# Beyond OOP : (2) Inheritance for Multidimensional Objects

Henry J. Borron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Beyond OOP : (2) Inheritance for Multidimensional Objects*

Henry J. Borron

N° 3158
Avril 1997

THÈME 2


# Beyond OOP : (2) Inheritance for Multidimensional Objects

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N° 3158 — Avril 1997 — 24 pages

**Abstract.** Inheritance is an essential aspect of any object oriented system, be it reactive or not. The work summarized in this synthesis generalizes to multidimensional objects the most powerful technique presently known, i.e. inheritance and combination based on a preliminary linearization (cf. CLOS and languages alike). The report depicts the problems encountered as well as the proposed solutions which are meant to be sound and practical. Fundamentally, a linearization is done dimension by dimension (hence, N "lines"). For memory representations, a simple and usual constraint makes each representation to belong to a single line : each line being independent, the combination is easy. Concerning methods, the lines are interwining since methods may satisfy several dimensions. To simplify the obtained structure, a condition (verified in practice) is proposed and a rule (quite simple to apply) is used which transforms the interwining structure into an arborescent one. This being done, methods are easy to combine. Very sophisticated combinations (as in CLOS) are supported while ensuring a pure declarative style (contrary to CLOS) : the order of methods can be obtained from the headers only, without looking inside the method bodies for "send-super" statements.

The report also studies under what conditions linearizations are easy to predict (cognitive aspect) : with this objective in mind, a proposal is made which is highly efficient at the same time.

The proposition made here can be taken as a source of inspiration, acting as an upper limit for less automatic and sophisticated inheritance schemes. It can also be adapted to the realization of an object-oriented system centered on Harel's statecharts, an important point for the industry given the crucial applications of reactive systems

**Keywords.** Inheritance, masking, combination, multidimensional space, linearization, monotonicity, congruency, method qualifiers, meta-object protocols, state, transition, memory representation, pre-method, post-method, invocation diagram, pure declarativeness.

*(Résumé : tsvp)*

* borron@chris.inria.fr

INRIA, Unité de Recherche de Sophia-Antipolis
2004, route des Lucioles. BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 — Télécopie : (33) 93 65 77 65

# Au-delà de la POO :
# (2) Héritage pour des objets multidimensionnels.

**Résumé.** L'héritage est un aspect essentiel de tout système orienté objet, réactif ou non. Le travail résumé dans ce rapport de synthèse généralise à des objets multidimensionnels la technique la plus puissante actuellement connue, celle d'héritage et de combinaison avec linéarisation préalable (cf. CLOS et les langages voisins). Le rapport dépeint les problèmes rencontrés et les solutions proposées qui se veulent pratiques et saines. Fondamentalement, une linéarisation est faite dimension par dimension (N "lignes"). Pour les représentations en mémoire, une contrainte simple (et habituelle) fait que toute représentation n'appartient qu'à une seule ligne : chaque ligne est indépendante ; la combinaison est donc facile. Pour les méthodes, les lignes s'entrelacent parce que des méthodes peuvent satisfaire plusieurs dimensions. Pour simplifier la structure obtenue, une condition (respectée en pratique) est proposée ; une fois vérifiée, une règle (facile à appliquer) est utilisée qui transforme la structure entrelacée en une structure arborescente. Ceci fait, il est aisé de combiner les méthodes. Des combinaisons très sophistiquées (comme en CLOS) sont supportées tout en assurant un style purement déclaratif (à la différence de CLOS) : l'ordre des méthodes peut être obtenu seulement à partir des en-têtes, sans avoir à rechercher dans les corps de méthodes l'existence éventuelle d'ordres "send-super".

Le rapport étudie aussi sous quelles conditions les linéarisations sont faciles à prédire (apect cognitif) : avec cet objectif en tête, une proposition est faite qui s'avère en même temps très efficace.

La proposition faite ici peut être prise comme source d'inspiration pour des schémas d'héritage moins sophistiqués ou moins automatiques. Elle peut aussi être adaptée à la réalisation d'un système objet centré sur les diagrammes d'états-transitions de David Harel, point important pour l'industrie étant donné les applications cruciales des systèmes réactifs.

**Mots-clés.** Héritage, masquage, combinaison, espace multidimensionnel, linéarisation, monotonie, congruence, qualifieurs de méthodes, protocoles métaobjets, état, transition, représentation en mémoire, pré-méthode, post-méthode, diagramme d'invocation, aspect purement déclaratif.

# Beyond OOP
## (2) inheritance for multidimensional objects

H. J. Borron†

## 1. INTRODUCTION

Started in 1994, the reported work concerns a new form of programming which derives from OOP, but which is about as far from traditional OOP than was OOP from structured programming. It models the behavior of an object using states and transitions in a multidimensional space. This converges with the objective of making statecharts the heart of an object-oriented system [8][1].

### 1.1) A Question

This paper is part of a new approach that pushes further the basic idea of OOP which is —in our view— to give responsibility to objects. Given this objective and this view, objects should be given more responsibility. In other words, the behavior of an object should depend not only on its class (as in traditional OOP), but also on its own state. States are described in a multidimensional space as well as the transitions between these states : a transition represents the existence and effect of a message (single dispatch) or generic function call (multiple dispatch). The obtained graph represents the whole behavior of a class of objects. This graph may be augmented with implementation items (memory representations, pre- and post-methods) : this way a flat implementation may be defined. Yet, because objects belonging to different classes may share a common behavior, abstract implementations may be defined. These consists in building a new class graph using existing ones. Two operators are defined in this purpose : the composition -which composes the graphs of independent base classes (superclasses) ; and the derivation -which alters a base graph with a mixin one for a particular instance state. Using these operators, a hierarchy of class graphs gets developed.

Given this context, a question appears (with numerous subquestions). An important concept of OOP is class inheritance. Does this concept mesh well with the proposed approach ? Is it useful ? Is it possible to say that a transition is inherited from a superclass or an ancestor graph (specification level) ? Is class inheritance useful for building implementations ? How does a memory representation is built from "inherited" memory representations if such an inheritance appears possible ? Same type of question for methods. Are they any constraints, drawbacks, advantages ? ... To summarize, can the class inheritance concept be re-built (from the OOP inheritance concept) when a multidimensonal space is considered ?

### 1.2) Plan of next sections

First, the quick presentation of our proposal just done above (cf. subsection 1.1) is made more precise (section 2). Then the inheritance of transitions is examined (section 3). It is shown to be closely related to the local inheritance of transitions ("local" means inside a same class graph). An example is set up (section 4) and is used for unveiling the basic principle of class inheritance for implementation items (section 5). The class inheritance of memory representations is considered (section 6). Then, the class inheritance of methods is examined in details, first in case of single dispatch, then in case of multiple dispatch (section 7). The combination of inherited methods is considered : it is shown to be purely declarative and possibly quite sophisticated (section 8). Finally, since linearization is used in our scheme, the question of making linearizations easy to predict is examined (section 9).

## 2. FORMALISM

To allow this paper to be read independently of its companion paper, we start by a brief description of the formalism we use for modelling the behavior of a class of objects.

---

† Inria Sophia-Antipolis, 2004 route des Lucioles, 06560 Valbonne, France. (borron@chris.inria.fr).

[1] Part of the work summarized in the companion paper (potential and mixin concepts) and in this one (class inheritance mechanism) is proposed to be a basis for an international cooperation aiming at Harel's foreseen system. Following our meeting on november 25 th 1996, this paper and its companion paper were sent to David Harel on december 5th 1996.

## 2.1 Class graphs

We previously introduced a formalism for modelling the whole behavior of a class of objects in a multidimensional space (see companion paper). A "**class graph**" was used in this purpose (one graph per class). Such a graph describes all possible **instance states** and all possible **regular transitions** between these states.

Regular transitions model the existence and effect of external events, i.e. of messages (single dispatch) or generic function calls (multiple dispatch). **Nodes** in a graph depict instance states or contributions to instance states. They are associated a **condition** (either implicitly or explicitly) and may be given a name. Next figure illustrates a graph transition between two nodes.

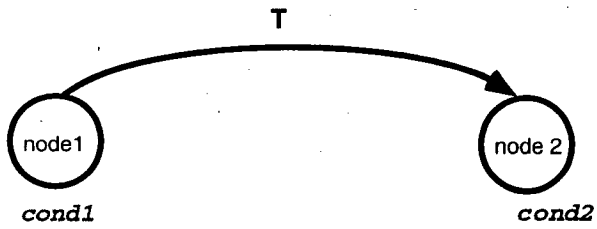
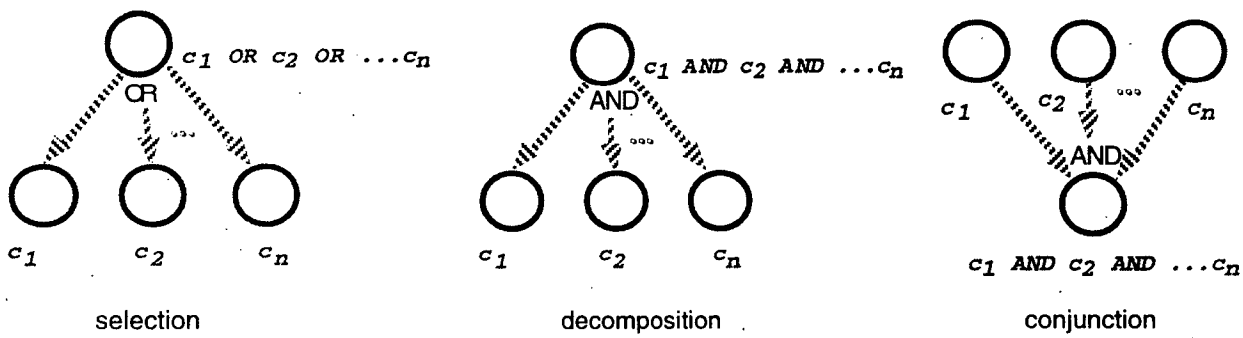



Figure 1.

A class graph is **structured** : nodes are associated into three types of **constructs** : the **decomposition**, the **selection** and the **conjunction**, the latter one being optional. Each construct is a one-level AND or OR-tree, either converging or diverging. The conjunction is a AND converging construct : the condition of its destination node ANDs the conditions of its two or more source nodes ; the selection (resp. decomposition) is a diverging OR (resp. AND) construct : the condition of its source node ORs (resp. ANDs) the conditions of its two or more destinations nodes. (NB : in terms of conditions, the destinations of a selection must form a partition of its source.)

The dynamic semantics of these three constructs may be expressed using a more basic element, the **reflex transition**. A reflex transition exists between each pair (source node, destination node) of a construct. (A reflex transition cannot exist by itself : it should be part of a construct.) **Armed** when its source reflects the current instance state or part of it, a reflex transition **fires** when the condition of its destination node becomes true. (This contrasts with the firing of regular transitions : these do not fire on an internal condition but because of an external event.)
Hence, when the source of a selection participates to the description of the instance state, one and only of the reflex transitions of the selection fires : a selection expresses thus a choice (it corresponds to an IF-THEN-ELSE). In the same initial situation, all the reflex transitions of a decomposition fire : a decomposition splits a state into substates (one substate per dimension). A conjunction also presents this property (simultaneous firing of all its reflex transitions), yet in the opposite way : a conjunction groups substates. Next three figures illustrate the concepts of selection, decomposition and conjunction.




