Deriving Normalized Is_a Hierarchies by Using Applicability Constraints

Nadira Lammari, Régine Laleau, Mireille Jouve, Xavier Castellani
CEDRIC-IIE (CNAM) laboratory
18 allée Jean Rostand 91025 Evry - France
Phone: (33 1) 69 36 73 93
Fax number: (33 1) 69 36 73 05
Email: {lammari, laleau, jouve, castellani}@iie.cnam.fr

Abstract. The paper presents an object-oriented normalization mechanism which splits a class into an inheritance hierarchy. It defines a normal form for classes. This normal form is required if an existant factorization mechanism (which is also an object-oriented normalization mechanism) is to be used.
The proposed splitting mechanism is applied for each class of a model. It uses a set of applicability constraints defined between characteristics of a class (methods and attributes). These constraints translate part of the semantics of an application that will be expressed in the generated inheritance graph. They are supplied by the designer and are of three kinds: exclusive applicability, conditioned applicability and mutual applicability constraints.

1 Introduction

Object-oriented concepts are a good basis for data models used in next generation database applications such as CAD and CAM systems, knowledge-based systems and multimedia information systems. The concept of inheritance is one of the essential elements of the object paradigm. It provides greater expressiveness, increases the reusability and facilitates the maintenance. It can be of specialization or implementation. The implementation inheritance expresses the reusability of a super-class characteristics by its subclasses. It is useful in the object implementation stage. The specialization inheritance is a more powerful concept. It not only expresses the characteristic reusability, but it also translates the IS_A semantics.

Many research projects look into the problem of the inheritance definition. In the literature, we have registered three research directions: validation rules, optimisation rules and inheritance derivation mechanisms. The validation rules ([Gol92], [Lin92], [Wir90], etc.) help the designer (or the used CASE tool) to judge about the validity of designed inheritances (e.g. no cycles in an inheritance hierarchy). The optimization rules ([Cas93], [Lal92], [Thi94]) guide the designer to decide about retaining or not a designed inheritance. The decision is taken after computing some ratios or metrics

(e.g. number of characteristics in a subclass) and comparing them to thresholds. The inheritance derivation mechanisms ([And92], [Ber91], [God93], [Lie91], [Thi93]) are processes for reorganizing classes. They are considered as object-oriented normalization mechanisms, since they aim at reducing semantic or syntactic redundancies. They are factorization mechanisms based on the analysis of syntactic and semantic similarities of classes (most of them only consider the class structure). Although dedicated to be applied in the conceptual design stage, these mechanisms derive specialization as well as implementation inheritances, in spite of the fact that implementation inheritances should not appear in this stage. Furthermore, the distinction between the two inheritance types during the process is not made and some implementation inheritances can be derived to the detriment of specialization inheritances.

One could believe that inheritances can be derived only by factorization. In reality, this is not true. Indeed, classes which, in fact, gather different entities having common characteristics hide inheritance hierarchies that must be exhibited. For example, let C be a class describing all the museums and the historic monuments of a country. Its structure is defined by the mandatory attributes *Name*, *Description*, *Address*, and *Fees* and by the optional attributes *Century* and *Specialized_Branches*. Obviously this class can be decomposed into four classes C1, C2, C3 and C4. C1 is described by the attributes *Name*, *Description*, *Address*, and *Fees*. The classes C2 and C3 are subclasses of C1. C2 has as specific and mandatory attribute *Century*. It represents the historic monuments of the country. C3 has as specific and mandatory attribute *Specialized_Branches*. It represents the museums of the country. C4 describes the set of tourist sites which are both museums and historic monuments.
Furthermore, if we try to apply the above-mentioned mechanisms to such classes (containing optional attributes), they will derive "bad" inheritances or they will deduce similarity of classes that are semanticaly different.

In this paper, we propose a mechanism that derives inheritances by splitting classes rather than factorizing them. It can be applied to each class of a model which contains optional attributes. In this way, inheritances that the factorization mechanisms cannot derive are deduced. The splitting of classes is not necessarily the ultimate goal we want to achieve. However, it allows a maximal splitting of classes according to supplied constraints. This decomposition can then be reviewed according to optimization problems. In this case, some factorizations are necessary but integrity constraints expressing the lost semantic of the decomposition will have to be added.

The proposed mechanism is based on the analysis of applicability constraints defined between class characteristics (attributes and methods). These constraints are supplied by the designer and capture part of the semantics of an application. They are of three kinds: mutual, exclusive and conditioned applicability constraints.
Conditioned applicability constraints between attributes (widely known in database literature as existence constraints) were originally studied by [Mai80], [Mai83], then by other authors such as [Atz83]. They have formally defined them (formal definition

and inference rules) and have also studied their interaction with functional dependencies. More precisely, these constraints between attributes have mainly been used to extend the relational theory to relations with null values. Another use was developed by [Halp91], [Bla94], [Kor95] and [Wei91]. Halpin [Halp91], for example, has used existence constraints as integrity constraints in order to model conceptual constraints such as subset constraints in a relational implementation. By comparison with these works, we use these three kinds of constraints between attributes (mutual, exclusive and conditioned applicability constraints) to deduce the structure of the different types of entities that are represented by a class [Lam94a]. Moreover, we have extended the formal definition of the conditioned applicability constraint to the exclusive and mutual applicability constraints. However, to our knowledge, no work, so far, defines or uses the applicability constraints between methods or attributes and methods. This paper gives only formal definitions of these constraints. Inferences rules, soundness and completeness proof of the given formal systems are presented in [Lam94a].

We finally describe the splitting mechanism. It is decomposed into three phases. The first one verifies the consistency of the set of applicability constraints. The second phase consists in deducing from the class structure the set of substructures compatible with the class applicability constraints. This phase has been completely described in [Lam94b]. It uses the applicability constraints between the class attributes. The last phase consists in defining all the classes of the inheritance graph. The classes are deduced by taking into account the applicability constraints between methods and between attributes and methods. A deduced class will have one of the structures previously defined in the second phase and its behaviour will be described by a subset of the methods of the initial class.

