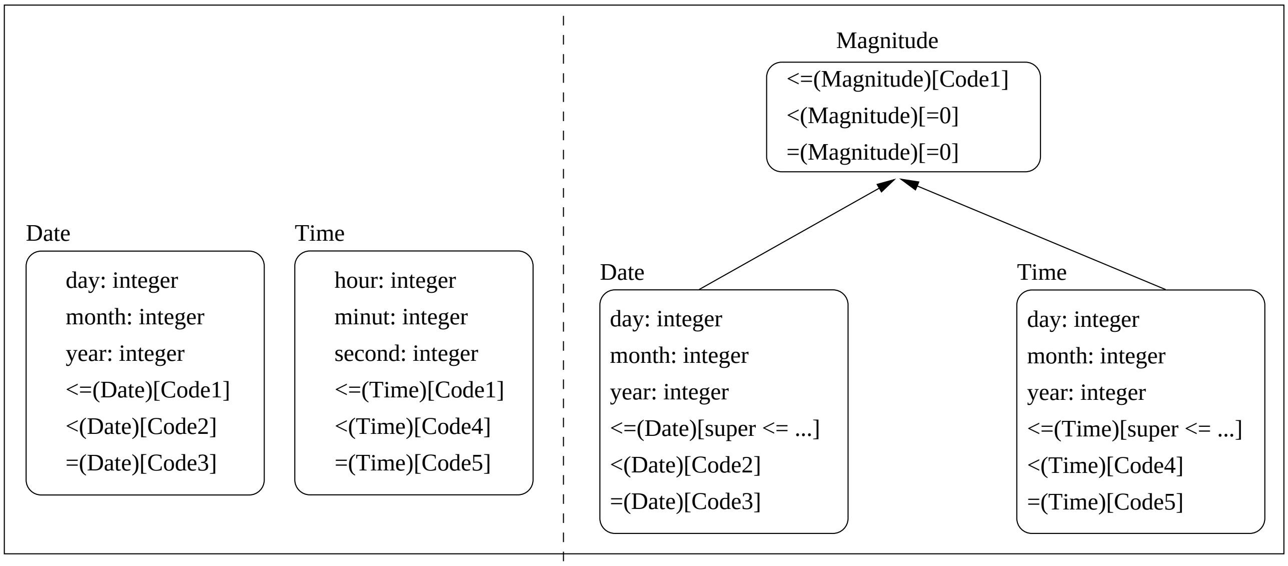


Figure 7: Using codes and signatures

To compare C with A amounts to compare the set of properties declared in C ($Declared(C)$) with the properties of A ($InitialProperties(A)$). The goal of this comparison is to build three other sets, used by the algorithm:

- $ExtractedProperties(C, A)$, the set of properties that should belong to a common superclass of C and A.

  Without overloading, this is simply the set of properties which belong to both $Declared(C)$ and $InitialProperties(A)$. With overloading, $ExtractedProperties(C, A)$ can include new properties that generalize two occurrences of the same generic property belonging respectively to A and C.

- $Remainder(C)$ (resp. $Remainder(A)$, the set of properties remaining in C (resp. A) after the extraction.

The computation of these sets thus requires, in presence of overloading, the computation of the lowest common generalizations (LCG) of two OGPs.

4.1 Automatic comparison of occurrences of generic properties

An LCG computation arises when $Declared(C)$ and $InitialProperties(A)$ contain p_A and p_C respectively, two OGP of the same generic property P[9]. $LCG(p_A, p_C)$ can be either p_A or p_C or the lowest property that both p_A and p_C specialize.

4.1.1 Keys for property comparisons

Additional definitions on OGP and on their signatures are necessary for the comparison of properties.

Instance variables are compared using their types (ordered by $<$). Methods are compared by mixing code comparison and signature comparison.

Code comparison. At this stage of the work, two cases have been considered, the codes of two methods of the same name to be compared are 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 lowest common supertype denoted by $sup(T1, T2)$[10].
As explained in the examples (cf. Section 3.3), we also need to separately consider self-referent signatures, *i. e.* signatures including the class in which the property is defined. Such elements of the signatures will be called anchored-types (in reference to Eiffel anchored type declaration).

[9]Note that, in this case and 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.

[10]When that type does not correspond to an existing class in the hierarchy, because T1 and T2 do not have a single lowest common supertype, that class has to be created by the algorithm. This case is rather uncommon in usual hierarchies.

Definitions of some comparison relationships between signatures used in the algorithm are presented. Let us consider two signatures $S_X = (T_1, T_2, ..., T_n)$ and $S_Y = (T'_1, T'_2, ..., T'_n)$, where T_i and T'_i are known types.

- S_X and S_Y are **equal** ($S_X = S_Y$) if $\forall i\ (T_i = T'_i)$
- S_X and S_Y are **potentially equal** ($S_X =_{pot} S_Y$) if $\forall i((T_i = T'_i)$ or (T_i and T'_i are anchored types))
- S_X and S_Y are **(strictly) comparable** ($S_X < S_Y$) if one is a specialization of the other, for example $S_X < S_Y$, i.e. if $\exists j\ s.t.\ T_j < T'_j$ and $\forall i \neq j\ (T_i \leq T'_i)$.
- S_X and S_Y are **(strictly) potentially comparable** ($S_X <_{pot} S_Y$) if one is a potential specialization of the other. S_X is a potential specialization of S_Y if $\exists j\ s.t.\ T_j < T'_j$ and $\forall i \neq j\ ((T_i \leq T'_i)$ or (T_i and T'_i are anchored types))
- S_X and S_Y are **incomparable**, if ($\exists i$ s.t. T_i and T'_i are incomparable) or ($\exists i, j$ s.t. $T_i < T'_i$ and $T'_j < T_j$).

We then define **equal** (resp. **potentially equal**) properties as properties with the same code (in the case of methods) and equal (resp. potentially equal) signatures.

4.1.2 Computing LCG of two properties

We deal in this section with the issue of computer-aided determination of $LCG(p_X, p_Y)$ in the working context defined in section 2.
Given $p_X\ :\ S_X[code_X]$ with $S_X = (T_1, ..., T_i, ..., T_n)$ and $p_Y\ :\ S_Y[code_Y]$ with $S_Y = (T'_1, ..., T'_i, ..., T'_n)$, the general formula of their LCG is for the signature : $(sup(T_1, T'_1), ..., sup(T_i, T'_i), ..., sup(T_n, T'_n))$, and for the code: $sup(code_X, code_Y)$.
There are several definitions of what is the sup of two codes. We have up to now only considered the simplest one: two codes are either identical or different, and it may happen that a code hides another one (overriding).
$sup(code_X, code_Y) =$
- $code_Y$ if ($S_X < S_Y$ or $S_X <_{pot} S_Y$)
- $code_X$ if $S_Y < S_X$ or $S_Y <_{pot} S_X$)
- $code_X$ or $code_Y$ if ($code_X = code_Y$) and ($S_X = S_Y$ or $S_X =_{pot} S_Y$ or S_X, S_Y incomparable)
- [= 0] if ($code_X \neq code_Y$) and ($S_X = S_Y$ or $S_X =_{pot} S_Y$ or S_X, S_Y incomparable)

Under our working hypothesis, the determination of $sup(T_i, T'_i)$ only raises a problem when T_i or $T'i$ is an anchored type. Three kinds of cases in the determination of $LCG(p_X, p_Y)$ can be distinguished:

- 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 clever code comparisons than those we have already done. We have neither considered yet the case where one is a subset of the other nor the case where the two codes have common subsets, we discuss that point in section 8. Our first studies have shown that to handle these cases induce important changes in the algorithm, indeed either every common sub-codes are factorized - which seems unreasonable -, either the maximal factorization property should be relaxed.
- Cases that can be handled with the above formulas. They are given by mixing code (2 cases) and signature (5 cases) comparisons. They allow *Ares* to deal with numerous and nontrivial configurations. A few examples of such configurations are given below, the whole cases are detailed in appendix A.