Figures 2, 3 & 4.

Reflex transitions are important for a second reason : they also are the support for what we term **local inheritance**, i.e. inheritance inside one graph - a form of inheritance which is to be distinguished from class inheritance (the very subject of this report). Local inheritance exists at the **specification level** : regular transitions are inherited along reflex transitions except those of a decomposition (in the next figure, the transition *T* in *s* is inherited in *a* : the *T* transition in *s* is termed a **factorized** transition). Local inheritance also exists at the **implementation level** : implementation items are also inherited along the same reflex transitions. In other words, **memory representations**[2] (associated to nodes) and methods (**pre- and post-methods** associated to transitions) are inherited under the same conditions than regular transitions (in the next figure, the memory representation in *s* is inherited in *a* ; the pre-method *T* and the post-method *U* in *s* are also inherited in *a*).

2 A memory representation is a set of cells (instance-variables in Smalltalk, slots in CLOS).

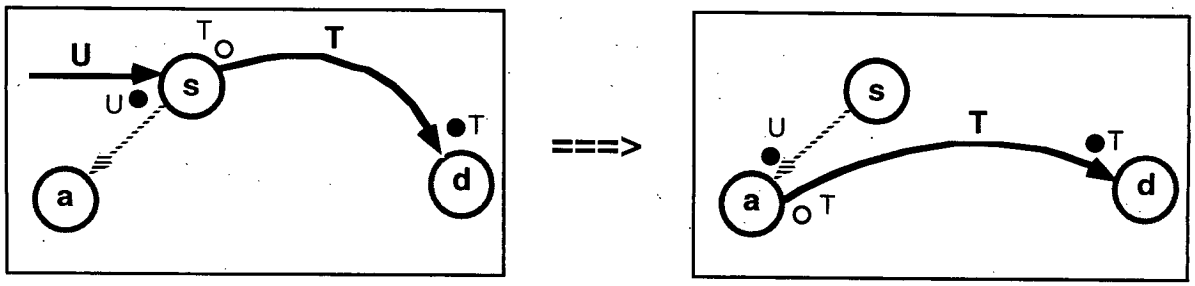



Figure 5.

As announced, this modelling is done in a multidimensional space. For example, a *Circle* instance state will be modelled according to a *radius* dimension and a *center* dimension. A state transition is understood as resulting from its projections (graph transitions) along the space dimensions. For example, the circular *draw* transition for a *Circle* instance depends on both the *radius* and *center* dimension. Valid only for a *fully-initialized* instance, it can be specified as being attached to this state only or as by its decomposition in two **constrained** circular graph transitions, each one being attached to one substate (resp. *radius-initialized, center-initialized*) and constrained by the other substate (resp. *center-initialized, radius-initialized*). This constraint is termed a **clause**. Circular graph transitions need not be specified (as long as at least one graph transition is specified), the specification of only one or the other constrained circular graph transition suffices. Because the dimensions are orthogonal, quite a number of state transitions depends only on one dimension (or, more generally, do not depend on all the space dimensions). For example, the *surface* transition for a *Circle* instance depends only on the *radius* dimension (no clause). From this analysis, it comes that the source and destination nodes of a graph transition should involved exactly the same dimensions.

A few constraint apply also to constructs. Concerning a selection, the same dimension(s) should be involved at the source and at each destination. Concerning a conjunction (resp. a decomposition), each involved substate should belong to a different dimension. A decomposition involves all the dimensions.

The current state of an instance is **marked** by one or several **tokens** (one per dimension). These tokens move along regular and reflex transitions when these fire. Each token may be associated a **potential** which is modularly updated and consulted by regular transitions (including testing transitions for conditions). It gives regular transitions an abstract body and, hence, enables checking and simulation dimension by dimension (modularity).

## 2.2 Hierarchy of class graphs

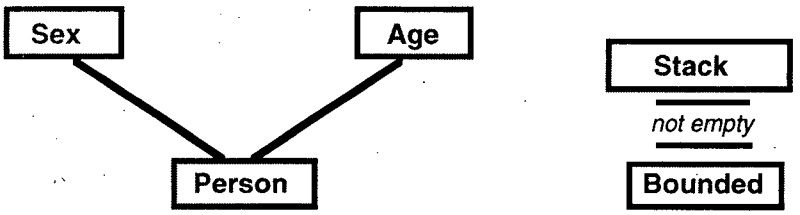
A class graph describes the full behavior of the instances of a class, be this class implemented in a flat way or thanks to a complex class hierarchy. Two operators are provided to build a class graph out of superclass graphs (**abstract implementation**).

The first one, termed the **composition**, directly derives from the decomposition construct. (The operator somehow promotes the construct from the local level to the hierarchy level.) It composes the dimensions of two or more base classes $C_1$, $C_2$,... (of degree $N_1$, $N_2$,...) into a ($N_1$+$N_2$+...) dimensions space. (The **degree** of a class is the number of its dimensions.) A local **increment** possibly with its own dimensions may also be added. We say that the composed class **inherit** from its base classes, the superclasses. The order in which the superclasses are listed is important if a **masking** effect is sought. **Renaming** of nodes and transitions is possible if necessary.

The second operator, termed the **derivation**, corresponds to a constrained form of composition ; it enables a base class to be extended with (a) **mixin** class(es) in a checkable and very modular way. The mixin is itself a multi-level selection with one or several INITial nodes. Its specification lists the required transitions and their required form at the hook node of a base graph : **CIR**cular (consulting or modifying), purely **OUT**coming or **IMP**ure ones (a path of reflex transitions exist between the destination node and the hook node). The specification also lists an **attachment condition** the base graph should satisfy in the hook node. An **initialization transition** for modularly supporting the concept of potential inside the mixin may also be provided. Concerning the regular transitions inside a mixin, simplification rules exist that enable the user to specify only the transitions differing from what is normally expected from the specifications of CIR, IMP and OUT transitions, and from the existence of INIT nodes. Our proposal also supports **parameterized mixins** for still more modularity and efficiency.

Next two figures respectively abstract a composition and a derivation : all details are rubbed out. What we only know is that the class *Person* is composed from the class *Sex* and *Age* (first figure) and that a derived class of *Stack* is produced by attaching the mixin *Bounded* (second figure) in the hook node *not empty* of the base class *Stack*.[3] (These examples were described in details in the companion paper.)

[3] The hook node may be not mentioned.




Figures 6 & 7.

## 3. MULTIDIMENSIONAL INHERITANCE OF TRANSITIONS IN A CLASS HIERARCHY

Inheritance in a hierarchy of class graphs is termed **class inheritance**. It is different from local inheritance, i.e. inheritance in a same class graph. Yet, it is related to.

Given that the composition operator corresponds in fact to the decomposition construct, it should not be surprising that class inheritance rules for transitions extend the local inheritance rules for transitions. As a matter of fact, class inheritance rules are a different way to apprehend the effects of the composition and derivation operators (and to take care of the local inheritance rules as well).

Given an instance state, the transitions that are valid in this state are searched dimension per dimension, first locally (i.e. according to the local inheritance rules), then in the graphs attached to the superclasses of the instance class (in the order defined by the list of superclasses). The algorithm is recursive (the superclasses of these superclasses may also be searched, etc.). If masking is allowed, a successful search prevents recursion. The obtained transitions can be thought as being locally attached to the considered class graph : in this case, the transition destinations are obtained exactly like in a regular class graph. (Alternatively, one may prefer to consider the transitions to be still attached to the ancestor graphs : in this case, before the backtracking and the propagation steps, the moves of the tokens in the composed graph are driven by the moves of marks in the ancestor graphs. )

As far as inheritance is concerned, the basic rules for class inheritance simply needs to be adapted to transition renaming and masking. Both are done dynamically. Let's suppose a message M is sent. Renaming is obtained by adopting the old name in place of the new name on entering the superclass given in the renaming declaration : the old name is considered inside the whole hierarchy of this class (except if it is itself the result of an inner renaming inside this same hierarchy ) ; it is abandoned on quitting it. Masking is obtained by abandoning the search once leaving the hierarchy of the first superclass where M is declared as masking (by construction, a M transition exists in this superclass, either inherited or locally defined). An example is given below (in subsection 4.1, to be precise.)

Given a class, each local description that introduces a new dimension may be materialized as a **virtual superclass**. A class is thus modelled as being made of several superclasses (virtual ones included) plus a **virtual subclass** that groups the refinements a class introduces vs. its superclasses. Considering the modelling of the local increment as a virtual superclass, the important point here is to consider this virtual superclass before all other superclasses so as correctly take into account masking -if transitions defined in the increment are (explicitly or implicitly) specified as such.

Because the derivation is a special form of composition, the algorithm which extends local inheritance to class inheritance works accordingly well for a derived class. We simply want the definitions made in the mixin graph (ex. : *Bounded*) may refine similar definitions at the hook node of the base graph (ex. : *Stack*) when the mixin participates to the description of the instance state : (1) the mixin graph is visited before the base one ; (2) the transitions defined in the mixin graph are implicitly declared as masking ; (3) the validity of the mixin graph is limited (the base mark should be in the hook node). First two rules make inheritance in this case not especially particular. Last rule is obtained by construction, the mixin graph being a constrained subtree whose clause is precisely the hook (see companion paper). The declaration of the mixin class before the base class in the list of superclasses is easy to check and to correct : the underlying system knows which is a mixin and which is a base.

Next figure abstracts the relationship between the three classes as a small hierarchy, like in OOP : note this manner is quite abrupt since it omits to mention the hook node, an important detail in fact.

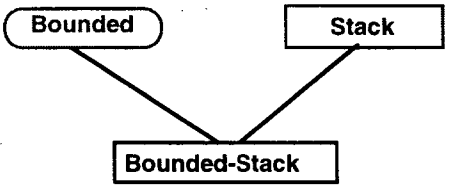



Figure 8.

For more details about all these points, refer to the detailed report [4]. Before focusing on the inheritance of implementation items (memory representations, pre- and post-methods), let's first define an example.

# 4 A CLASS INHERITANCE EXAMPLE

## 4.1 The Example

The example we consider here is a bit more complex than the *Person* one shown in the companion paper. It illustrates masking and renaming facilities. It deals with the composition of a particular type of objects behaving both like a *Stack* instance and like a *Queue* instance. One can *push* an element on its front or *enqueue* it at the rear ; however, elements are only obtained from the front using a *pop*. The specification is to first *pop* all the elements that have been pushed and only afterwards the elements that have been enqueued. Let's name *STQ* the class of such objects. This class is composed using the *Stack* and *Queue* classes as seeds. The *Stack* and *Queue* classes are themselves built from the *Bag* class while the *Bag* class derives from the *Object* class. Hence, the figure.