Our paper is organized as follows. The example to which we refer all through the paper is mentioned in Section 2. The three kinds of applicability constraints are formally defined in Section 3. In Section 4 we present the mechanism which splits a class into an inheritance hierarchy. Section 5 concludes our work.

2 Example

In this section, we present the example that illustrates the concepts we use. It concerns the class Person of the model of the information system which manages students and teachers of an university. Its attributes are: Name, First_Name, Address, Academic_Year, Academic_Cycle, Hire_Date, Function, Thesis_Subject, Firm_Name, Firm_Phone. Teachers are of three kinds: part-time lecturers, full-time lecturers and temporary assistants. Part-time lecturers are full-time workers in their company. Students are in an academic cycle (1, 2 or 3) and in an academic year. Some PhD students are hired as temporary assistants. Students cannot be part-time lecturers. The designer has defined the following methods: Change_Thesis_Subject, Hire_Teacher, Display_Students_Temporary_Assistants, Display_PhD_Students,

Display_Students_Academic_Cycle_1, Display_Students_Academic_Cycle_2, Display_Part_Time_Lecturer, Register_Student, Exclude_Student, Modify_Address, Display_Teachers.

3 Applicability Constraints

In this section we present a kind of constraints that we consider worthwhile because they allow the reorganization of a class into a set of classes related by IS_A relationships: applicability constraints between characteristics of a class. Their definition are based on the notion of applicability domain that we first outline.

3.1 Applicability Domain

According to [Cod90], a missing value for an attribute in an instance of its class, in practice, results from the fact that either the value for this attribute is temporarily unknown but applicable, or that its value can never be known because this attribute is inapplicable to this instance. In this latter case, we can define an applicability domain to each attribute of a class, that is the set of instances for which the attribute may have a value. For instance, the attributes Name, First_Name and Address are applicable to every instances of the class Person. On the other hand, the applicability domain of the attributes Academic_Cycle and Academic_Year is constituted only by instances of Person describing Students.

Each class method also has its applicability domain. It is constituted by the set of instances that can be consulted, modified, suppressed or created by this method. For example, the method Modify_Address is applicable to every instance of the class Person because every person can change his/her address. On the other hand, the applicability domain of the method Display_Teachers is reduced to the set of possible instances that describe teachers because only these can be displayed using this method.

3.2 Applicability Constraints

Our aim is to gather characteristics having the same applicability domain into the same class. However, we cannot define precisely an applicability domain at the design stage. But, we can approximate it by comparing it to other applicability domains. This is translated by the applicability constraints. They are of three kinds: mutual, conditioned and exclusive applicability constraints.

Intuitively, an exclusive applicability constraint between two characteristics X and Y of a class, denoted $X \nleftrightarrow Y$, captures that, for each instance where X is applicable, Y is not applicable and vice versa. In other words, their applicability domains are disjoined. For example, the applicability of the attribute Academic_Cycle to an instance of the class Person excludes the applicability of the attribute Firm_Name and vice versa. This results from the fact that the first attribute is applicable to every

instance describing a student, whereas the second one is applicable to every instance describing a part-time lecturer who cannot be a student.

A mutual applicability constraint between two characteristics X and Y of a class, denoted $X \leftrightarrow Y$, translates that, for each instance of the class where X is applicable, Y is also applicable and vice versa. In other words, the applicability domains of the two characteristics are equal. In the class Person, for instance, the attribute Academic_Cycle and the method Register_Student are mutually applicable. Every student who is registered is in an academic cycle and conversely every student who is in an academic cycle has previously been registered.

The conditioned applicability constraint between two characteristics X and Y, denoted $X \mapsto Y$, expresses that, for every instance of the class where X is applicable, Y is also applicable; the converse is not always true. In other words, the applicability domain of X is included in the applicability domain of Y. For example, the applicability of the method Change_Thesis_Subject implies the applicability of the method Modify_Address. Indeed, only Ph.D. students have thesis subjects and therefore can change the subject of their thesis. However, any instance of the class Person, so including Ph.D. students, can have its address changed.

Although presented between characteristics, these constraints can also be used to capture relationships between sets of characteristics. We take this into account in the following formal definitions.

Formal definitions.

Let:

- C (U, M, I) a class. U represents the set of attributes of C, M the set of methods and I the set of *all the possible* instances of C.
- X and Y two sets of $U \cup M$ composed respectively by $x_1,\ldots,x_2,\ldots,x_k,\ldots,x_n$ and $y_1,\ldots,y_2,\ldots,y_p,\ldots,y_m$.
- D_{x_k} (respectively D_{y_p}) the applicability domain of the characteristic x_k (respectively of y_p).

We say that:

- X and Y are mutually applicable or $X \leftrightarrow Y$, if and only if:

$$\forall x_k \in X, \forall y_p \in Y,\ D_{x_k} = D_{y_p}$$

- the applicability of X excludes the applicability of Y or $X \not\leftrightarrow Y$, if and only if:

$$\forall x_k \in X, \forall y_p \in Y,\ D_{x_k} \cap D_{y_p} = \varnothing$$

- the applicability of X depends on the applicability of Y or $X \mapsto Y$, if and only if:

$$\bigcap_{k=1}^{k=n} D_{x_k} \subseteq \bigcap_{p=1}^{p=n} D_{y_p}$$

Remark 1: let us note that $X \leftrightarrow Y \Rightarrow X \mapsto Y \wedge Y \mapsto X$. The converse is true only if X and Y are sets of only one characteristic.
Remark 2 The inference rules for each kind of applicability constraints and those expressing the interaction between them are described in [Lam94a]. The completeness and soundness proof of the formal systems expressed throughout these inference rules are also presented in [Lam94a].
Remark 3: An attribute of a class can be atomic (Boolean, Character, Integer, ...), or composite if the tuple or set constructor is used. It can also be a reference to another class. For example, we can define for the Person class, instead of Firm_Name and Firm_Phone, the attribute Person_Firm that references the Firm class. Thus the constraint "Person_Firm $\mapsto$ Name" can be stated.
Remark 4: To maintain the structure of the attributes of the initial class in the classes generated by the reorganization mechanism, among all constraints between attributes only those that are defined between attributes which are not components of other ones are considered. In the following, the word "first level attributes" is used to designate this kind of attributes.