4.1.3 Example 1 : p_X and p_Y have the same code, and their signatures are potentially equal

Here, both signatures have at least one anchored type at the same position).
For instance, if p_X is $p_X(T_1, ..., T_i, X, ..., T_n)[code1]$, and p_Y is $p_Y(T_1, ..., T_i, Y, ..., T_n)[code1]$,
then $LCG(p_X, p_Y) = p_m = p(T_1, ..., T_i, sup(X, Y), ..., T_n)[code1]$, where $sup(X, Y)$ is the lowest common superclass [11] of X and Y in which the algorithm will store p_m, if p_m is not already "declared" in a superclass Z containing $p_Z(T_1, ..., T_i, Z, ..., T_n)[code1]$.

An example of such a situation can be found in the *Magnitude* example (cf. Figure 7), where X is $Time$, Y is $Date$, and the considered property is <=. The following LGC is computed by the algorithm and added to $ExtractedProperties(Time, Date)$.
$LCG(<= (Time)[code1], <= (Date)[code1])$ is $<= (sup(Time, Date))[code1])$

Then the remainders are computed:

[11] Superclass in a broad sense, which can be X or Y.

$Remainder(Time) = \{hour : integer, minut : integer, second : integer, ...\}$
$Remainder(Date) = \{day : integer, month : integer, year : integer, ...\}$

Knowing that neither $Remainder(Date)$ nor $Remainder(Time)$ are empty, *Ares* deduces that a factorization class $C' = Magnitude$[12] has to be created in which properties stored in $ExtractedProperties(Time, Date)$ will be declared (among them is $<= (Magnitude)[code1]$, note that its signature has been updated).

4.1.4 Example 2 : p_X and p_Y have the same signatures, and their codes are different.

In such a case, at least a deferred property can be declared on $sup(X, Y)$:
p_X is of the form $p_X(T_1, ..., T_n)[code1]$, and
p_Y is of the form $p_Y(T_1, ..., T_n)[code2]$, then
$LCG(p_X, p_Y) = p(T_1, ..., T_n)[= 0]$.
This situation has been encountered in the second example of section 3.3, (cf. Figure 6) when comparing the methods *display* of classes *Square* and *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_X could also be a specialization of p_Y (or the opposite). Determining this requires either a human expert or more sophisticated techniques for class comparison (is p_X's code a specialization of p_Y's code? Do p_X and p_Y have a worthwhile common subpart to be factorized? Do the other properties allow to conclude ?).

4.1.5 Example 3: p_X and p_Y have different codes, and their signatures are comparable.

One of the properties is a specialization of the other, if for instance $S_X < S_Y$, then $LCG(p_X, p_Y) = p_Y$. This case occurs in the "car-truck" example (cf. Figure 5) if we consider that the two methods *registerDriver* have different codes (*code*1 for *Car* and *code*2 for *Truck*).
The computed LCG is $registerDriver(Driver)[code1]$ that will be declared in the common superclass of *Car* and *Truck*[13].

4.2 Computation of *ExtractedProperties* and *Remainder*

It is now possible to define precisely the sets *ExtractedProperties* and *Remainder* (A still representing the class to be inserted, and C the current visited class in the hierarchy).

- $ExtractedProperties(C, A)$ is the set of all LCG of properties of $Declared(C)$ and $InitialProperties(A)$ having the same name, which are neither equal nor potentially equal to a property inherited by C[14].
- *Remainder(C)* (resp. *Remainder(A)*) is obtained by removing from *Properties(C)* (resp. *InitialProperties(A)*), each property either equal or potentially equal to a property of *ExtractedProperties(C, A)* or inherited from a superclass of C and A.

5 The *Ares* Algorithm

This section describes the kernel of the algorithm.

5.1 Specifications

Input: Inputs are a class hierarchy $H_i = (\mathcal{C}_i, <_{H_i})$ and a meaningful class A to be inserted. $\mathcal{C}_{Mean}$ is the set of meaningful classes of H_i.

[12] We will use the name Magnitude for clarity but of course, *Ares* does not find the name.
[13] The difference with the case 3 (cf. Appendix A is that here $registerDriver(TruckDriver)$ clearly overrides $registerDriver(Driver)$.

[14] More formally, $ExtractedProperties(C, A) =$
$\{p_m = LCG(p_C, p_A)\ s.t.\ \exists P,\ p_C \in P,\ p_A \in P,\ p_C \in Declared(C),\ p_A \in InitialProperties(A),$
$and\ \forall p_i \in Inherited(C),\ p_m \neq p_i\ and\ p_m \neq_{pot} p_i\}$

Algorithm $Ares(H_i,A)$
begin
;;Initializations
$AalreadyCreated \leftarrow false$; $Superclasses(A) \leftarrow \emptyset$; $EmptyClasses \leftarrow \emptyset$
$NewDeclared(A) \leftarrow InitialProperties(A)$; $DirectSubclasses(A) \leftarrow \emptyset$
$LEH_i \leftarrow ComputeLinearExtension(>_{H_i})$

;;Comparison of A with all classes of LEH_i
For (every class C in LEH_i) do
  $Compute(ExtractedProperties(C,A))$
  $Compute(Remainder(C))$; $Compute(Remainder(A))$
  if $Remainder(C) \neq \emptyset$ then
    if $((Remainder(A) = \emptyset) or (ExtractedProperties(C,A) \neq \emptyset))$
      ;;A new class C' has to be created as a superclass of C
      $Create(C')$
      $Declared(C') \leftarrow UpdateSignatures(ExtractedProperties(C,A))$
      $ImmediateSuperclasses(C') \leftarrow MinCommonSups(C, Superclasses(A))$
      $ImmediateSuperClasses(C)$
        $\leftarrow (ImmediateSuperClasses(C) \bigcup \{C'\}) \setminus MinCommonSups(C, Superclasses(A))$
      $Declared(C) \leftarrow Difference(Declared(C), Declared(C'))$
      if $Remainder(A) = \emptyset$ then ;;C' = A, and C is a subclass of A
        $AalreadyCreated \leftarrow true$; $\mathcal{C}_{Mean} \leftarrow \mathcal{C}_{Mean} \bigcup C'$
        $DirectSubclasses(A) \leftarrow DirectSubclasses(A) \bigcup \{C\}$
      else ;;C' is a superclass of A
        $Superclasses(A) \leftarrow Superclasses(A) \bigcup \{C'\}$
        $NewDeclared(A) \leftarrow Difference(NewDeclared(A), Declared(C')$
      endif
      if $Declared(C) = \emptyset$ and $C \notin \mathcal{C}_{Mean}$ then
        $EmptyClasses \leftarrow EmptyClasses \bigcup \{C\}$
      endif
    endif
  else ;;$Remainder(C) = \emptyset$
    if $Remainder(A) \neq \emptyset$ then ;;C is a superclass of A
      $Superclasses(A) \leftarrow Superclasses(A) \bigcup \{C\}$
      $NewDeclared(A) \leftarrow Difference(NewDeclared(A), Declared(C))$
    else $AalreadyCreated \leftarrow true$;;Remainder(A)= $\emptyset$: C = A
    endif
  endif
endfor