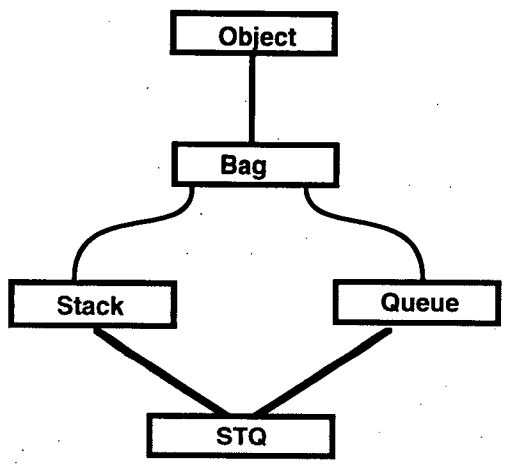



Figure 9.

The obtained figure is not different from what would have been drawn in traditional OOP. Yet, there is a difference : dimensions. *Object* features one dimension (noted *object*) ; *Bag* has two dimensions : one inherited from *Object* and one local (noted *bag*) ; *Stack* and *Queue* have three dimensions : two inherited (from *Object* and *Bag*) and one local (respectively *LIFO* and *FIFO*) ; *STQ* inherits all these dimensions and does not introduce a new one. Next figure refines the previous drawing. Each oblique line represents a different dimension. (Not all dimensions are named : the name of a dimension only appears if it differs from the name of the class that introduces it.)

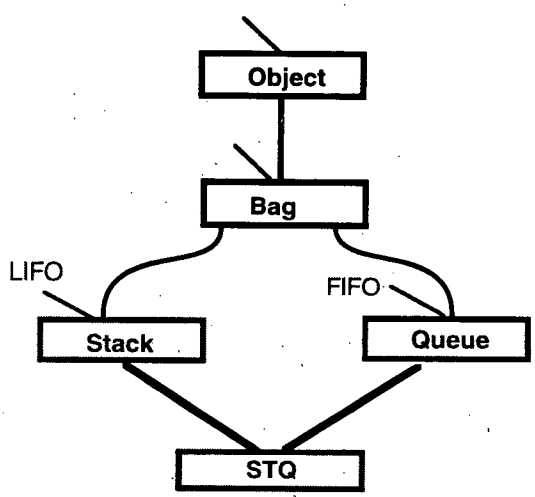



Figure 10.

Let's be a bit more precise. The *object*, *bag*, *LIFO* and *FIFO* dimensions all appear as a local increment in the declaration of the *Object*, *Bag*, *Stack* and *Queue* class graphs[4] :

[4] The declaration of LIFO and LIFO, interesting from a modelling point of view, is not mandatory in practice.

— the *object* dimension is declared as being **abstract** : no memory representation can be associated to it. This dimension exhibits but a unique node (the condition of which is *t*, i.e. systematically true). These conditions (abstraction, unicity) correspond to the declaration of a **property** (the node may be declared as such instead of mentioning the *t* condition and the *:abstract* keyword). For the purpose of the example, we consider that the *Object* class only introduces the *print* transition ;

— the *bag* dimension is introduced by *Bag*. via a local increment composed of a selection with the *put:* and *get* regular transitions. The empty transition is inferred from the conditions at the selection destination nodes ;

— the *LIFO* dimension is abstract and exhibits a single node with the *t* condition : it is thus also declared as a property (like *object*). A comment is worth to be added here. Typically, a local increment —like the *Bag* one— is composed of a few nodes with transitions between them. Yet, in some cases, the increment properties cannot be formalized as simply : in this case, they are identified using **property** nodes. This idea is adapted from the property identifiers introduced by P. America ([1], p. 71). The *LIFO* organization of a *Stack* is such a kind of property ;

— the *FIFO* dimension of *Queue* deserves the same type of comments. The *LIFO* and *FIFO* properties model the fact that the *Stack* graph and the *Queue* graph are different from the *Bag* graph and different from each other too, even if they all result in a topologically identical graph.
The *STQ* class is composed using the *Stack* and *Queue* classes as seeds. *Stack* is before *Queue* in the superclass list of *STQ* and this order is taken advantage of for **masking** : when some elements have been pushed and some have been enqueued, the *pop* transition inherited from *Queue* is to be masked by the *pop* transition inherited from *Stack* . The *pop* transition inherited from *Queue* corresponds in fact to the *dequeue* transition of *Queue*, but *STQ* renames it as being *pop* (**renaming**). As a matter of fact, *Bag* provides a *get* transition and *Stack* (resp. *Queue*) renames *get* into *pop* (resp. *dequeue*). Suppose now a *pop* transition is searched in *STQ* : *pop* will not be found in *STQ*, nor in *Stack*, but in *Bag* : when entering *Bag*, *get* will be searched in place of *pop* and will be found. Since the *pop* transition of *Stack* is declared as masking in *STQ*, *pop* will not be searched in *Queue*.

The hierarchy is shown in the **natural ordering of specialization** (placing *Queue* on the left and *Stack* on the right would appear unnatural since *Stack* is before *Queue* in the superclass list of *STQ*). A new dimension appears before the inherited dimensions.

For more details, report to the appendix : it describes the *STQ*, *Stack*, *Queue*, *Bag* and *Object* class graphs.

## 4.2 Interpretation in the state space

By default, the dimensions *object* and *bag* are unique. This is easy to interpret in the state space : it simply means the corresponding axes exist but once. The state space has four dimensions (axes) : *LIFO*, *FIFO*, *Bag*, *Object*. Next figure illustrates this : a given state of an *STQ* instance (say $stq_n$) is projected onto the four axes of the state space. The order of the axes is important in case masking is used. Hence, the question : what is the order of the dimensions ?

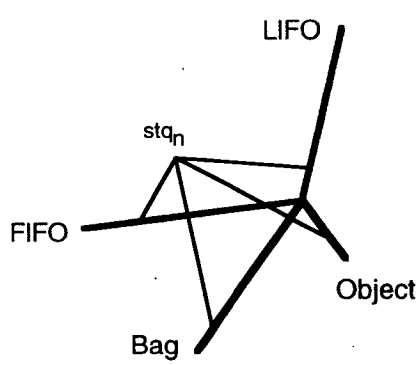



Figure 11.

A second question is : what classes **impact** the four dimensions ? The *LIFO* and *FIFO* dimension paths both traverse *STQ*, and each one respectively traverses either *Stack* or *Queue*. The *bag* dimension path traverses *Bag*, *Stack*, *Queue* and *STQ*. The *object* path traverses the same classes plus the *Object* class.

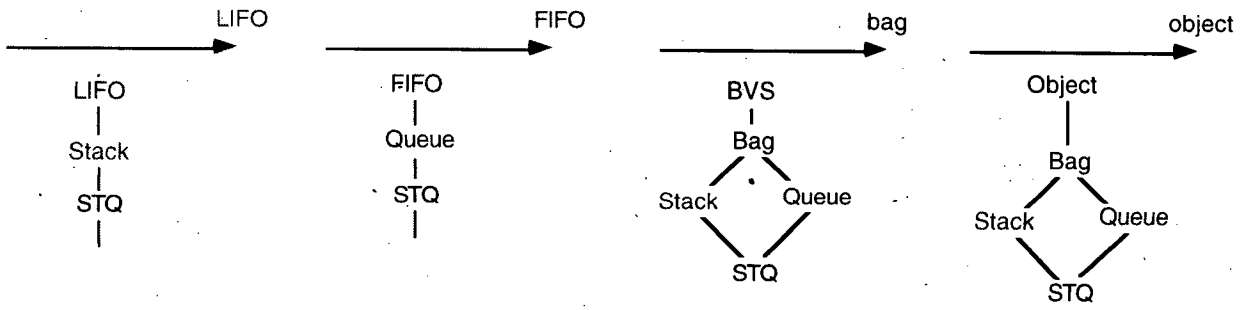



Figure 12.

The above figure shows the four dimensions (axes) together with the classes that impact them. Taking a theoretical point of view, it also mentions *LIFO*, *FIFO* and *BVS* which are virtual superclasses. (*BVS* is an acronym for *Bag* virtual superclass). Figure 10 may also be interpreted in this way : except for *Object*, the oblique line represents both the dimension and the virtual superclass ; concerning *Object*, it represents but the *object* dimension (*Object* introduces this dimension by itself : no virtual superclass is needed.) In figures 10 and 12, virtual subclasses are named like actual classes.

## 5. MULTIDIMENSIONAL INHERITANCE OF IMPLEM. ITEMS IN A CLASS HIERARCHY

Let's focus on the class inheritance of implementation items (memory representations, pre- and post-methods). For the moment, we define a general principle which, as shown in [4], acts as an extension of the local inheritance rules for implementation items.

### 5.1 Principle

Our proposal basically consists in linearizing each hierarchy made by the classes that impact a dimension : methods and memory representations are ordered along the obtained lists of classes and then combined. We will see below that a condition and a rule will be defined that will slightly modify this principle. Yet, the ones who know LISP-based OO languages (ex. : Flavors, LOOPS, CLOS,...) will recognize a generalization of the linearization principle of these languages. The major difference lies in the fact that several "lines" of inheritance exist (one per dimension) instead of a single one.

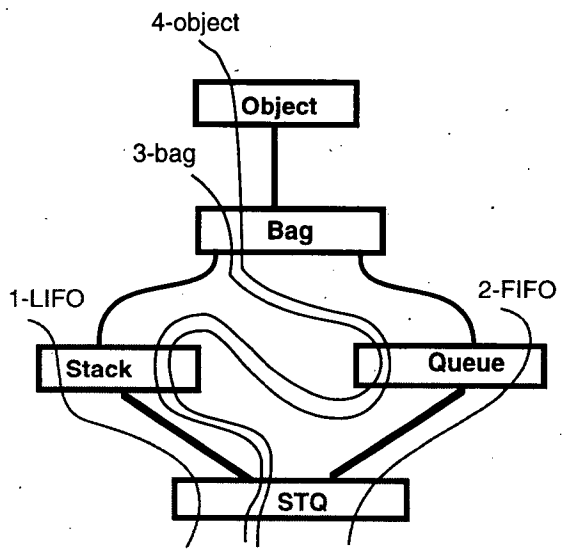



Figure 13.

The linearization principle solves the question : "What is the relative order of *Stack* and *Queue* in the *object* and *bag* paths ?". Our example is chosen so that any linearization algorithm will produce the same relative order for the considered classes[5]. The relative order of any two classes both impacting two same dimensions (ex. : the *Stack* and *Queue* classes vs. the *bag* and *object* dimensions) also happens to be the same. But this is not the general case. This sort of problems will be analysed in subsequent sections. Next figure shows the order of classes for each dimension.

[5] The traversing algorithms proposed for Flavors, LOOPS, CLOS may all differ in their results when a different hierarchy is considered. More recently, a still different algorithm has been proposed which has the interesting property of being monotonic [7]. We also recently devised different monotonic algorithms [5] [5b].