4 Splitting Mechanism

Two splitting approaches can be considered: a structure-oriented approach and a behaviour-oriented approach. The first one aims at building a hierarchy of classes by gathering instances with "similar" structures (structures defined by attributes having the same applicability domain). It consists first in deducing the substructures included into the structure of the initial class, then in linking sets of methods to these substructures in order to construct the hierarchy classes. The second approach aims at building a hierarchy of classes by gathering instances with "similar" behaviour (behaviour described by a set of methods having the same applicability domain). It consists first in deducing sets of methods with the same applicability domain, then in associating a structure to each of these sets (the set of attributes required for the applicability of the set of methods) in order to build the hierarchy classes. In this last approach, since, generally, the user processing requirements are evolving quite rapidely [Bou94], the derived inheritance graph will need to be often remodeled. On the contrary, structures are usually far more stable. The main reason is that in a database implementation we must avoid, for performances reasons, changing too often the database structure. This does not mean that the structures of the generated classes are completely stable but that they evolve more slowly. So a graph based on structures will need less changes.This justifies our choice for the first approach which is developed in this paper.
The splitting mechanism is based on the analysis of the applicability constraints supplied by the designer and those derived by using inference rules [Lam94a]. It is decomposed into three phases (see Figure 1). The first one verifies the consistency of the set of applicability constraints. This verification is carried out according to

validation rules (such as $X \leftrightarrow Y \Rightarrow \neg(X \not\leftrightarrow Y)$) described in [Lam94a]. Incompatible constraints and violated rules are shown to the designer.

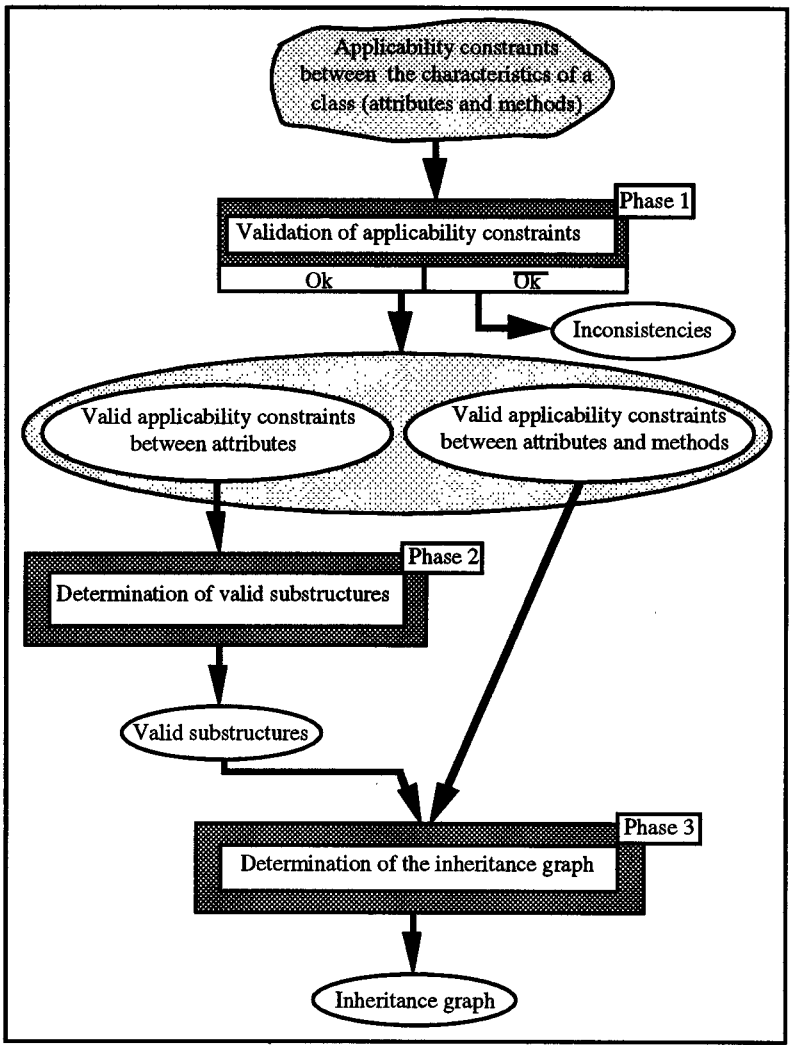


Fig. 1. Description of the reorganization process

The second phase consists in deducing from the class structure the set of substructures compatible with the applicability constraints defined between attributes. This phase is described in [Lam94b] and briefly recalled throughout the example of the class Person in the following subsection. The last phase consists in defining all the classes of the inheritance graph. The classes are deduced by taking into account the applicability constraints between attributes and methods. A detailed description of this last phase is given in Subsection 4.2.

For example, a reorganization of the class Person, taking into account the constraints presented in Figure 2 and according to the structure-oriented approach, gives the inheritance graph of Figure 3. The intermediate results will be given when we describe the second and the third phases.