;;Creating class A and binding it to $Superclasses(A)$
if not $AalreadyCreated$ then
  $Create(A)$; $\mathcal{C}_{Mean} \leftarrow \mathcal{C}_{Mean} \bigcup A$
  $ImmediateSuperClasses(A) \leftarrow Min(Superclasses(A))$
  $ImmediateSubClasses(A) \leftarrow DirectSubclasses(A)$
  $Declared(A) \leftarrow NewDeclared(A)$
endif
$DeleteEmptyClasses$
end

Figure 8: **The Ares Algorithm**

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

- **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 $\mathcal{C}_{Mean} \bigcup \{A\}$.
- **Preservation of the maximal factorization of properties**
  If H_i is maximally factorized, so is H_f. If H_i is not maximally factorized, H_f will not be but everything common to the hierarchy and the class to be inserted will be factorized.
- **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 $\mathcal{C}_{Mean} \bigcup \{A\}$

5.2 The algorithm

Conceptually, the algorithm (*cf.* Figure 8) can be split in: (1) search of A superclasses and subclasses, (2) deletion of non-meaningful empty classes. We focus here on part (1).
The algorithm visits successively all classes in the input hierarchy H_i going down from the root and following a linear extension LEH_i of $>_{H_i}$. A class is thus visited after all its superclasses.
At each step, the sets $ExtractedProperties(C, A)$, $Remainder(A)$ and $Remainder(C)$ are computed and used to update the following variables (global to the whole visiting process):

- $AalreadyCreated$: a Boolean stating whether a class equivalent to A has been found or created.
- $Superclasses(A)$: the current set of A's superclasses.
- $DirectSubclasses(A)$: the current set of A's immediate subclasses.
- $EmptyClasses$: the current set of the non-meaningful classes which, after a factorization, do not declare any more properties.
- $NewDeclared(A)$: the current set of properties defined in the class A.

Here is the detail of each visit step.

- If $Remainder(C)$ is not empty.

  A is a superclass of C or A and C are incomparable. In the first case $ExtractedProperties(C, A)$ (may be empty) represents the initial properties of A not inherited from $Superclasses(A)$. It is necessary to create a class representing A. In the second case, if furthermore $ExtractedProperties(C, A)$ is not empty, a new common superclass will be create.

  In these cases (characterized by $Remainder(A) = \emptyset$ or $ExtractedProperties(C, A) \neq \emptyset$), we remark that all properties in $ExtractedProperties(C, A)$ are factorized[15] in a new common superclass C', created as a superclass of C, and as a subclass of $MinCommonSups(C, Superclasses(A))$. The function $MinCommonSups$ returns the set of the minimal classes which are C superclasses and belong to $Superclasses(A)$. C becoming a subclass of C', the function $Difference$ (described below) computes which properties should subsequently be removed from $Declared(C)$. In presence of overloading, this operation is not just a single set difference. The function $UpdateSignatures$ simply replaces, if present in the signature of C', $sup(A, C)$ by C'.

  - If $Remainder(A)$ is empty,
    The class C' exactly corresponds to the class A to be inserted.
    For example: $A = C_4$ and $C = C_3$ in Figure 2 (hierarchy H_2). $ExtractedProperties(C, A) = \{c\}$, $Remainder(C) = \{d, e\}$ and $Remainder(A) = \emptyset$.
  - If $Remainder(A)$ is not empty,
    C' is a superclass of A, $NewDeclared(A)$ has thus to be updated in the same way as C.
    An example of such a case can be found in Figure 2 (hierarchy H_3) with: $A = C_5$ and $C = C_3$, $ExtractedProperties(C, A) = \{d\}$, $Remainder(C) = \{e\}$, $Remainder(A) = \{f\}$ and $C' = C6$.

- If $Remainder(C)$ is empty,

[15] And possibly updated to solve potential anchored types indecision in their signatures.

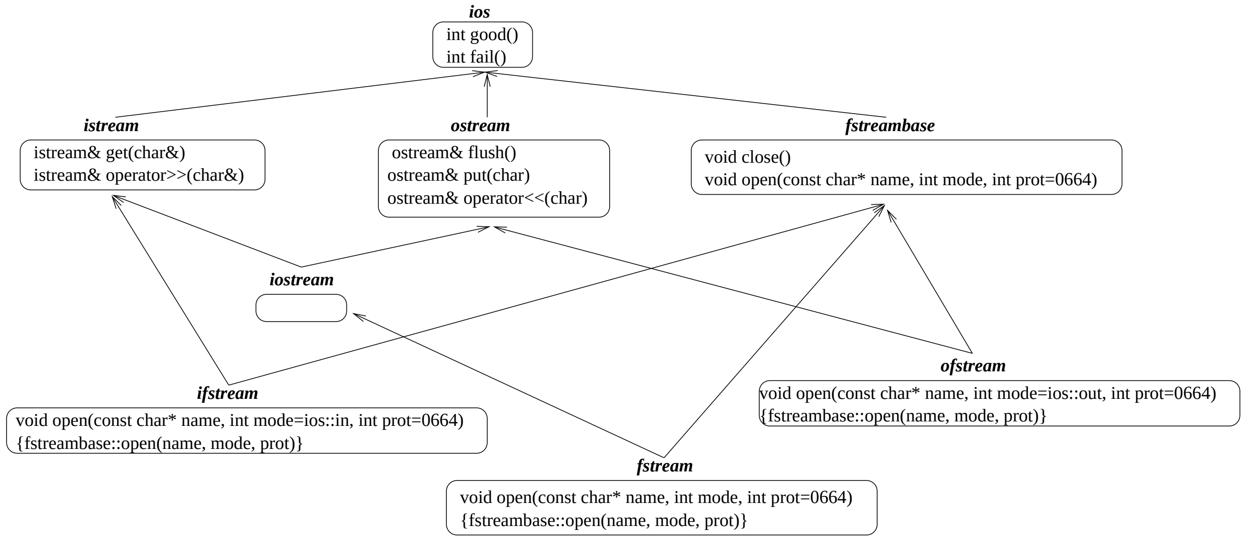


Figure 9: A part of C++ stream hierarchy

- If $Remainder(A)$ is not empty,
  C is a superclass of A, both classes have to be updated subsequently.
  For example: $A = C_3$ and $C = C_2$ in Figure 2 (hierarchy H_1). $ExtractedProperties(C, A) = \{b\}$, $Remainder(C) = \emptyset$ and $Remainder(A) = \{c, d, e\}$.
- If $Remainder(A)$ is empty,
  The class C exactly corresponds to the class A to be inserted.