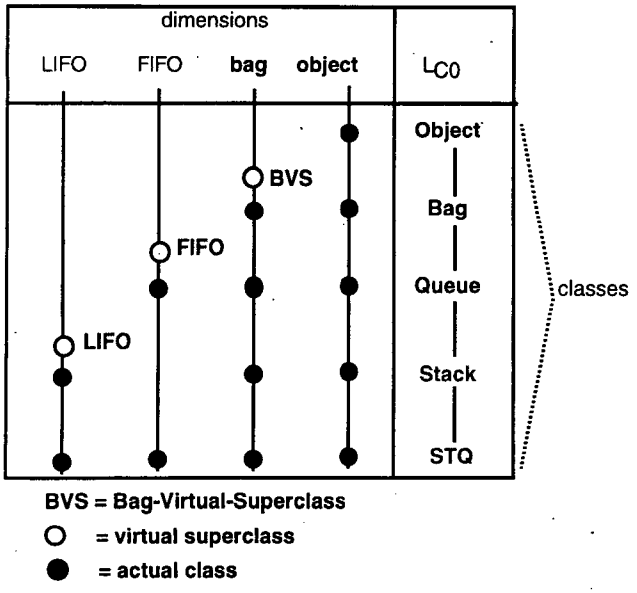



Figure 14.

## 5.2 A few notations

Let's name **subhierarchy** $H_{C0}(C)$ a class hierarchy rooted in *C0* and with summit *C*. This subhierarchy is of interest when it introduces one or several dimensions in *C*. For example, the subhierarchy $H_{STQ}(Stack)$ introduces the *LIFO* dimension in *Stack* : it is made of the *STQ* and *Stack* classes. Another example is the *Bag* hierarchy which introduces the *bag* dimension in *Bag* : it is made of the *STQ*, *Stack*, *Queue* and *Bag* classes. (In both examples, we implicitly includes the virtual superclass.)

The linearization of $H_{C0}(C)$ will be noted $L_{C0}(C)$. In our example, we have : $L_{STQ}(Stack)$ = *(STQ Stack)* ; $L_{STQ}(Queue)$ = *(STQ Queue)* ; $L_{STQ}(Bag)$ = *(STQ Stack Queue Bag)* ; $L_{STQ}(Object)$ = *(STQ Stack Queue Bag Object)*.

The linearization of classes for a given dimension, say *LC0(d)*, is easy to derive from that. (Here, we introduce the distinction between the virtual superclass(es) and the virtual subclass of a class.) $L_{STQ}(LIFO)$ = *(STQ Stack LIFO)* ; $L_{STQ}(FIFO)$ = *(STQ Queue FIFO)* ; $L_{STQ}(bag)$ = *(STQ Stack Queue Bag) BVS* ; $L_{STQ}(Object)$ = *(STQ Stack Queue Bag Object)*.

## 5.3 The order of dimensions

In above subsection 4.2, a question was raised : "What is the order of dimensions ?". In above figure 13, the number preceding the name of each dimension gives its order. (For example, *bag* is ranked third.)

The following constraints apply :
$\alpha$ : in *STQ*, the *Stack* superclass is listed before the *Queue* superclass, thus *LIFO* precedes *FIFO* ;
$\beta$ : in *Stack*, *LIFO* (new dimension) is before *bag* and *object* (inherited dimensions) ;
$\gamma$ : in *Queue*, *FIFO* (new dimension) is before *bag* and *object* (inherited dimensions) ;
$\delta$ : in *Bag*, *bag* (new dimension) is before *object* (inherited dimension).

Next figure shows the corresponding **precedence graph**. Recursively taking the node to which no arrow points determines the order of dimensions : *LIFO*, *FIFO*, *bag*; *object*.

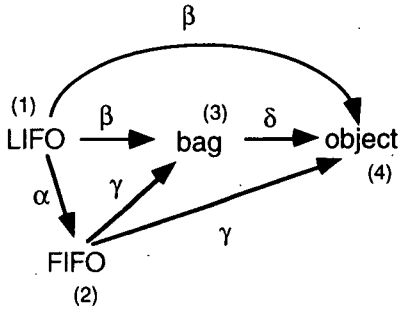



Figure 15.

## 6. INHERITING MEMORY REPRESENTATIONS

As expressed in the companion paper, a memory representation can be specified for each state (in a unidimensional graph) or basic substate (in a multidimensional graph). Thus memory representations may be made specific in a much more easy way than in traditional OOP. For example, a specific memory representation could possibly be chosen for a *Stack* instance when *empty* (since no elements need to be held, no cell is necessary) and an other one when *not empty*. These specific memory representations may be changed independently. Change of representations are done automatically when necessary. This malleability can be taken advantage of to progressively pass from simple data-structures to more complex ones yet still keeping the same functionalities : this can be used, for example, when bootstrapping the underlying system itself, or for making an application smoothly evolve from a poorly efficient implementation (ex. : prototype) to an efficient one.

Class inheritance for memory representations is quite simple. Given an instance of class *C0* in a certain state, it is done on a per dimension basis. For each dimension *d*, a class-precedence-list[6] (cpl, for short) exists : it is obtained by linearizing the list of classes belonging to the subhierarchy $H_{C0(d)}$ that impact the dimension in question (classes in this list are normally ordered from most specific to least specific). The memory representation for the considered instance of *C0* is obtained by an appropriate combination of the specifications of memory representations collected along the cpl of each **involved dimension**. Abstract dimensions like *object* of class *Object*, *bounded* of mixin *Bounded* or *LIFO* of class *Stack*, are not involved. The combination may be particular to a language. It may also be changed via a meta-object protocol [10]. Masking is also possible.

This proposal thus extends in a simple way what is done in CLOS. In this language, the specifications of memory representations are listed according to a single cpl, the class-precedence-list of *C0* obtained by linearizing the hierarchy of classes rooted in *C0*.

Next figure illustrates our proposal. It is supposed to be drawn for a *C0* instance in a given state. Two dimensions *dx* and *dy* are supposed to be involved for the considered state of the instance. Each line corresponds to a dimension. The classes are represented by ovals. The cpl for a given dimension is thus represented by the classes traversed by the line representing that dimension. When a memory representation is declared for a given dimension in a class, it is represented by a small grey square in that class, just above the line of the considered dimension. The memory representation thus combines the memory representations obtained for *dx* and for *dy*. The memory representation for *dx* may itself combine the two specifications that are found (as done in CLOS). These two combinations are different.

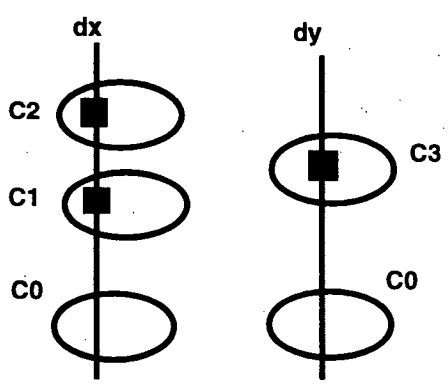



Figure 16.

A diagram of this sort is obtained when considering that *C0* is the *Circle* class. Then, *dx* and *dy* are the *radius* and the *center* dimensions. Let's consider a fully initialized *Circle* instance. In case of a flat implementation (no class inheritance), the local inheritance rules will make a memory representation to be searched in the augmented *Circle* graph for the *radius* and the *center* dimension. The two lines in the above figure exist but each one traverses only the *Circle* class. The obtained results (a cell for *radius*, a cell for *center*) will be combined (the default combination method is here the concatenation).
Let's now suppose that *center* dimension is introduced by the superclass *Centered-Object*. Then, we may have *LCircle(Centered-Object) = (Circle Centered-Object)* and *LCircle(Circle) = (Circle)* from which we respectively infer *LCircle(center)* and *LCircle(radius)*. Supposing, for the *center* dimension, that a memory representation has been specified in *Centered-Object* and none in *Circle*, then the memory representation for the *center* dimension will be found in *Centered-Object*. The memory representation for the *radius* dimension will still be found locally (in *Circle*). Same results and same combination as before. This makes clear that class inheritance rules extend local inheritances rules.

6 Here, we use the CLOS vocabulary. Note a cpl exists for each dimension (in CLOS, only one exists per class).

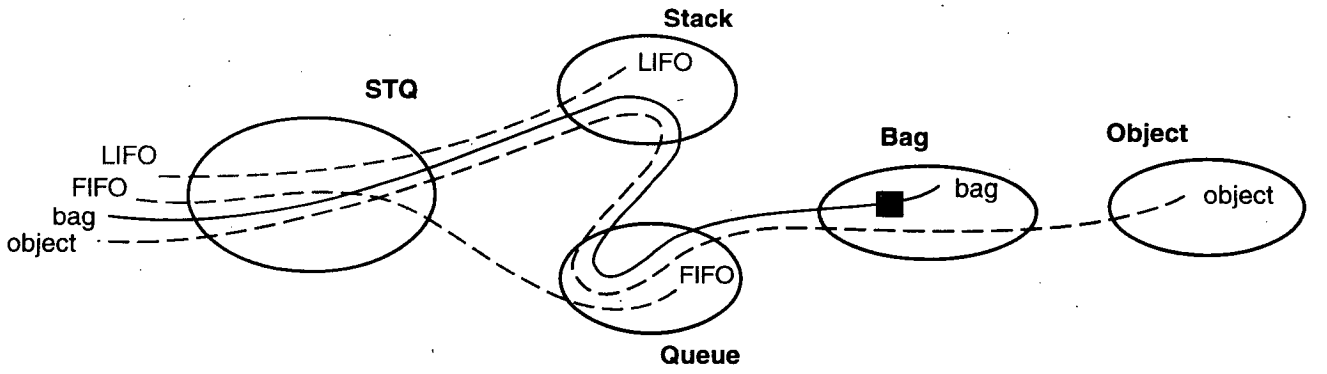



Figure 16.

Let's turn to our *STQ* example. The above figure shows four lines, each one corresponding to a different dimension. Abstract dimensions are shown as dashed lines. To simplify the drawing, the virtual superclasses are not shown. The figure supposes that the implementation provided by *Bag* (*elements* cell) is not refined in subclasses. This implementation is the one inherited in *STQ* when following the *bag* path.

The inheritance scheme for memory representations is simple. It comes from the fact that no memory representation may be defined for nodes resulting from a conjunction, a rule stated at the local level (in a single class graph) and which is kept when considering a class hierarchy. For methods, more flexibility is required and the corresponding inheritance scheme is thus a bit more complex as shown below.

## 7. INHERITING METHODS

In this part, we do not consider the existence of qualifiers like **before:**, **after:**, ... (the next section takes care of them). Once again, the existence of dimensions enables a high level of modularity. For pedagogical purpose, we progressively address each level of complexity (single dispatch, multiple dispatch) and make a comparison with what is done or could be done in OOP (CLOS, an OOP language which already attains a high degree of modularity, is again our reference).

### 7.1 Single Dispatch

#### a) in OOP

Let's suppose a message *m* is issued. Be *C0* the class of the receiver. Methods are listed according to the cpl of *C0*. (The way they are combined is expressed in the next section.) Next figure illustrates this case in a way quite similar to what was done above for memory representations.