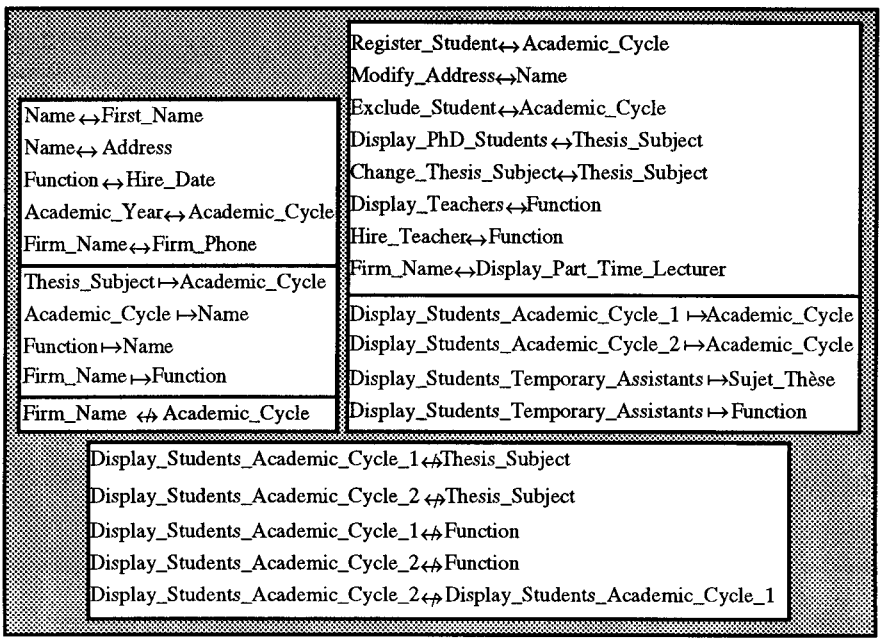


Fig. 2. Applicability constraints supplied by the designer

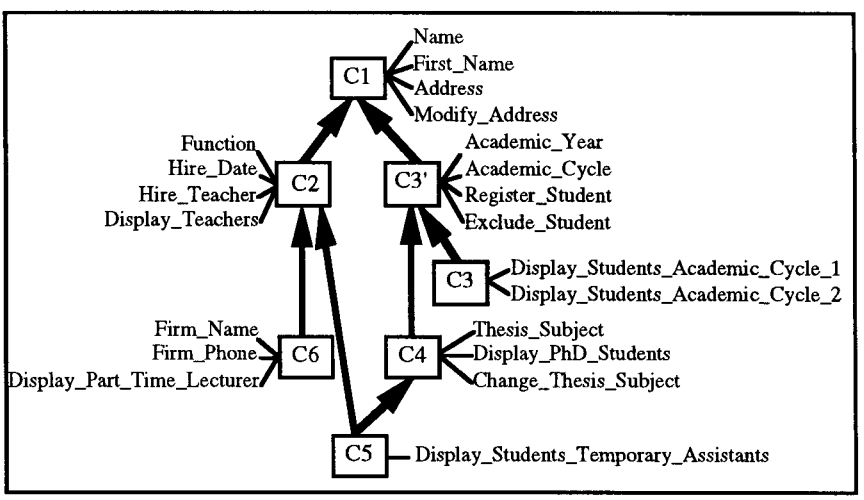


Fig. 3. Generated inheritance graph

Remark: The conditioned applicability constraints between methods and attributes could be very extensive but they can in great part be deduced from the definition of the method signatures. The deduction is further improved if the methods are expressed in a formal specification language (for example VDM or Z). This is a work we want to investigate.

4. 1 Determination of Substructures

Definition of a Substructure.

Intuitively, a substructure of a class is defined by the set of attributes describing it. It constitutes a subset of the class attributes. For instance, among substructures included in the class *Person*, we find: *Student, Teacher, etc.* For example the attributes *Name, First_Name*, Address, *Hire_Date, Function* implicitly describe a substructure that can be called *Teacher*. Among all the possible associations of the class attributes (the power set of the class attributes), only some of them are valid. The valid substructures are those which are compatible with the applicability constraints and, therefore, those to which we can associate possible instances. For example, the set of attributes Name, First_Name, Address and Hire_Date is not a valid substructure because of the applicability constraint: Hire_Date $\leftrightarrow$ Function. In fact, Person instances with only Hire_Date or only Function cannot exist.
So, if we denote $\wp(U_{first})$ the power set of U_{first} (U_{first} represents the set of the first level attributes of a class C) and if we have a set of mutual, conditional and exclusive applicability constraints then, a set p of $\wp(U_{first})$ is a valid substructure if and only if:
Rule (i): Any attribute of U_{first} which is mutually applicable with any attribute of p is in p.
Rule (ii): There are no exclusive applicability constraints between attributes of p.
Rule (iii): Any attribute of U_{first} which determines the applicability of any subset of p is in p.

Formal Definition.

Let:
- C be a class.
- U_{first} be the set of the first level attributes of U: $U_{first} = \{x_1, x_2, ..., x_i, ...\}$.
- F_{cond_app}, F_{mut_app} and F_{exc_app} be respectively the set of conditioned, mutual and exclusive applicability constraints linking the attributes of U_{first}.
- $F^+_{cond\_app}$, $F^+_{mut\_app}$ and $F^+_{exc\_app}$ be respectively the closure of F_{cond_app}, F_{mut_app} and F_{exc_app}.
- $T \in \wp(U_{first})$

We say that T is a valid substructure of C if and only if the following conditions are verified:

$$- \forall x_i \in T,\ x_i \leftrightarrow X \in F^+_{mut\_app} \Rightarrow X \subset T \qquad (1)$$

$$- \forall x_i \in T,\ \forall x_j \in T,\ x_i \not\leftrightarrow x_j \notin F^+_{exc\_app} \qquad (2)$$

$$- \forall Y \subset T,\ Y \mapsto X \in F^+_{cond\_app} \Rightarrow X \subset T \qquad (3)$$