When the whole hierarchy has been visited, if A has not already been found, it is created, connected to its immediate superclasses (stored in $Min(Superclasses(A))$), to its immediate subclasses (stored in $DirectSubclasses(A)$) and receives its properties (stored in $NewDeclared(A)$).
Due to the restructurations of the hierarchy, some classes do not declare any more properties, if they are non meaningfull, they are deleted properly.

The function $Difference$ computes the new set of declared properties of a class X that becomes, in the context of the algorithm, a subclass of another Y. Properties of X equal to a property of Y are removed and properties of X having the same name and code than a property of Y are updated (their code is replaced by a "super" call)[16].
To replace a code by a "super" call implies simple adaptations of the functions that compare properties and their code that are not described here.

6 Reorganization of actual hierarchies - Case studies

We have applied the algorithm to small but actual hierarchies with their complete set of properties.

6.1 The C++ stream hierarchy

The Figure 9 shows a part of the stream hierarchy taken from the Gnu C++ library. The property sets of classes ios, $istream$, $ostream$, $fstreambase$ is partially represented, whereas the classes $iostream$, $ifstream$, $ofstream$, $fstream$ are complete. The constructors are not represented since they do not modify the results, and in first analysis, default values in method signatures have been ignored.
This stream hierarchy raises, from our point of view, two issues : the method *open* of $fstream$ is useless, and $fstream$ should be a subclass of $ifstream$ and $ofstream$.

[16] More formally,
$Difference(SP_X, SP_Y) = SP_X \backslash E \bigcup F$ where :
$E = \{p_x \in SP_X \ s.t. \ \exists p_y \in SP_Y, p_x \in P, \ p_y \in P, code(p_x) = code(p_y)\}$
$F = \{p'_x \ s.t. \ \exists p_x \in SP_X, \exists p_y \in SP_Y, p_x \in P, \ p_y \in P, code(p_x) = code(p_y), signature(p_x) < signature(p_y),$
$p'_x \leftarrow p(signature(p_x))[super \ p]\}$

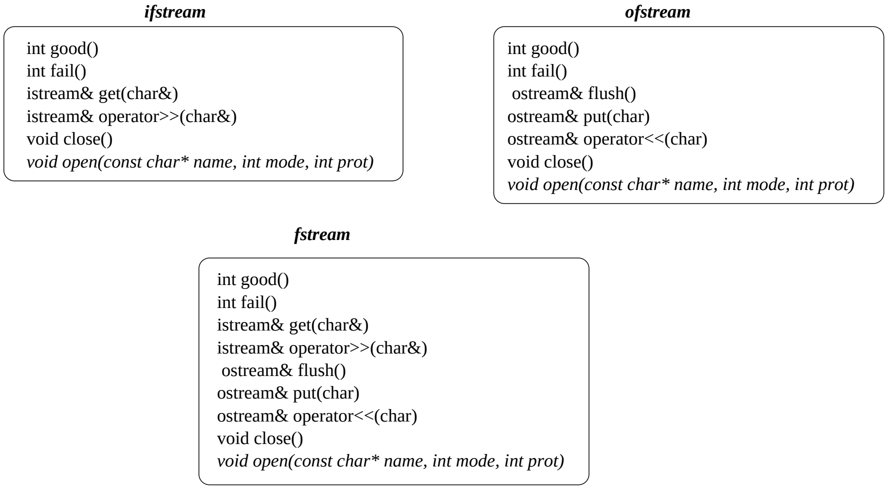


Figure 10: Flattened classes

We have used $Ares$ to "rebuild" this hierarchy, by flattening classes $ifstream$, $ofstream$ and $fstream$, and by reintroducing them. Flattening the classes[17] produces the property sets shown in Figure 10.
The topological sorting on classes of the initial hierarchy H_i (that is the hierarchy of Figure 9 without $ifstream$, $ofstream$ and $fstream$) could be : ios, $istream$, $ostream$, $iostream$, $fstreambase$. To reintroduce $ifstream$, $Ares$ compares successively the flattened class $ifstream$ with the classes of H_i. The first comparison with ios gives :

$$
\begin{aligned}
&ExtractedProperties(ios, ifstream) = \{int\ good(), int\ fail()\}\\
&Remainder(ios) = \emptyset\\
&Remainder(ifstream) =\\
&\quad \{istream\&\ get(char),\\
&\quad istream\&\ operator >> (char),\\
&\quad void\ close(),\\
&\quad void\ open(constchar*, int, int))\}
\end{aligned}
$$

Ios is thus considered as a superclass of $ifstream$. Another way of understanding this is to remark that $Properties(ifstream)$ contains $Properties(ios)$.
For similar reasons $istream$ and $fstreambase$ are recognized as superclasses of $ifstream$.
$ExtractedProperties(ostream, ifstream)$ and $ExtractedProperties(iostream, ifstream)$ are both empty, thus their visit has no effect. Indeed, neither $ostream$ nor $iostream$ declare properties also owned by $ifstream$. This leads to the introduction of $ifstream$ with an empty property set (cf. Figure 11). The discussion for $ofstream$ is symmetric, as is symmetric its place in the hierarchy.
Clearly $fstream$ properties set is exactly the union of the properties sets of all classes in the current hierarchy. $Fstream$ is thus placed, with an empty property set, as a subclass of the lowest classes, that are $iostream$, $ifstream$ and $ofstream$.
The final hierarchy (cf. Figure 11) solves the two issues we had noticed. Its interesting structure is the boolean lattice built up from three atoms corresponding to three characteristics of streams: "input", "output" or "file". Moreover, the diamond lattice reduced to ios, $istream$, $ostream$ and $iostream$, corresponds to the basic "input/output" components. The cubic lattice (cf. Figure 11) can be interpreted as the extension of the diamond lattice to a new component ("file"). The first diamond is reproduced in the diamond induced by $fstreambase$, $ifstream$, $ofstream$, $fstream$, that was not the case in the hierarchy of Figure 9.
Applying the same reconstruction to classes representing input/output services in main memory ($strstream$), we would obtain the hierarchy of Figure 12, that we consider clearer than the original one. There are more edges in hierarchy of Figure 11 than in hierarchy of Figure 9, but the regularity of the structure helps to understand the hierarchy. A lowest multiple inheritance edge number is not necessarily a good measure of the readability.

[17] Concerning the method $open$, we have considered that the codes of the methods $ifstream :: open$, $ofstream :: open$ and $fstream :: open$ are made of a simple call to the super-method $fstreambase :: open$. The "flattened" version only contain one method $open$ with the code given in $fstreambase$.