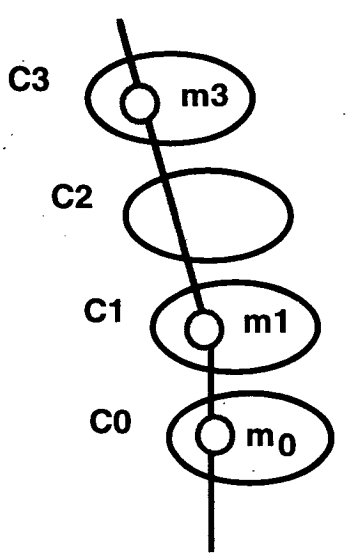



Figure 17.

## b) our proposal

When a message *m* is sent to an object of class *C0*, valid methods also depend on the receiver's state. The search is made according to the cpl of each involved dimension. Here, a dimension is **involved** when the *m* transition is defined along this dimension. (For example, for a *Person* instance, both dimensions are involved when searching methods named *long-lived* : *sex*, to determine the *expected-lifespan* ; *age*, to test whether the *age* is greater than the *expected-lifespan*) The result is an **invocation sequence diagram** listing methods along all involved dimensions and possibly along other dimensions that are **employed** (by the *m* methods) but not involved (by the *m* transitions). (For example, the *print* transition is defined for the sole *object* dimension, but the *print* methods generally use all the dimensions.)

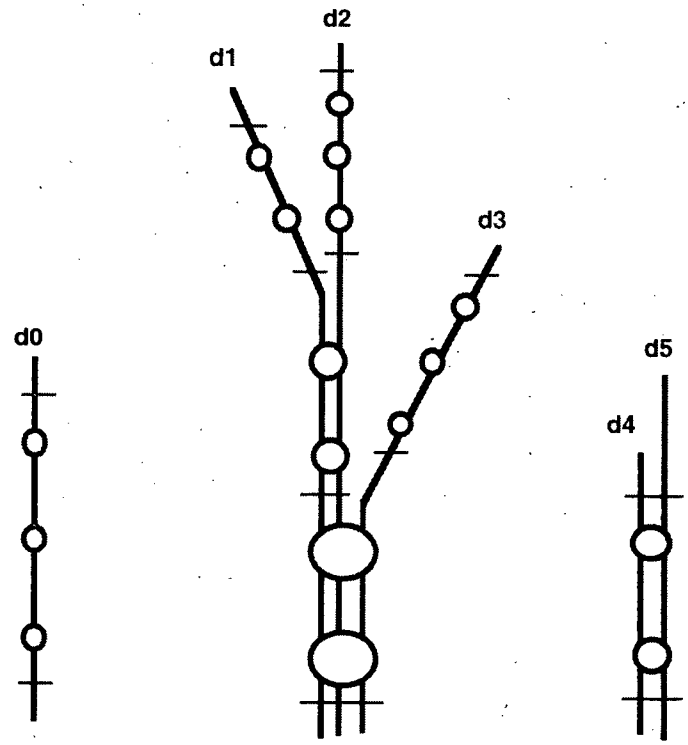



Figure 18.

If a certain condition holds and if a certain rule is applied, this invocation sequence diagram is a list of trees. Each tree is due to one or several dimensions that are initially parallel (they traverse the same classes), then progressively diverge. (See figure.)

### b.1) the condition

This condition is termed **regularity** . A hierarchy is regular vs. its methods named *m*, if all *m* methods of degree $\geq K$ that exist along a same dimension do satisfy the same *K* dimensions [4]. The **degree** of a method is the number of its dimensions. For example, the *draw* method for a *Circle* instance (a fully initialized one) is two : both the *center* and *radius* dimensions are required Regularity vs. methods, although not automatic, is obtained in practice.

Next two figures respectively show a case of NON regularity and a case of regularity. In these figures, a method *mxy* is a method that effectively requires both *dx* and *dy* dimensions (as does the *draw* method in our *Circle* example).

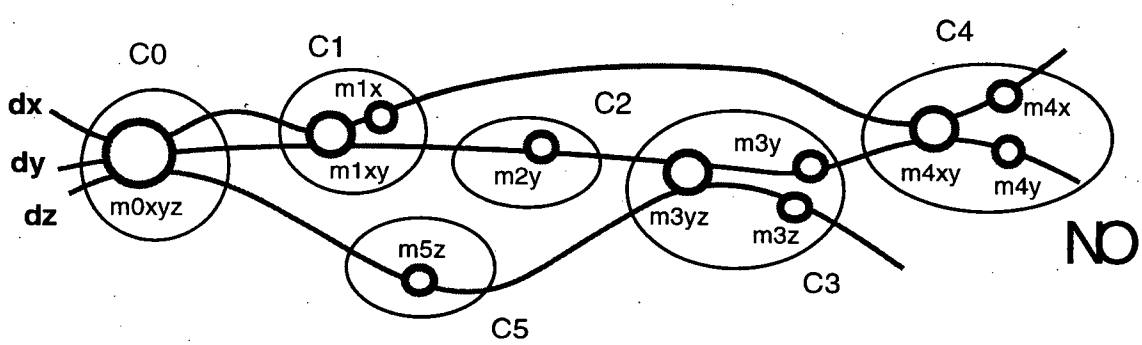



Figure 19.

In the above figure, regularity is not obtained : along the *dy* dimension, the methods of degree 2 are *m1xy*, *m4xy* and *m3yz*. The first two both satisfy the *dx* and *dy* dimensions, but the third one satisfies the *dy* and *dz* dimensions. Hence, three dimensions *dx*, *dy* and *dz* instead of two. In the next figure, the condition of regularity is always observed.

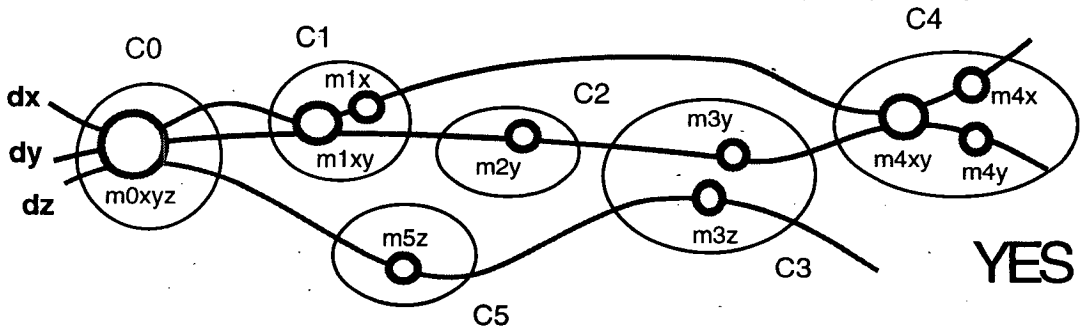



Figure 20.

Figure 19 shows an example of interleaving dimensions. Regularity does not exist for this hierarchy. Yet, the coexistence of methods enabling such a topology is however questionable in practical situations. It is difficult to imagine that two dimensions that are employed to implement a method *m* of class *C* may be considered apart in a subclass of *C* for implementing the same *m* method. If we were to *draw* a *Cylinder* instance, the *radius* and *center* dimensions will remain associated as they were in a *Circle*

Absence of regularity thus normally appears as a consequence of the linearization. However, it seems that all potential problems disappear in fact due to explicit masking (this one being set up by the user for semantic reasons). The *STQ* hierarchy provides two examples that exactly follow this pattern :

— the *STQ* hierarchy is a priori not regular vs. the possible *print* methods (see next figure). Two *print* methods, one inherited from *Stack* (*LIFO*, *bag* and *object* dimensions), one from *Queue* (*FIFO*, *bag* and *object* dimensions), do induce interleaving dimensions after linearization. Yet, the problem is naturally eliminated by the user : since the *STQ* instance is not to be *print*ed twice —once as a *Stack* and once as a *Queue*— but only once, explicit masking is specified ;

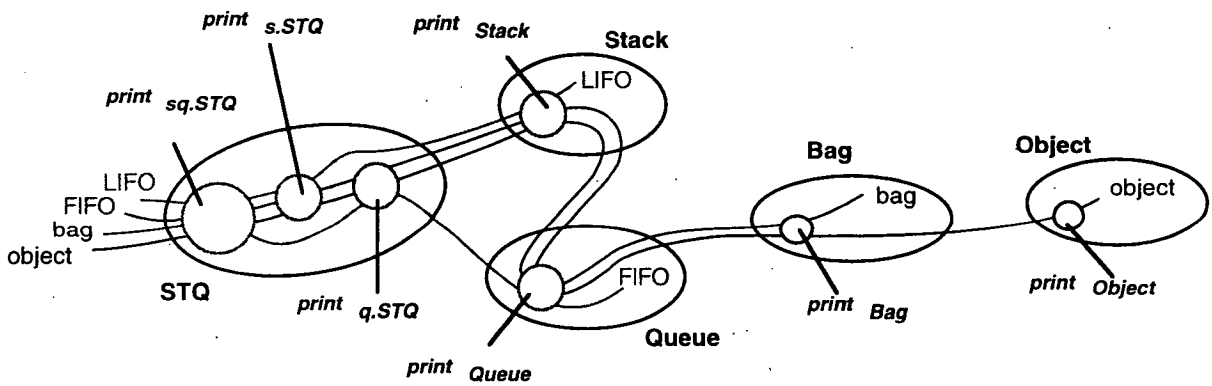



Figure 21.

— another potential problem is also eliminated naturally concerning *STQ*. This class inherits from two *pop* methods : one from *Stack* and one from *Queue*. The *STQ* hierarchy is thus potentially not regular vs. *pop*. Yet, when a *pop* message is issued, a single element is to be removed from the *STQ* instance and not two. Masking is thus used when necessary (cf. next figure).

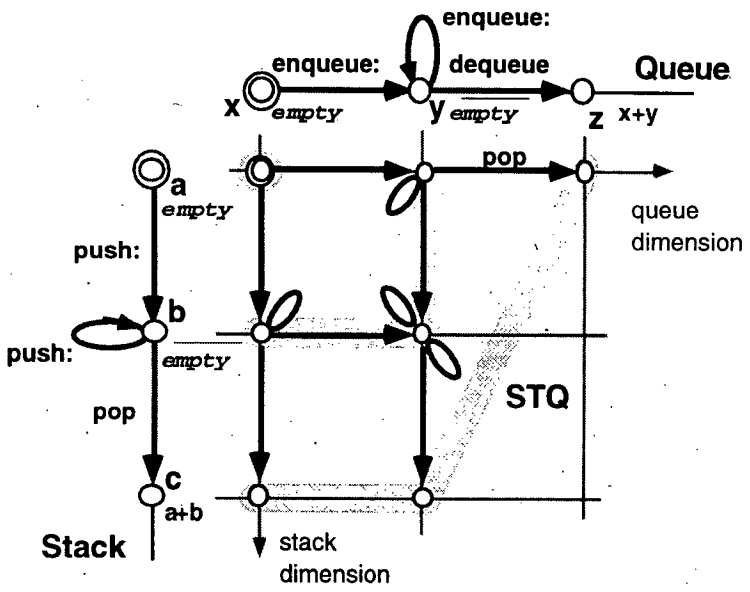



Figure 22.

Our expectation is thus that a hierarchy is normally regular vs. its methods once methods explicitly masked have been removed.

A hierarchy may also appear non regular in case two methods having no semantic relationship happen to be given the same name (name collision). This is an abnormal condition. The user is warned and invited to rename one or both methods.