Conditions *(1)*, *(2)* and *(3)* of this definition respectively match the above rules *(i)*, *(ii)* and *(iii)*. We can, now, reduce the number of conditions by reinforcing the hypotheses. By considering the set of mutual applicability constraints of $IRR(F^+_{mut\_app})$ (minimal cover of $F^+_{mut\_app}$) we can decompose U_{first} into sets of attributes mutually applicable (those sets are denoted X_k and represent groups of coexisting attributes). Then, the selection of substructures among the elements of the power set of U_{first} according to Conditions *(1)*, *(2)* and *(3)* is equivalent to the selection of substructures among the elements of the power set of the X_k sets. Thus Condition *(1)* being satisfied in advance, the selection is made exclusively according to Conditions *(2)* and *(3)*. So, condition *(1)* is replaced by Hypothesis *(j)* in the below-mentioned corollary. Moreover, if we eliminate from these latter elements (elements of $\wp(\{X_1,...,X_k,...\})$) those that cannot coexist because there is at least an exclusive applicability constraint linking two sets of attributes mutually applicable, we can, once more, reduce the number of conditions that must satisfy a substructure. So, Condition *(2)* of the definition is replaced by Hypothesis *(jj)* of the following corollary:

Corollary.

Let:

- C be a class.
- U_{first} be the set of the first level attributes of U: $U_{first} = \{x_1, x_2,..., x_i,...\}$.
- F_{cond_app}, F_{mut_app} and F_{exc_app} be respectively the set of conditioned, mutual and exclusive applicability constraints linking the attributes of U_{first}.
- $F^+_{cond\_app}$, $F^+_{mut\_app}$ and $F^+_{exc\_app}$ be respectively the closure of F_{cond_app}, F_{mut_app} and F_{exc_app}.
- $IRR(F^+_{mut\_app})$ be the minimal cover of $F^+_{mut\_app}$

$$-X_k = \overline{x}_k = \{x_i \in U_{first}, x_i \leftrightarrow x_k \in IRR(F^+_{mut\_app})\} \qquad (j)$$

- $\{T_1, T_2,..., T_m,...\} \subset \wp(\{X_1, X_2,..., X_k,...\})$ such that:

$$\forall X_p \in T_m,\ \forall X_q \in T_m,\ X_p \not\leftrightarrow X_q \notin F^+_{exc\_app} \qquad (jj)$$

We say that T_m is a valid substructure of C if and only if:

$$\forall Y \subset T_m,\ Y \mapsto X \in F^+_{cond\_app} \Rightarrow X \subset T_m$$

Approach for Deducing Substructures.

According to the above rules, it is obvious that the mechanism which calculates all the substructures of a class is based on the analysis of the applicability constraints supplied by the designer. A first naive solution is to deduce all the substructures by selecting among the elements of $\wp(U_{first})$ those verifying the applicability constraints. Another less costly solution is to apply the following mechanism based on the above corollary and detailed in [Lam94b]:
- *Step 1*: determination of the closely linked groups of attributes (sets of attributes mutually applicable to instances of the class).
- *Step 2*: selection, among all the possible unions of the closely linked groups of attributes, of those capable of generating valid substructures (unions that are compatible with the exclusive applicability constraints).
- *Step 3*: selection, among all the selected unions of the closely linked groups of attributes, of those that are self-sufficient to build valid substructures: unions that are compatible with the conditioned applicability constraints.

For example, if the designer supplies the set of applicability constraints between the attributes of the Person class (Figure 2) then the mechanism will provide:

- in *Step 1* the closely linked groups of attributes of Figure 4,

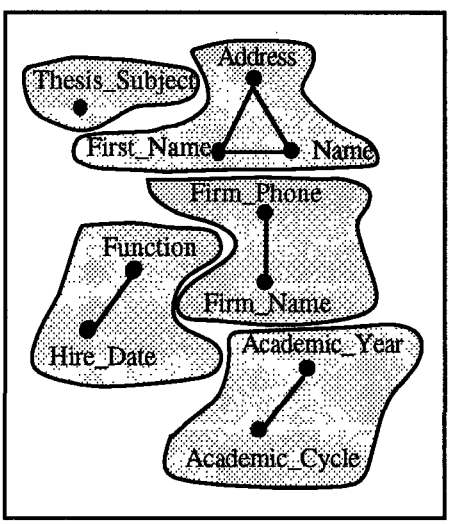


Fig. 4. Groups of coexisting attributes

- then in *Step 2*, the selected unions of closely linked groups of attributes. These unions correspond to the complete subgraphs of Figure 5.

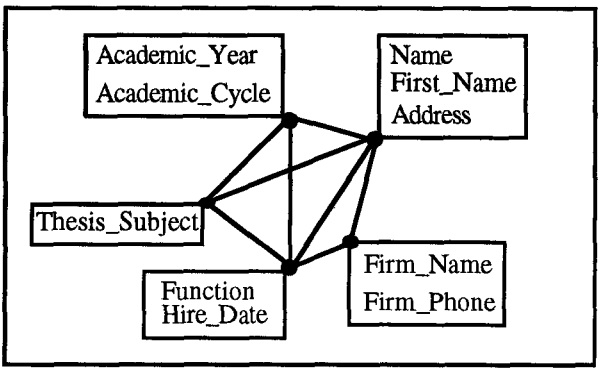


Fig. 5. Possible unions of groups of coexisting attributes

- and then, in *Step 3*, all the valid substructures of Figure 6.