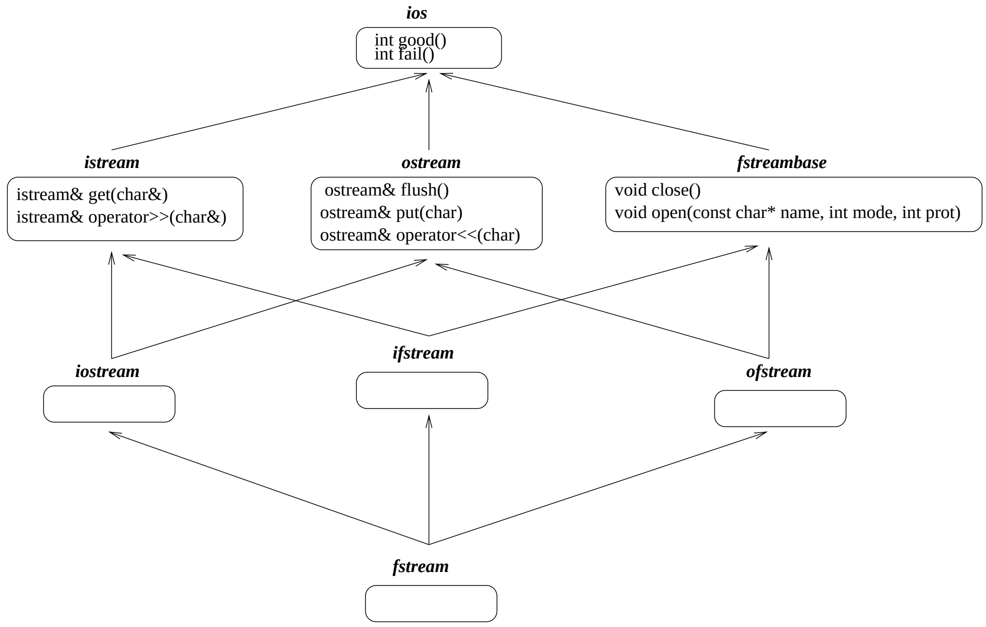


Figure 11: file streams

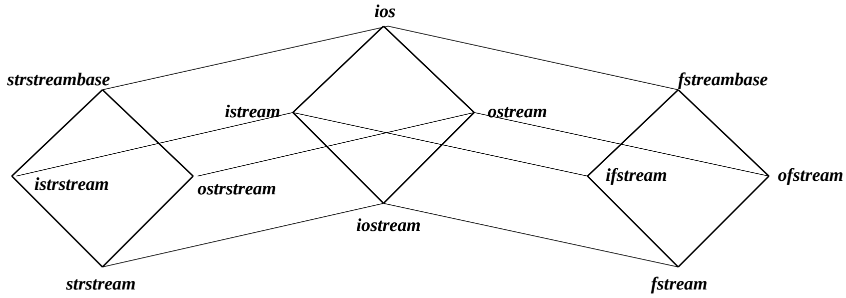


Figure 12: Towards an alternative organization of C++ streams

In order to illustrate the interest of a correct handling of self-referent signatures, consider the classes $istream\_withassign$ (class of the standard input stream (*cin*)), *ostream_withassign* (class of the standard output stream (*cout*)), and *iostream_withassign* which represent the streams having an assignment operator.

```
class istream_withassign : public istream {
public:
  istream_withassign& operator=(istream&); };

class ostream_withassign : public ostream {
public:
  ostream_withassign& operator=(ostream&); };

class iostream_withassign : public ostream {
public:
  iostream_withassign& operator=(iostream&); };
```

The effect of flattening and reintroducing them in the hierarchy would be to create a new class (which does not exist in the current C++ hierarchy), say $withassign$, declaring only a method *withassign*& *operator* = (*ios*&). As previously, a copy of the diamond *ios*, *istream*, *ostream*, *iostream* would be done to represent the new component "withassign", and linked to the initial diamond (cf. Figure 13). Classes *istream_withassign*, *ostream_withassign* and *iostream_withassign* would keep their own *operator* = for typing reasons. $Withassign$ has the great interest of materializing the concept of "streams with assignment" which was previously fragmented in the original C++ hierarchy (see Figure 14). Finally the various versions of the method *open* with parameters default values can be reintroduced in the classes $ifstream$, $ofstream$ and $fstream$ of the hierarchy produced by *Ares*.