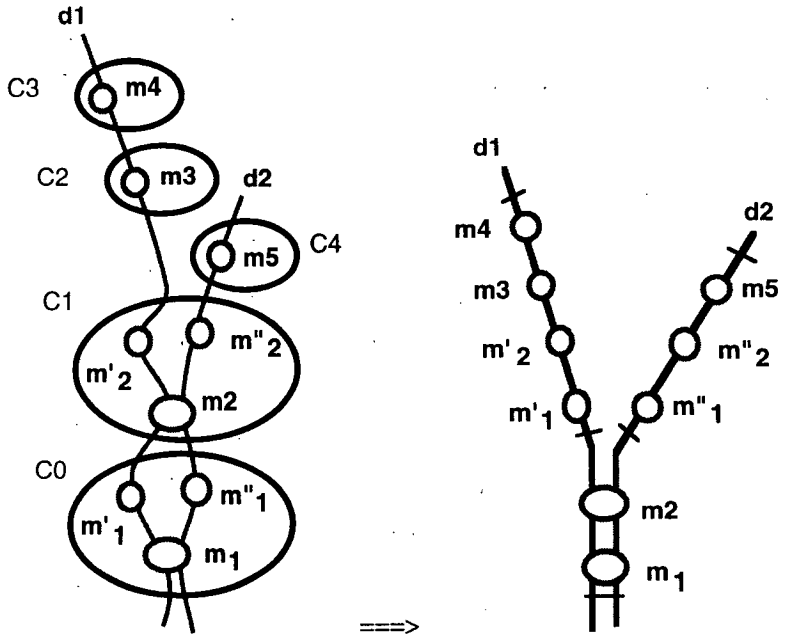
Regularity will thus be supposed for the rest of the paper.

## b.2) the rule

Let's come back to the invocation diagram of figure 18. We suppose regularity. Yet, compared to the original ordering of methods when recursively found along each dimension, this tree is possibly the result of a simple re-ordering[7] : the rule, termed the "prevalence of combined dimensions" rule, privileges more specialized methods vs. less specialized ones.

Next figure illustrates that : method *m2* is initially ordered after *m'1* and *m"1* ; because *m2* is more specialized than the others two (it is based on two dimensions instead of one), it is finally ordered before them. This enables the expression of modularity by successive refinements at different granularity levels: method *m1* refines method *m2* ; method *m'1* refines method *m1* ;... ; method *m"1* refines method *m"2* ; ... This basic scheme may be refined by masking in a rather sophisticated way : for example, *m1* may me declared as masking methods valid for both dimensions *d1* and *d2* (i.e. *m2*) while not masking methods valid for a single dimension.

[7] When two dimensions do not agree on the ordering of two methods, the ordering along the first dimension is selected. (An analogous rule exists in CLOS when, in the case of multiple dispatch, two cpls do not agree on the ordering of two methods.)




Figures 23 & 24.

## 7.2 Multiple Dispatch

### a) in OOP

Let's suppose a generic function call *m* is issued with a number of required arguments *x*, *y*,... of class $C_{X0}$, $C_{Y0}$, .... The *m* methods that satisfy the required arguments are primarily ordered according to the cpl of the first required argument class ; when several methods are attached to the same class in this first cpl, the cpl of the second required argument class is used to sort these methods ; if still tied, the cpl of the third required argument class is used in turn ; etc. For example, in the next figure, methods *m1*, *m2*, *m3*, *m4* and *m5* are ordered according to the cpl of *CX0* ; moreover, the first two methods are ordered according to the cpl of *CY0*. (The way these methods are combined is expressed in the next section.)

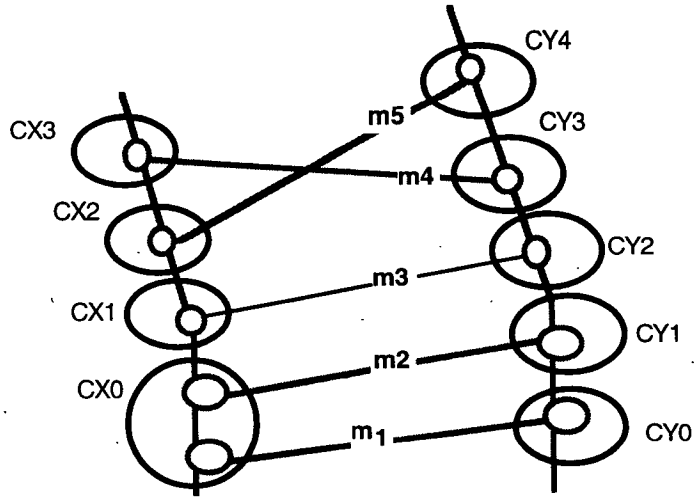



Figure 25.

### b) our proposal

The regularity condition is generalized (**global regularity**) : the same constraint is to be observed, but the dimensions of all arguments are to take into account. The "prevalence of combined dimensions" rule is also applied.

When considering a single required argument, the structure of the invocation sequence diagram corresponds to what was described in case of single dispatch.

In the general case, due to global regularity, a number of topological properties can be stated [4]. Next figure illustrates them. First of all, to each group of the *dxi* dimensions (first argument) corresponds a group of the *dyi* dimensions (second argument). For example, the tree made by the *dx1* and *dx2* dimensions is associated to the tree

made by the *dy1*, *dy2*, and *dy3* dimensions. Even if these two trees do not contain the same number of dimensions, they are isomorphic : they both have three parts (a trunk and two branches). The number of nodes on each part (trunk included) is the same in the two trees (two on each trunk, two on each left branch, one on each right branch). There is one crossing due to methods *m4* and *m5* : the classes of their arguments are not found in the same order in the *CX0* and *CY0* hierarchies. No crossing may exist between parts which are not in correspondence.

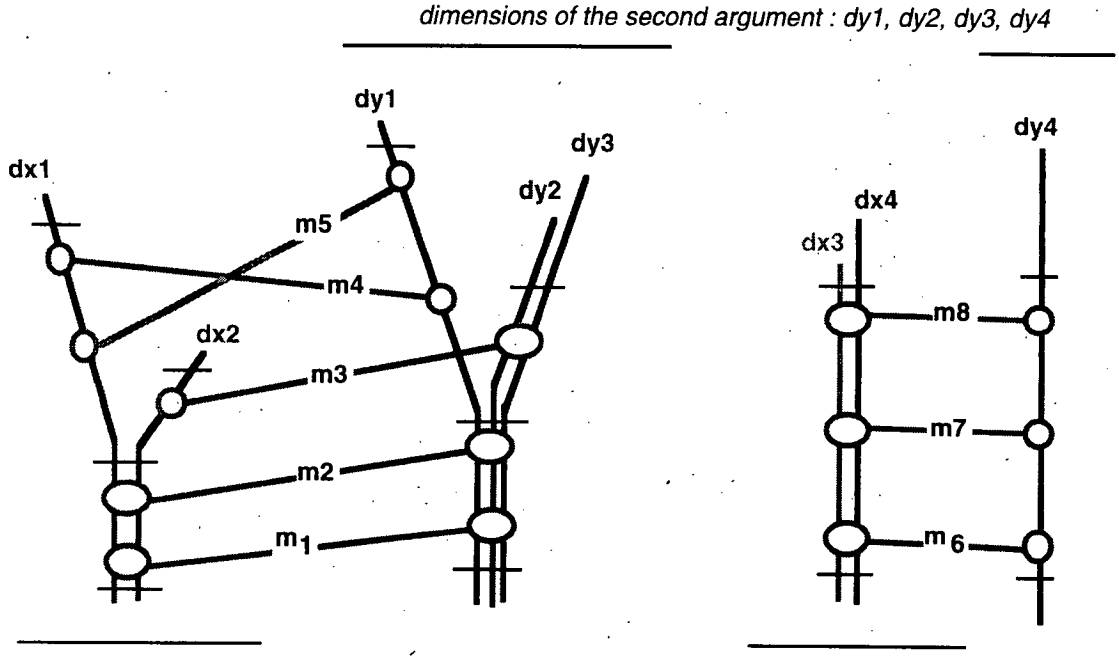



Figure 26.

Given these topological properties, the OOP approach is generalized : methods are primarily sorted according to the ordering defined by the first required argument, the ordering defined by the other required arguments being used only for breaking ties if any. An example is given in the above figure with methods *m1* and *m2*.

Let's now turn to the combinations of methods, with an emphasis on a nice aspect : pure declarativeness.

# 8. METHOD COMBINATIONS : A PURELY DECLARATIVE LANGUAGE

## 8.1 Problem

A question which is somewhat perturbing in traditional OOP is that it mingles declarative and imperative programming techniques for method combination.

The declarative technique consists in attaching qualifiers (like **:before,** or **:after**) to method declarations and using these qualifiers as a basis for method combination : present in CLOS, this technique is (unfortunately) not quite frequently supported by OO languages.

To be general, the imperative technique is due to the OO construct used for calling, from inside a given method body, a method having a same name but normally masked (for example, because it is defined in a superclass of the class in which this method body exists). Such a construct (ex. : **super** in Smalltalk ; **call-next-method** in CLOS) allows reuse while preventing infinite looping, but it is non modular. Sonya Keene, a member of the CLOS committee, expresses her worry about it in clear terms[8]. While the declarative technique lets the user *"predict the order of methods without looking at the code in the bodies of the methods"*, the imperative technique is *"in a sense (...) a violation of modularity"* and should thus be used *"only when that power is truly necessary"*. Unfortunately, *"some programs"* cannot be written *"without resorting to the imperative technique"*.

8 [9], section 5.8, p.111 : *"Guidelines on controlling the generic dispatch"*.

As shown below, our proposal takes advantage of the existence of pre- and post-methods ; it consists in running the pre-methods in bottom-up order, post-methods in top-down order and (if necessary) in attaching to the header of micro-methods (or, possibly, transitions) the indication 'masking should occur' in the form of a keyword (**:masking**), exactly like the **:before** or **:after** keywords. No **call-next-method** (nor **super**) is any longer necessary. By default, masking is done vs. all methods ranked after the masking method along the same dimension(s) ; but, it may also be specified as being done only vs. the methods valid for exactly the whole set of same dimensions.

## 8.2 Sophisticated Combinations without the Send-Super construct

Our proposal can be tuned so as to mimic –without the harmful **send-super** construct– the mechanisms supported by traditional OOP, for example the sophisticated CLOS ones.

In our proposal, instead of a simple list of classes for each argument, we have a list of trees (cf. the preceding section), each tree being due to one or several dimensions that are initially parallel (they traverse the same classes), then progressively diverge. So, to ease the initial comparison, we first express the basic idea for declarativeness supposing a single dimension per class hierarchy (i.e. per required argument) . This restriction is removed in the subsection afterwards (i.e. in 8.2.2).

### 8.2.1 Basic idea

Let's mimic the different cases of method combinations in CLOS

#### a) The standard method combination

This frequent combination deals with **:before**, **:after**, primary and **:around** methods. Original methods of a CLOS program will generally be split up according to the possible instance states. Certain **call-next-method** invocations may vanish in the process.