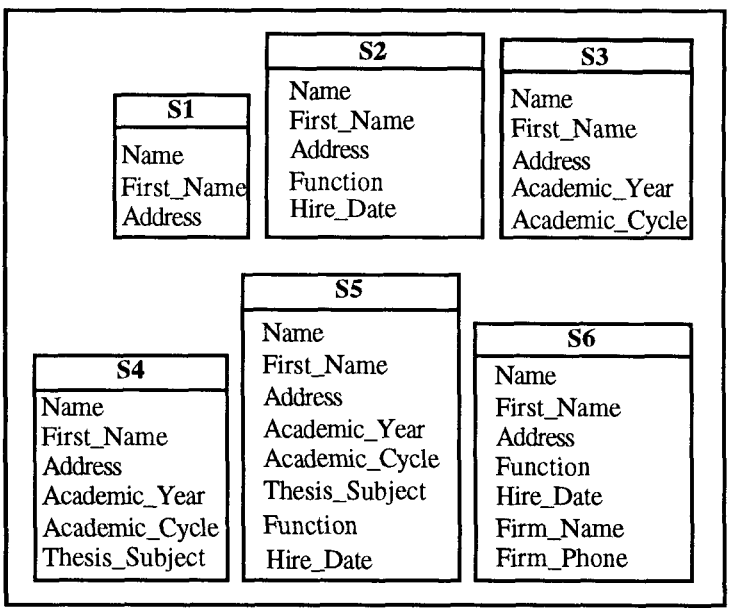


Fig. 6. Valid substructures

The obtained substructures can be organized into an oriented inclusion graph, denoted G(Substructures, Inclusions). G is defined as follows:

Substructures = the set of all the deduced substructures,

$$\text{Inclusions} = \{(S_i, S_j) \in \text{Substructures}^2 / ((S_j \subset S_i) \wedge (\nexists\, S_k \in \text{Substructures} / (S_j \subset S_k) \wedge (S_k \subset S_i)))\}$$

This inclusion graph is the starting point of the next phase of the splitting mechanism. For example, the inclusion graph that corresponds to Substructures S1, S2,..., S6 of the class Person is described in Figure 7.

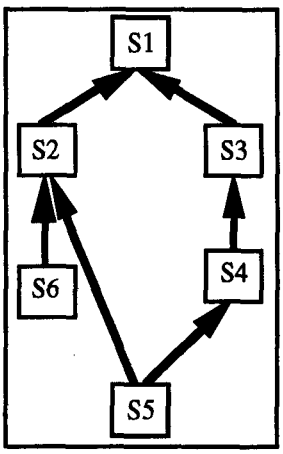


Fig. 7. Inclusion graph

Remark: the inclusion graph has always one sink (S1 in Figure 7) and at least one source (S5 and S6 in Figure 7). It has one sink because we assume that a class has at least one mandatory attribute that gives conceptual meaning to this class (don't forget that we are in the conceptual design stage).

4.2 Determination of the Inheritance Graph

Method Analysis: Linking Methods to Substructures.

As we have used applicability constraints between attributes to deduce substructures, we are going to use the applicability constraints between methods and attributes to decide whether or not a method is to be linked to a substructure. In fact, a necessary condition for Method m to be linked to Substructure S of the initial class is that all the attributes which are necessary to the applicability of m are in S. For example, let us consider Method Display_Students_Cycle_1 and Substructure S3 of Figure 6. The applicability of Display_Students_Cycle_1 is determined by the applicability of the attributes Name, First_Name, Address, Academic_Cycle and Academic_Year. These attributes are in S3. So, the method is able to be linked to S3. This condition can be expressed by the property *(i)* of the following definition.
However, this condition is necessary but not sufficient. To illustrate this, let us refer, once again, to our previous example. Display_Students_Cycle_1 cannot be linked to S4, because its applicability domain and the one of Subject_Thesis are disjoined: Display_Students_Cycle_1 $\not\leftrightarrow$ Subject_Thesis. So, Method m is linked to

Substructure S, if all the attributes that exclude its applicability are not in S. This is expressed in the following definition by the property *(ii)*.

Definition.

Let:
- C be a class,
- S a substructure of C,
- U_{first} the set of the first level attributes of C.
- m a method of C,
- $X_{m,\mapsto} = \{x \in U_{first} / m \mapsto x\}$
- $X_{m,\not\leftrightarrow} = \{x \in U_{first} / m \not\leftrightarrow x\}$.

Method m is linked to Substructure S if and only if these two properties are verified:
- $X_{m,\mapsto} \subseteq S$ *(i)*
- $X_{m,\not\leftrightarrow} \cap S = \varnothing$ *(ii)*

Remark: $X_{m,\mapsto}$ contains the attributes obtained from the conditioned applicability constraints (those inferred are included) and the attributes obtained from the mutual applicability constraints by using the above inference rule:

$$X \leftrightarrow Y \Rightarrow X \mapsto Y \wedge Y \mapsto X.$$

Algorithm for Linking Methods to Substructures.

To determine all the substructures to which Method *m* can be linked, we execute a search in the substructure inclusion graph. The two following properties allow us to reduce the number of comparisons:

(1) $X_{m,\mapsto} \subseteq S_i$, $S_i \in$ Substructures $\Rightarrow (\forall S_j \in$ Substructures $, S_j \supset S_i \Rightarrow X_{m,\mapsto} \subseteq S_j)$

(2) $X_{m,\not\leftrightarrow} \cap S_i = \varnothing$, $S_i \in$ Substructures $\Rightarrow$

$$(\forall\ S_j \in \text{Substructures},\ S_j \subset S_i \Rightarrow X_{m,\not\leftrightarrow} \cap S_j = \varnothing)$$

So, the subgraph, the sink of which represents the smallest substructure containing $X_{m,\mapsto}$ (property *(1)*), can be selected in the inclusion graph. To select this subgraph, the search is carried out in depth from the sink to the sources. The selected subgraph will contain all the substructures to which Method *m* can be linked (property *(i)* of the definition). The substructures to which *m* is really linked are obtained by exploring the subgraph from the sources to the sink and by eliminating all sources that don't verify the property *(ii)* of the definition (use of the property *(2)*). We obtain, the following algorithm:

```
For each Method m Do
    Compute X_{m,↦} and X_{m,↮};
    List_m = ∅;  /* List_m represents the set of substructures to which m is linked */
    Explore G(Substructures, Inclusions) from the sink to the sources until finding a
     substructure S including X_{m,↦};
    Studied_Nodes = the set of the substructures of the graph identified by Sink S;
      /* Explore the graph, identified by S, from its sources to S */
    While Studied_Nodes ≠ ∅ Do
        let s_i = a substructure of Studied_Nodes;
        If X_{m,↮} ∩ S = ∅
        then
                Add to List_m all the descendants of s_i
                Eliminate from Studied_Nodes all the descendants of s_i
        Else eliminate s_i from Studied_Nodes
        End_If;
    End_While;
End_for.
```

The application of this algorithm to the example (Figure 7) gives the result described in Figure 8.