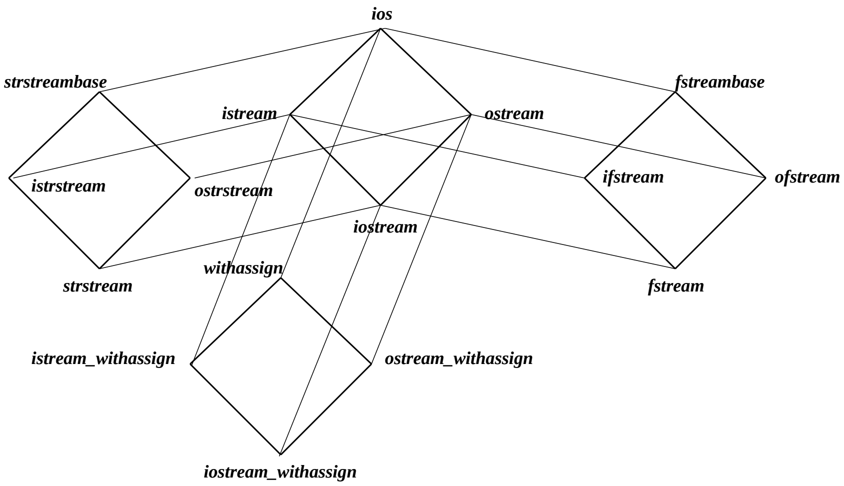


Figure 13: Introduction of *withassign* class

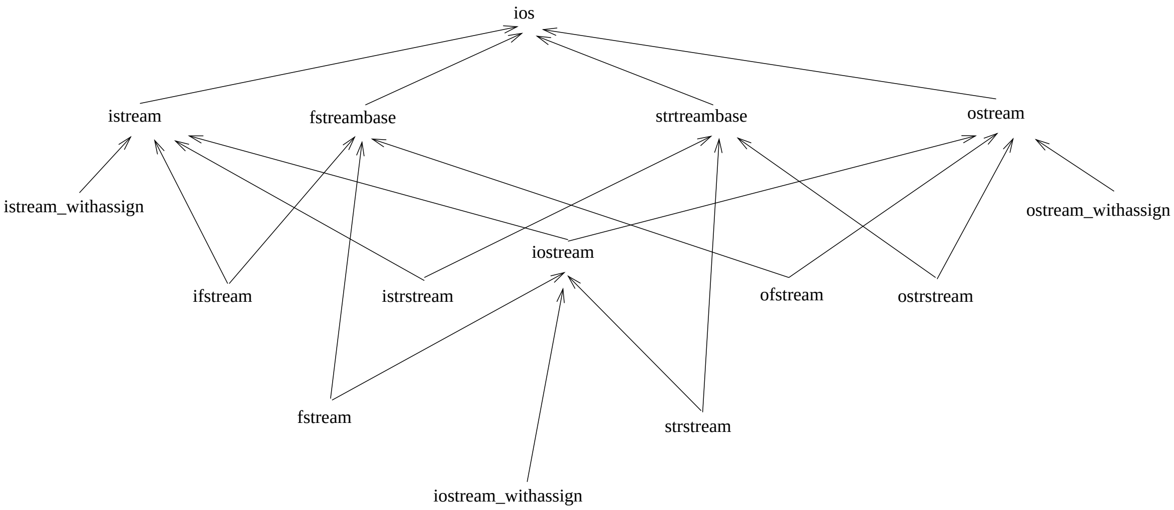


Figure 14: C++ stream hierarchy

6.2 A case study in Smalltalk

Ares has also been applied to a small subpart of the Smalltalk hierarchy. The absence of signatures (untyped world) makes the task of the algorithm more complex because of the lack of data for LCG computation.
Here the result is also interesting for two reasons. On the one hand, it brings again to the fore some factorization problems in the hierarchy. It shows how the algorithm can be used, not necessarily to produce a new final hierarchy but to help human designers to improve the original one. On the other hand it shows some limits of our today's version of the algorithm and proves that the maximal factorization criteria certainly has to be relaxed to deal in a realistic way with actual hierarchies[Cas94].
The subset of the Smalltalk hierarchy we have studied is simply composed of the classes $Object$, $Magnitude$, $Time$ and $Date$ as shown in left part of Figure 15. The properties mentioned in the figure are those that raises factorisation issues ([= 0] still means that a method is deferred, "from" indicates where a factorized method comes from). The initial hierarchy has been broken and rebuild by successive insertions of the four classes. The description of the reorganization is omitted and the result is presented in the right part of the figure. The factorization classes (named $C'1$, ..., $C'5$) are numbered by the order in which they have been created[18].
The positive results lie in the discovery of new abstract classes that can be, for some of them, of interest.

- *Ares* has found classes more general than the initial $Object$ class. For example, the class $C'2$ has been created because the methods = and *hash* are redefined as deferred on $Magnitude$. Thus $C'2$ is a new abstract class that represents all the classes that do not want to inherit (or in other words that override) the default versions of equality and hashing provided by Objects.

[18]This order could change if the order in which classes are inserted also changes - e.g. $Time$ before $Date$ or $Date$ before $Time$ -, but the final result, except for classes names, would be the same.

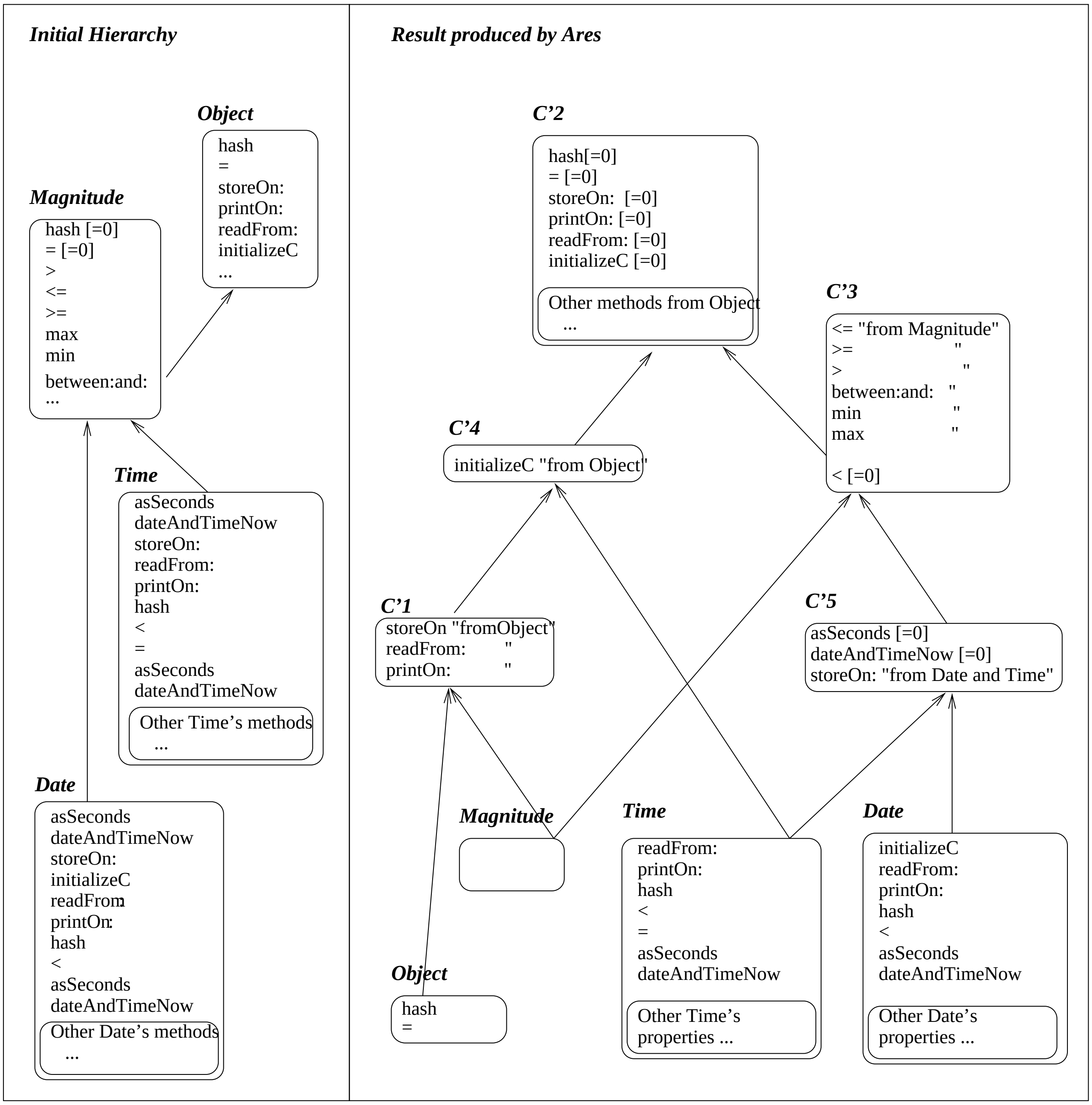


Figure 15: A case study with the Smalltalk hierarchy.

- The three methods *storeOn* : *printOn* : and *readFrom* : have been integrated to $C'2$ (as deferred methods) because they are redefined with different codes in the classes *Date* and *Time*. *Ares* is thus able to group together sets of classes that redefine the same set of methods. A human designer can then decide whether those abstract classes are of interest.

- The class C'5 factorizes what is common to *Date* and *Time*, it suggests a new abstract class that could be called *Time Measurement*

- The new class *Object* is now the superclass of classes that want to inherit a general method for equality and hashing.

Concerning the dark side of the result, the complexity of the resulting hierarchy has to be noticed. This complexity has two causes. It first reflects some imperfections of the language; for example, the class $C'4$ has been created to store the *Object* class method *initializeC*. In fact that method has only one goal, to initialize the *Object* class, and should not be inherited by anyone. The second cause is that *Ares* performs a maximal factorization of properties and creates classes useful in theory but useless in practice. For example, the classes *Date* and *Time* are no more subclasses of *Magnitude* but of $C'3$ instead. The reason is that *Date* (or *Time*) and *Magnitude* own or inherit different versions of the methods *storeOn* :, *printOn* :, and *readFrom* :. To leave *Date* and *Time* as subclasses of *Magnitude*, would require that *Date*'s *printOn* : be a specialization of *Magnitude*'s *printOn* :.
The relaxation of the maximal factorization criterion is then necessary but requires important modifications of the algorithm.