The solution we propose applies to the bodies of the resulting methods :
- **:before** methods are represented as pre-methods, **:after** methods as post-methods ;
- primary and **:around** methods are represented as pairs of pre- and post-methods.

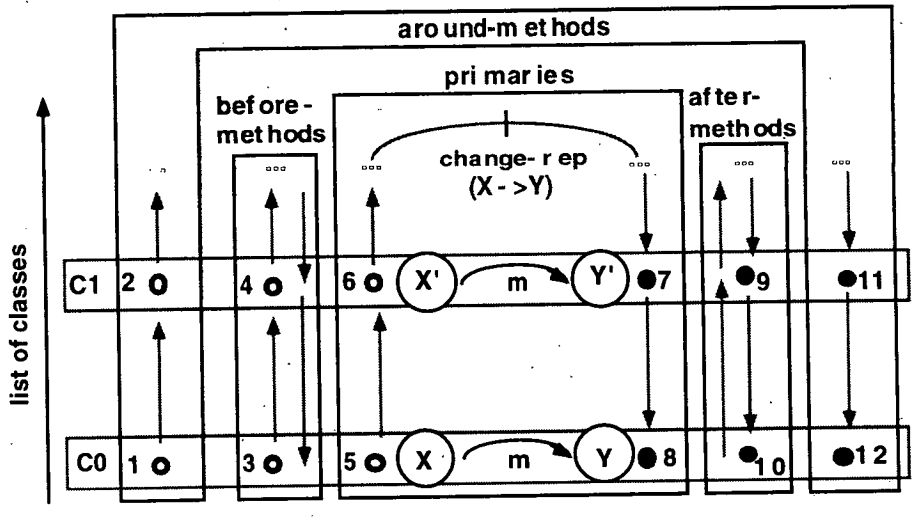



Figure 27.

The above figure illustrates our proposal (the numbers give the order of method executions).

Note the relative ordering of methods in CLOS (ex. : the running of **:before** methods from most specific to least specific or the running of **:after** methods from least specific to most specific) appears as a mere consequence of our more general mechanism (pre-methods in ascending order, post-methods in descending order).

See [3] for the specification of details (extra data control to respect the possibilities of the **call-next-method** semantics as well as the consequence of method splitting about temporaries.)

b) Other method combinations

Besides the standard method combination, CLOS offers a number of other built-in method combination types (simple ones, like **progn** or **max**). In these, the **:around** and primary roles are recognized. The former role may use **call-next-method** (and **next-method-p**) exactly as in the standard method combination : the above analysis thus applies. The latter role can't use **call-next-method** (hence, a simpler treatment than with the standard method combination) ; order of primaries is defaulted to **:most-specific-first** but can be changed to **:most-specific-last** : obviously, such an effect can easily be obtained in our approach by specifying the primary methods as being pre- or post-methods.

A user is also given two possibilities to define other method combination types. The first one, termed "short form", leads to the considerations just evoked (the simple built-in method combination types act as if they were defined using this form). The second one, termed "long form", provides great flexibility. From the perspective of the current discussion, this form leads to the considerations already evoked, either about the standard method combination type (which can be defined using it) or about the order of methods.

### 8.2.2 Generalization

Two method combinations are considered. The first one, termed **[combination]**, combine methods of a same degree in a same group of dimensions (like *m1* and *m2* in figure 24). It is like the CLOS combination along a cpl. The second combination, termed **{combination}**, combines items found along two or more diverging branches. As a matter of fact, the items to be [combined] (resp. {combined}) may themselves result from a {combination} (resp. [combination]).

Using this notation, the result of figure 24 is the following : *[m1 m2 { [m'1 m'2 m3 m4] [m"1 m"2 m5] } ]*. The result of figure 26 is : *{[m1 m2 { [m5 m4] m3} ] [m6 m7 m8] }*.

The interpretation of the [combination] is different in case of pre-methods (ascending order is the default : methods are processed from left to right between the brackets) and post-methods (descending order is the default : methods are processed from right to left between the brackets).

When qualifiers are used, the independence of dimensions allows various —and a priori equivalent— formulas to be used for combining the whole set of methods. The default is to combine in order the combined methods resulting from: the combination of all around pre-methods, the combination of all before methods, the combination of all primary pre-methods, the combination of all primary post-methods, the combination of all after methods, the combination of all around post-methods. This generalizes the CLOS standard combination. For other method combinations, the remark made in above subdivision b applies. (For an in depth study, see [4], subsection 10.3.)

## 9. MAKING LINEARIZATIONS EASY TO PREDICT

### 9.1 Problem

Inheritance is an interesting feature. Linearization is a simple and systematic way to handle it, at least as a default mechanism. Yet, it can be surprising by its results. A keyword in this matter is thus prediction.

This aspect has been the motivation for **monotonicity**, a property identified in [6]. This property is obtained when, for any hierarchy *HC0*, the cpl in a superclass *Si* of *C0* is a sublist of the cpl in *C0*, i.e. is the list of classes obtained by removing from *LC0* all classes that do not belong to the hierarchy *HSi* rooted in *Si*. When a linearization algorithm is monotonic, the cpl (global behavior) in class *C0* can be predicted from the cpl (global behavior) in the sole superclasses *S1... Sn*, plus *C0* (increment behavior). This property is supported by the linearization algorithms respectively proposed in [7] and in [5].

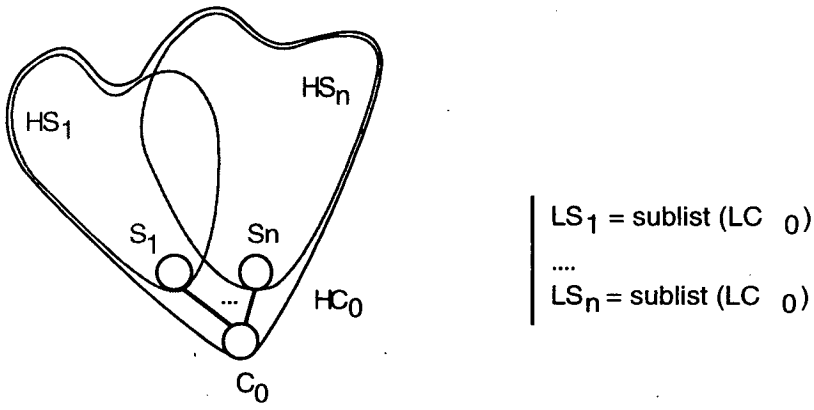



Figure 28.

Given our proposal, an additional property is desirable : **congruency** (cf. [4]). It is obtained when the cpl of the subhierarchy *HC0(C)* (hierarchy rooted in *C0* and with summit *C*) is a sublist of the hierarchy rooted in *C0*, whatever *HC0* and the summit *C*. Besides enabling a fast computation of the cpl of each dimension, this property also expresses a form of regularity.

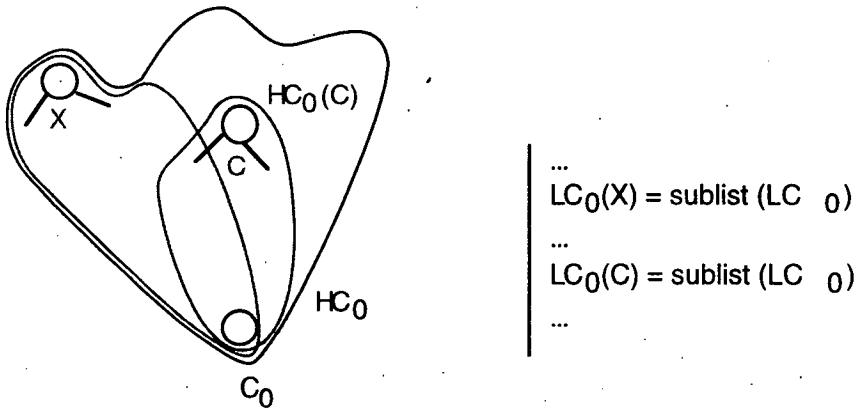


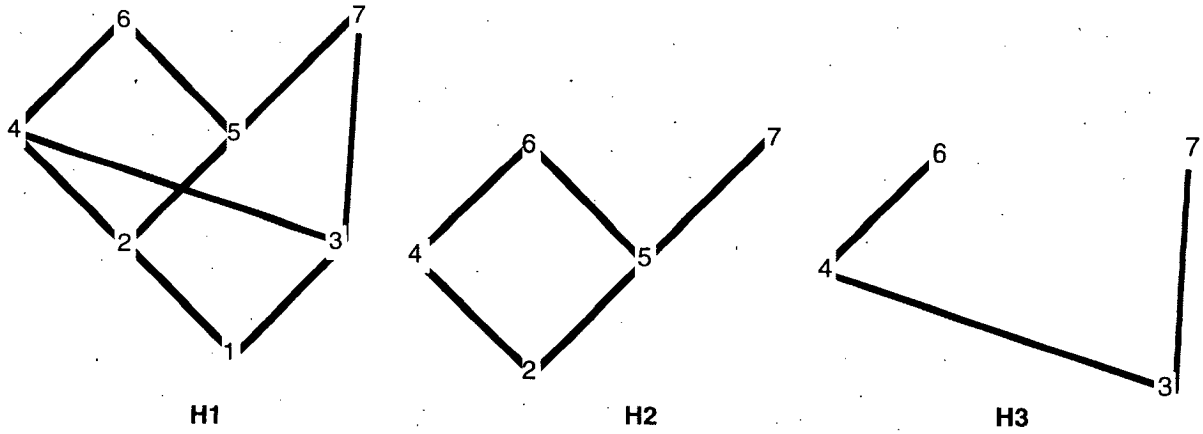

Figure 29.

However, as demonstrated in [4], congruency and monotonicity are antagonistic properties : a linearization algorithm cannot be congruent and monotonic for all hierarchies if it lists nodes blindly (i.e. if it lists nodes along a branch in the same way be this branch leading to a converging node or not). Thus, three solutions were envisaged. The last one was retained as being the most helpful for the programmer.

## 9.2 Solution 1

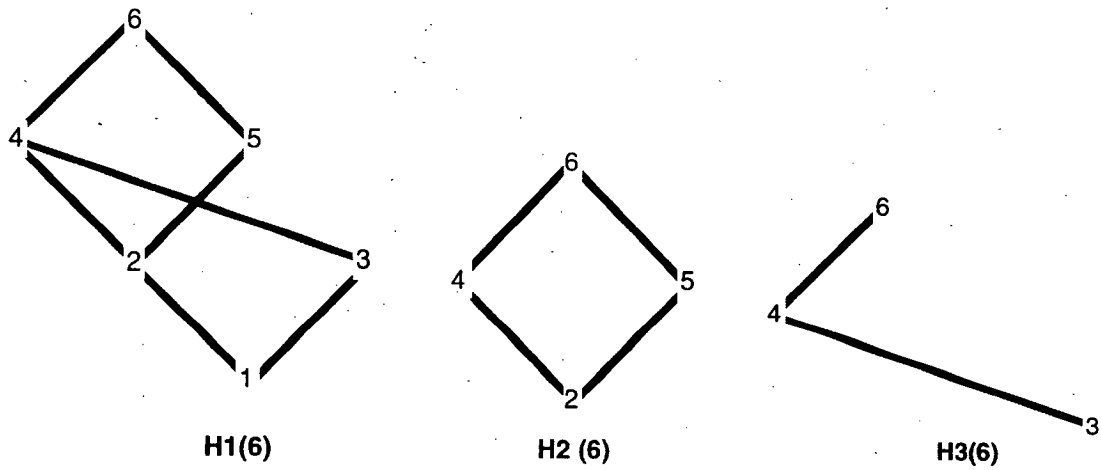
It consists in using a congruent linearization algorithm, for example and notably the LOOPS one (this one is demonstrated to be congruent in [4]). Under this hypothesis, the cpl for any dimension existing in *C0* is easily obtained from *LC0*. However, in this case, monotonicity is not guaranteed : thus, the cpl in *C0* cannot be predicted from the cpls of the superclasses of *C0* ; and the cpl for a given dimension of *C0* cannot be predicted from the cpls of the superclasses of *C0* for the same dimension.