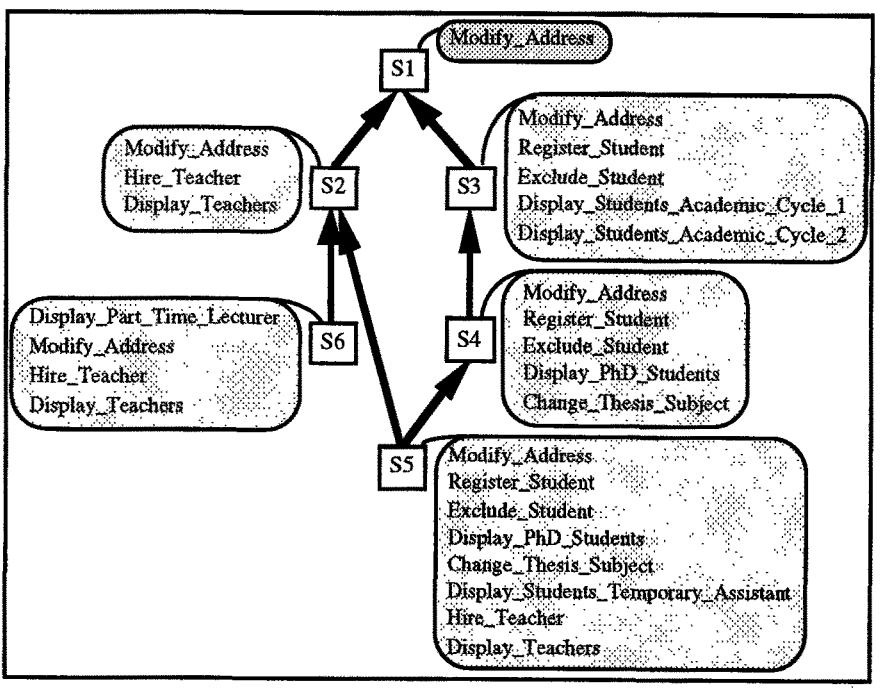


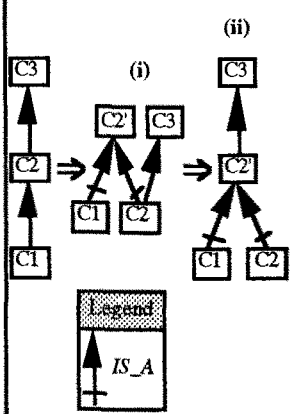
Fig. 8. The linking of methods to substructures

Determination of the Classes and the Inheritance Graph.

This step is the last one in our splitting mechanism. It consists in building the inheritance graph according to the previously defined substructures and according to the set of methods linked to each of them. A class of the inheritance graph is defined by one of the substructures and has a behaviour described by the set of methods which are associated to it. The inheritance graph is not obtained immediately because it may happen that a method could not be linked to all classes of an inheritance branch. It is the case of methods Display_Students_Cycle_1 and Display_Students_Cycle_2 which are linked to S3 but not to S4 and S5 (see Figure 8). The inheritance relationships between some classes cannot, in this case, be defined neatly (without using the concept of masking). To overcome this problem, we define a new class -probably abstract- (in the example C3' of Figure 3) containing only the methods linked to all the classes of an inheritance branch and having as a subclass the class containing the methods which are not linked to all the classes of the inheritance branch (in the example C3 of Figure 3). We obtain, the following algorithm for building the inheritance graph.

Algorithm for building the inheritance graph.

```
Take the inclusion graph, convert the substructures
into classes and keep links (initially unmarked);
/* each substructure defines a class having a behaviour described
by the set of methods that are linked to this substructure */
While it remains unmarked arcs Do
   Let (C1, C2) be an unmarked arc;
   If C1 inherits C2 Then mark (C1, C2) by the label IS_A
   Else /* there are methods of C2 which are not in C1 */
      (i) Create a class C2' such as the structure of C2' is the one of
          C2 and the set of methods of C2' is the set of methods of
          C2 that are inherited by C1
          Mark (C1, C2') and (C2, C2') by the label IS_A
          Eliminate (C1, C2)
      (ii) For each k such as a link (C2, Ck) exists Do
             Change (C2, Ck) into (C2', Ck) keeping the same label
          End_For
   End_If
End_While.
```



The application of this algorithm to Figure 8 gives the inheritance graph of Figure 3. The class C3 can be further reorganized by the designer if there are many methods that are applied to students of academic cycle 1 (or academic cycle 2).

Remark 1: We can merge the above algorithm with the one mentioned in 4.2.1. They have been presented separately for reason of clarity.
Remark 2: If we accept the use of the masking concept in an inheritance graph, the above algorithm is reduced to the initialisation step.
Remark 3: In the general case, at this stage of conceptual design, class methods and instance methods are not distinguished. However, in an object-oreinted implementation and by taking into account the features of the OODBMS used,it may be necessary or convenient to modify the inheritance graph by creating classes to manage (by means of class methods) collections of instances.

5 Conclusion

We have defined a splitting mechanism that derives an inheritance hierarchy from a class. It is based on applicability constraints between characteristics of a class. It is decomposed into three phases. The first phase verifies the consistency of the given set of applicability constraints. The second phase deduces all the substructures included in a class. The third phase builds the classes of the inheritance hierarchy.