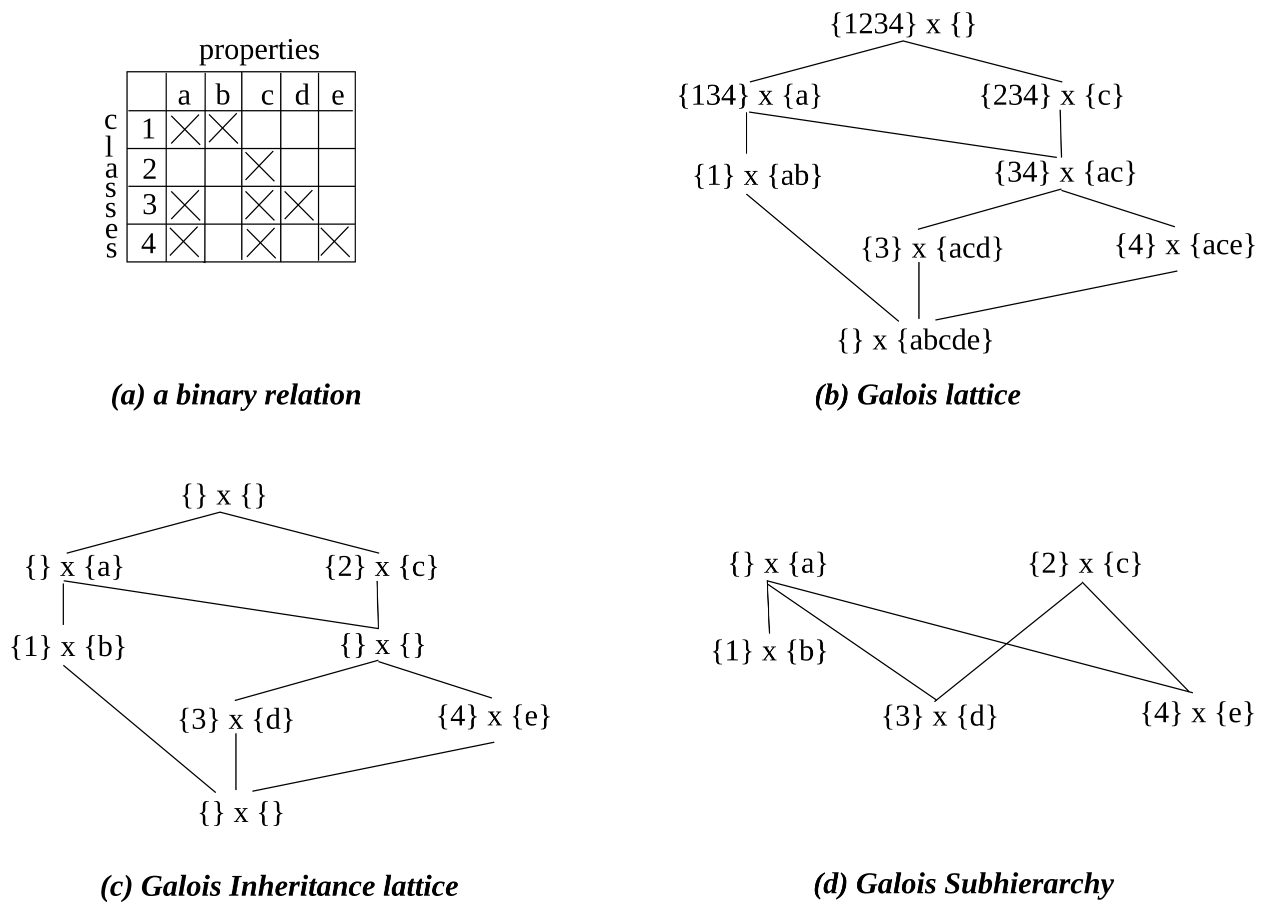
7 Underlying theoretical model

After these case-studies, we come back to the algorithm and focus in this section on its underlying theoretical model.
Many works on class reorganization are based on the maximal property factorization principle which is a strong constraint but does not ensure unique results, the way properties are grouped or the number of classes can be different for two maximally factorized hierarchies. The best compactness can be obtained by using a mathematical structure known in data analysis under the name of "Galois lattice" [Aig79] or "concept lattice"[Wil82, Wil89, Wil92].
Galois lattices [Aig79] are used (at least) in domains such that knowledge acquisition and representation, data analysis, information retrieval, data mining and their interest for hierarchy organization was recently highlighted by [GM93]. In section 8, we will show that some algorithms produce a well characterized sub-structure of a Galois lattice. We recall here the related definitions.
Galois lattice can be seen as a structure which shows all non empty intersections between class property sets, thus making all sharing explicit. A Galois lattice enables the building of unique more compact hierarchies in which properties are maximally factored.
Here is the definition of its basic structure. The point is to associate a class set $\mathcal{K}$ and a property set $\mathcal{F}$ such that:
- all the classes in $\mathcal{K}$ share exactly all properties in $\mathcal{F}$ and nothing more
- symmetrically the properties of $\mathcal{F}$ are owned by all the classes of $\mathcal{K}$ and only by these classes.

Galois lattice (from [Bor92] [19]). 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 [20].
- 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 $\mathcal{F} \supset \mathcal{F}'$)

[19] Alternative definitions can be found in the previously mentioned publications.
[20] $\mathcal{K} \otimes \mathcal{F}$ 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 $\mathcal{F} \subseteq \mathcal{F}'$.



(a) a binary relation

(b) Galois lattice

(c) Galois Inheritance lattice

(d) Galois Subhierarchy

Figure 16: Galois lattice and substructure

The Figure 16, [21] shows a binary relation (part a) and its Galois lattice (part b). 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. The isomorphism is built by changing the label vertices by keeping the right member of the cartesian product.

In order to produce something closer to an inheritance hierarchy, a new labeling removes redundant information. A vertex v, labelled by $\mathcal{K} \otimes \mathcal{F}$ can 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 16 part (c)).

Galois inheritance lattice
A Galois inheritance lattice $GIL(\mathcal{R})$ is isomorphic to a Galois lattice. To $V = \mathcal{K} \otimes \mathcal{F} \in GL(\mathcal{R})$ is associated 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 labeling of the same structure, the size of which being exponential in number of classes and number of properties. It is possible to store only a subpart of it in a Galois Sub-hierarchy [GM93].

Galois Sub-hierarchy
The Galois Sub-hierarchy $GSH(\mathcal{R})$ is the order deduced from $GIL(\mathcal{R})$ by removing "empty" vertices[22], *i. e.* vertices such that $V_r = \emptyset \otimes \emptyset$.