The proposed mechanism is a normalization process because it removes from classes all optional attributes that would generate null values in the object implementation stage. Furthermore, this mechanism is a mandatory pre-requisite to any factorization mechanism because those latter don't take optional attributes into account and so can generate "bad" inheritances. It derives specialization inheritances which are the only ones useful to deduce during the conceptual design stage, by opposition to the implementation inheritances. Finally, this mechanism removes all the specified existence constraints. They are expressed through the derived inheritance hierarchy. This is, at the present time, interesting because most of the existant OODBMS don't allow the definition of integrity constraints. So, the designer is obliged to integrate in the class methods some integrity controls. By splitting classes, we avoid the duplication, in methods, of controls related to existence constraints.

The splitting mechanism can be applied at the end of the conceptual design stage of a database with an object-oriented design, preferably with an object-oriented implementation in order to keep all the benefits of the inheritance concept. As mentioned earlier, the maximal decomposition obtained can be reviewed for optimization reasons.

It is also an interesting tool in the perspective of evolving from a traditional design, where data and operations are independent, to an object oriented design where data and operations are encapsulated.

Till now, the implementation of the mechanism is in progress on a Sun station and we intend to integrate it into a case tool [Kar95].
At the present time, we are studying the generalization of the mechanism to a schema including inheritance hierarchies. We are also studying the mean to extract applicability constraints from formal specifications like B or VDM.

References

[And92] Andonnof E., Sallaberry C. and Zurfluh G., *Interactive design of object oriented databases.* 4th International Conference CAISE'92, Manchester, may 1992.

[Atz83] Atzeni P. and Morfuni M., *Functional dependencies and existence constraints in databases relations with null values.* Information System Analysis Institute. Technical report n°R77, december 1983.

[Ber91] Bergstein P. et Lieberherr J., *Incremental Class Dictionary Learning and optimization.* Proceedings de ECOOP'91, Geneve, Suisse, July 91. Lecture Notes in Computer Science 512. Editor Pierre America Springer-verlag.

[Bla94] Blaha M., Premerlani W. et Shen H., *Converting OO Models into RDBMS Schema.* IEEE software Mai 1994. Editor IEEE Computer Society.

[Bou94] Bouzeghoub M., Gardarin G. and Valduriez P., *Du C++ à Merise Objet. Objets: Concepts, Langages, Bases de données, Methodes et interfaces.* Eyrolles, France, 1994.

[Cas93] Castellani X., *Mechanisms of Standardized Reusability of Objects (MCO methodology).* Proceedings de Working conference on Information system development process, IFIP WG 8.1. Como, Italia, 1-3 september 1993. Editor North-Holland.

[Cod90] Codd E. F., *The relational model for database management. Version 2.* Addison-Wesley Publishing Company, Inc., 1990.

[God93] Godin R. and Mili H., *Building and maintaining analysis-level class hierarchies using Galois Lattices.* OOPSLA'93. ACM SIGPLAN notices, volume 28, number 10. October 1993.

[Gol92] Goldstein R. and Storey V., *Unravelling Is_a Structures.* Information Systems Research Revue 3:2, June 1992.

[Hal91] Halpin T., *A fact-oriented approach to schema transformation.* Proc. MFDBS'91, Springer-Verlag lecture notes in Computer Science, n°495, Rostock, 1991.

[Kar95] Kara-Zaîtri N., Castellani X., *TCO: A Process and aTool to Map Communication Components of Data Flows to Objects.* 33th conference of ACM Southeast Conference. Clemenson, South Caroline. 17 and 18 march 1995.

[Kor95] Kornatzky Y., Shoval P., *Conceptual design of object-oriented database schemas using binary-relationship model.* Data & Knowledge Engineering revue, Volume 14, Number 3, February 1995. North Holland.

[Lal92] Laleau R.,Castellani X. and Jouve M., *Normalized design of the specialization inheritance.* Proceedings Indo-French Workshop on object-oriented systems. Goa, India, November 1992.

[Lam94a] Lammari N., Jouve M., Laleau R. and Castellani X., *Dépendances d'existence: définitions formelles et utilisation pour déterminer des liens*

d'heritages entre classes. CEDRIC-CNAM, Paris.Research report, may 1994.

[Lam94b] Lammari N., Jouve M., Laleau R. and Castellani X., *An algorithm for IS_A Hierarchy Derivation.* Proc. OOIS'94 (International Conference on Object-Oriented Information systems), Springer-Verlag editor, London, december 1994.

[Lie91] Lieberherr L., Bergstein P. et Silva-Lepe I., *From objects to classes: algorithms for optimal object-oriented design.* Software Engineering Revue 6:4, july 91.

[Lin92] Ling. T. W, Teo. P. K., *On conflicts in class hierarchies of object-oriented systems.* Department of information systems of the university of Singapore. Technical Report n°TR-C4-92, 1992.

[Mai80] Maier D., *Discarding the Universal Instance Assumption: Preliminary Results.* XP1 Workshop on relational database theory. Suny at Stony Brook, NY, June-July 1980.

[Mai83] Maier D., *The theory of relational databases.* Computer Science Press, Inc., America, 1983.

[Pic90] Pichat E. and Bodin R., *Ingénierie des données.* Masson, Paris, 1990.

[Thi93] Thieme C. et Siebes A., *Schema Integration in Object-Oriented Databases.* Proceedings of CAISE'93.

[Thi94] Thieme C. et Siebes A., *An approach to schema intagration based on transformations and behaviour.* Technical Report n°CS-R9403, january 94.

[Wei91] Wei G. and Teorey T. J., *The Orac model: a unified view of data abstractions.* Proc. of 10th Conference on Entity-relationship Approach, San Mateo, California, USA, october 1991.

[Wir90] Wirfs-Brock R., Wilkerson B. and Wiener L., Designing Object-Oriented Software. Prentice-Hall, Inc. USA 1990.