Due to the difference of size between Galois lattice $GL(\mathcal{R})$ and Galois Sub-hierarchy $GSH(\mathcal{R})$ (c.f. Figure 16, part d), algorithms that build directly $GSH(\mathcal{R})$ are more efficient than those that build $GL(\mathcal{R})$ and then reduce $GL(\mathcal{R})$ to $GSH(\mathcal{R})$. [MGG90] have proposed a "global" algorithm, that builds the whole $GSH(\mathcal{R})$ from the

[21] Usually, only Hasse diagram (also called transitive reduction) is drawn to represent order and lattices.
[22] Vertices can be deleted because they do not 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.

class descriptions, here the $\mathcal{R}$ matrix. Former versions of *Ares*[DDHL94a, DDHL95] as well as [GMM95] are "incremental" [23] algorithms, that use directly an already built $GSH(\mathcal{R})$. Today's version of *Ares* hierarchy representation [DDHL96] has an underlying Galois sub-hierarchy structure.

8 Comparison with related works

Related works may be studied from three viewpoints: the strategy used to reorganize hierarchies, the features of the hierarchy and the handling of overloading.

8.1 Strategies

Besides a factorization algorithm, a decomposition principle is presented in [Cas94], which makes explicit the different abstraction steps used in an inheritance edge. This decomposition is an interesting complement to the factorization, and mixing them is a promising issue.
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.
- Factorization is not the single possible strategy for reorganization, Cas94 has described a decomposition algorithm able to "separate various abstraction steps merged in a single inheritance link" [Cas94]. Decomposition can be an interesting complement to the factorization, and mixing them is a promising issue.
- Another approach has been called refactoring. It consists in the factorization of all instructions common to several methods[Moo96]. Such factorizations are only based on syntactic criteria, they will produce numerous methods and classes, the semantics of which being certainly quite unclear.
- Algorithms can be global or incremental.
  Global algorithms [MGG90, LBSL91, LBSL90, Cas91, Moo95, MC96, CL96] 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, DDHL96].
  [Cas92] proposes an interesting variation where the user limits the exploration to a part of the hierarchy.

All of these strategies may lead to the same results. 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 set of classes is $\mathcal{E}$, starting from A and $\mathcal{E}$ and forgetting the structure of the hierarchy. 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.

8.2 Underlying hierarchy models

The underlying model used to represent hierarchies is more or less restrictive. [Cas92] does not impose any constraints on the input hierarchy but does not give any formal characterization of the result. In [LBSL91, LBSL90], there is a strong constraint on hierarchies in which only leaves can represent instanciable classes, and the hierarchies being produced using heuristics, there is no easy way to characterize the final result.

A second set of algorithms use implicitly ([Run92, MS89, MC96, CL96]), or explicitly ([GM93, GMMM95, DDHL94a, DDHL95, DDHL96]) with further adaptations, the Galois lattice of the *Class-Property* relation to encode hierarchies.
[Run92, MS89] use the whole lattice (precisely a sup-semi-lattice) and this raises some problems. Firstly because of space consumption and secondly, because this structure imposes some constraints on the hierarchy. For example, Figure 2 shows how it can forbid the deletion of a class (C_6 in H_5); indeed, if the deletion is achieved (as in H_6) then C_3 and C_5 have two lowest common superclasses C_2 and C_8 and the hierarchy is no longer a lattice.

More cleverly, [GM93] proposed to use a Galois sub-hierarchy as defined in section 7 to improve space complexity. However, for the same example, the Galois sub-hierarchy imposes an opposite constraint: the class C_6 must be deleted even if it is meaningful (cf. section 2.3).

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

Two kinds of algorithms produce Galois sub-hierarchies, [MGG90, MC96, CL96] are global and [DDHL94a, GMM95, DDHL95][24] are incremental.
The current version of our algorithm also produces a Galois sub-hierarchy and thus has formally well characterized results. The main difference with a above ones lies in the handling of overloaded properties as explained below.

8.3 Taking overloading into account

Initial studies [LBSL90, LBSL91] did not take overloading into account. A first advance has been proposed in [Cas92], which allowed a pure virtual method to be overriden by an implemented one which itself cannot be overriden. A second step has been described in [MS89, Run92, GM93, DDHL95]: overloading can be taken into account, provided there is an "oracle" able to compare two occurrences of the same generic property, and give their lowest common generalization(s). Furthermore, these algorithms require, to handle overloading, that all occurrences of all generic properties be stored somewhere (in the class-property binary relation table for global ones and the classes for incremental ones).

The current version of *Ares* presented in this article integrates the idea of a computed comparison of overloaded properties not reduced to simple equality. Only declared properties are stored in classes and LCGs are computed when needed. This version also proposes a partial automation of the comparison of two properties based on code and signatures, including cases of "self-referent" signatures.

9 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 automation, *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.

We have presented two case-study, one in a typed world that highlights the *Ares* possibilities in the handling of overloaded properties, another in an untyped world showing both the interest of reorganization and some of the limits of the algorithm.
These limits define our future works.
The first issue (highlighted by the Smalltalk case study) is related to the limitation of the number of factorization classes. The solution requires a relaxation of the maximal factorization criteria and implies important modifications of the algorithm.
We also plan to extend the number of handled cases of automatic comparison of generic properties. Code comparison can be extended to the cases where a code is included in another and where two codes have common subsets. In this, we will not exactly follow the refactoring school ([OJ93] and [Moo96]) since we plan to limit ourselves to refactoring sub-parts of occurrences of same generic properties. This study is in progress.
Another difficult issue would be to combine this work with linearization algorithms [DHHM94] used to solve conflicts in hierarchies with multiple inheritance.

Finally, 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.

[24]In [DDHL95] a slight modification is required to avoid the deletion of meaningful classes.

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.

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

[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.

[Cas94] E. Casais. Automatic reorganization of object-oriented hierarchies. *Object-Oriented Systems*, 1(2):95–115, 1994.

[Cas95] E. Casais. Managing class evolution in object-oriented systems. In O.Nierstrasz and D.Tsichritzis, editors, *Object-Oriented Software Composition*, pages 201–244. Prentice Hall, 1995.

[CL96] J.-B. Chen and S. C. Lee. Generation and reorganization of subtype hierarchies. *Journal of Object Oriented Programming*, 8(8), 1996.

[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 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.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96.*, 31(10):251–267, 1996.

[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.

[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.

[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.

[Wil82] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, in I. Rivals (Eds)*, 23, 1982.

[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.

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

Figure 17: Mixing code and signature comparison

A Appendix: ten properties comparisons cases.

This appendix summarizes the 10 possible cases of properties comparison that *Ares* is able to handle. The issue is to compute the lowest common generalization of two occurrences of the same generic property: $LCG(p_A(S_A)[Code_A], p_C(S_C)[Cod$ (p_A being declared in the class A to be inserted and p_C in a class C).

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*.

cf. Section 4.1.2

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 $LCG(p_A, p_C) = p_C$.

In the "car-truck" example (*cf.* Section 3.3, Figure 5), no hypothesis have been made regarding 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.**

Cf. Section 4.1.2

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.**

Cf. Section 4.1.2

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 $LCG(p_A, p_C)$
$= p(T_1, ..., T_i, X, ..., T_j, sup(A, C), ..., T_n)[code1]$.

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 properties) factorization classes. The issues here are (1) how to obtain a more precise rule and (2) how to optimize the hierarchy thereafter.