Next figure illustrates this. Here, the cpl in node 1 is *L1 = (1 2 5 3 4 6 7)*. It cannot be predicted from the cpls in node 2, i.e. *L2 = ( 2 4 5 6 7)*, and in node 3, i.e. *L3 = (3 4 6 7)*. (*L2* and *L3* are poorer in informations than *H2* and *H3* ; the lost information appears essential for determining *L1*.)



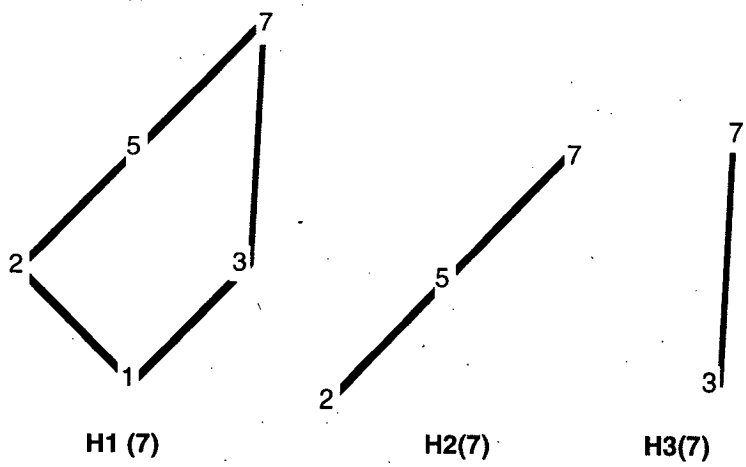

Figures 30, 31 & 32.

Concerning the dimensions that are introduced in class 6, the ordering of interest concerns the subhierarchy *H1(6)*. This ordering can be computed directly : *L1 (6) = (1 2 5 3 4 6)*. Since congruency is known to exist, *L1 (6)* can be obtained from *L1* by restricting this one to the sole classes that exist in *H1(6)*, i.e. by removing class 7 : *(1 2 5 3 4 6 7)*. However, because monotonicity is not guaranteed, the user cannot predict the ordering of a subhierarchy like *H1(6)* from the ordering of *H2(6)* and *H3(6)*, the sub-hierarchies with summit 6 and rooted in the superclasses of class 1. Here, *L2(6) = (2 4 5 6)* and *L3(6) = (3 4 6)*. (*L2(6)* is not a sublist of *L1(6)* : class 5 is not ordered in the same way in both lists.)




Figures 33, 34 & 35.

A similar study can be made vs. the subhierarchy *H1(7)*. In this case, results are not provoking any surprise : monotonicity happens to be observed.




Figures 36, 37 & 38.

Next figure graphically illustrates the study made so far vs. (the dimension(s) introduced by) class 6, as well as the similar study that would be made vs. class 7.

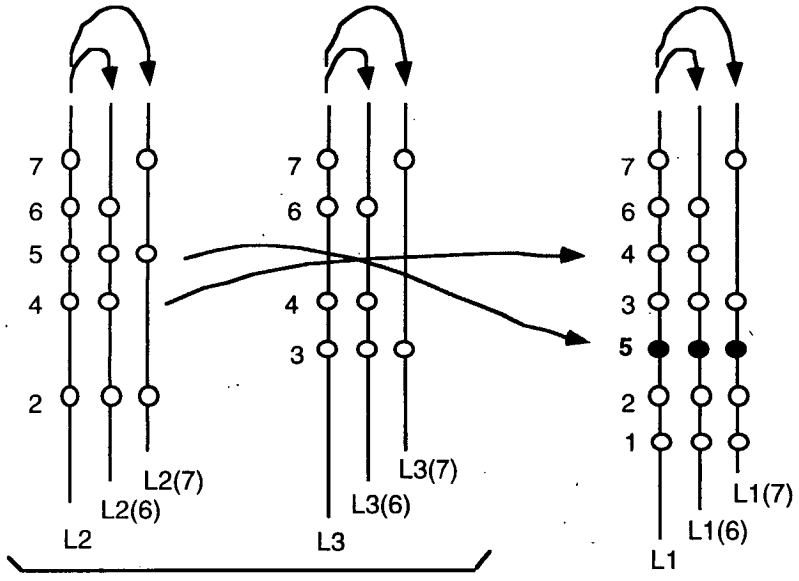



Figure 39.

With a congruent algorithm, the problem is thus that the user should consider the whole hierarchy each time : the behavior of the ancestor classes of *C0* cannot be abstracted in the sole superclasses of *C0*. The ordering in *C0* may be a surprise vs. the orderings previously obtained (in the superclasses of *C0*). On the opposite, the good point is that the ordering for each dimension always parallels the ordering obtained for the whole hierarchy (no surprise).

## 9.3 Solution 2

A second solution consists in using a monotonic linearization dimension per dimension. In other words, for each class *C* that introduces one or more dimensions $d_{Ci}$ in a hierarchy *HC0* rooted in *C0*, the ordering which is considered for these dimensions is *LC0(C)* where *L* is monotonic. The various cpls so obtained for various *C* may not order two same classes in the same way. Yet, for each dimension $d_{Ci}$, the order is monotonic : the cpl in each superclass *Si* abstracts the hierarchy *HSi(C)* ; on that sole basis, the user may predict the order in *C0* .

In our example, this yields the following orderings :

— dimension(s) introduced in class 6 : *L2 (6) = (2 4 5 6). L3(6) = (3 4 6). L1(6) = (1 2 3 4 5 6). L1(6)* can be obtained directly from the linearization of *H1(6)* : the ordering depends only on the classes impacting the dimension(s) in question. It can also be obtained from the orderings (global behaviours) first abstracted in 2 and 3, i.e. from *L2(6)* and *L3(6)* ) ;

— dimension(s) introduced in class 7 : *L2 (7) = (2 5 7). L3(7) = (3 7). L1(7) = (1 2 5 3 7). L1(7)* can be obtained directly from the linearization of *H1(7)*, or from the *L2(7)* and *L3(7)*.

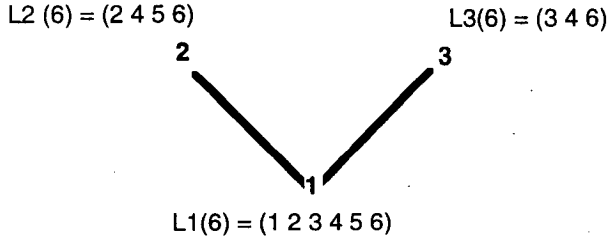



Figure 40.

## 9.4 Solution 3

### a) Properties

Given a monotonic linearization algorithm (cf. [10], [7] or [8]), this solution consists in computing *LC0* and, for each class *C* introducing a dimension, in restricting *LC0* to the classes belonging to *HC0(C)*. Be *LC0(C)* this ordering.

Because the linearization algorithm is monotonic, *LC0* may be obtained incrementally (from the cpl of each superclass of *C0*). The cpl for each dimension is easily computed. Yet, because congruency cannot be guaranteed in all cases, this *LC0(C)* ordering may well be different from *LC0(C)*, i.e. from the one obtained by directly linearizing the hierarchy *HC0(C)*. In other words, the $\mathcal{L}$ ordering cannot be predicted by isolating the classes impacting the dimension in question.

Yet, this $\mathcal{L}$ ordering happens to be monotonic. Since the linearization algorithm is monotonic, *LSi* is a sublist of *LC0* (if *LC0* exists). Since *LSi(C)* is obtained from *LSi* by removing a number of classes, it is a sublist of *LSi* ; in the same way, *LC0(C)* is a sublist of *LC0*. Thus, *LSi(C)* is a sublist of *LC0(C)* (they order the same classes in the same way ; and all classes in *LSi(C)* are part of *LC0(C)*). This property is valid for any superclass *Si* of *C0*. (If the orderings differ for *LSi(C)* and *LSj(C)* (with $i \neq j$), then *LC0* does not exists : the linearization algorithm rejects *HC0*). We call this property **$\mathcal{L}$ monotonicity**.

## b) Example

Let's consider the same example as above. Using the algorithm defined in [10], *L1* is equal to *(1 2 3 4 5 6 7)*. For the dimension(s) introduced by class 6, we use the restriction of *L1* to the classes of *H1(6)*. Hence, *$\mathcal{L}$1(6) = (1 2 3 4 5 6)*. Similarly, for the dimension(s) introduced by class 7, we use *$\mathcal{L}$1(7) = (1 2 3 5 7)*.

While the ordering *$\mathcal{L}$1(6)* used for the subhierarchy *H1(6)* happens to be equal to *L1(6)*, the ordering *$\mathcal{L}$1(7)* used for the subhierarchy *H1(7)* is <u>not equal</u> to *L1(7)*, i.e. to *(1 2 5 3 7)*. This is an unexpected result for a programmer attempting to compute the ordering for dimensions originating from class 7 by isolating the subhierarchy *H1(7)* : the ordering selected for such dimensions does <u>not</u> depend solely on the classes impacting these dimension(s). The reason for this is known : because the linearization algorithm is blind and monotonic, it is not systematically congruent.
Yet, this solution presents a good point : the $\mathcal{L}$ ordering vs. each dimension is <u>monotonic</u>.
— First, *L2 = (2 4 5 6 7)*. Thus, *$\mathcal{L}$2(6) = (2 4 5 6)*. Similarly, *$\mathcal{L}$2(7) = (2 5 7)*. *$\mathcal{L}$2(6)* is a sublist of *$\mathcal{L}$1(6)*, and *$\mathcal{L}$2(7)* is a sublist of *$\mathcal{L}$1(7)*.
— Second, *L3 = (3 4 6 7)*. From which we compute *$\mathcal{L}$3(6) = (3 4 6)* ; and, similarly, *$\mathcal{L}$3(7) = (3 7)*. *$\mathcal{L}$3(6)* is a sublist of *$\mathcal{L}$1(6)*, and *$\mathcal{L}$3(7)* is a sublist of *$\mathcal{L}$1(7)*.

Next figure graphically illustrates this study. *L1(6), L1(7), L2(6), L2(7), L3(6), L3(7)* are shown for comparison only.

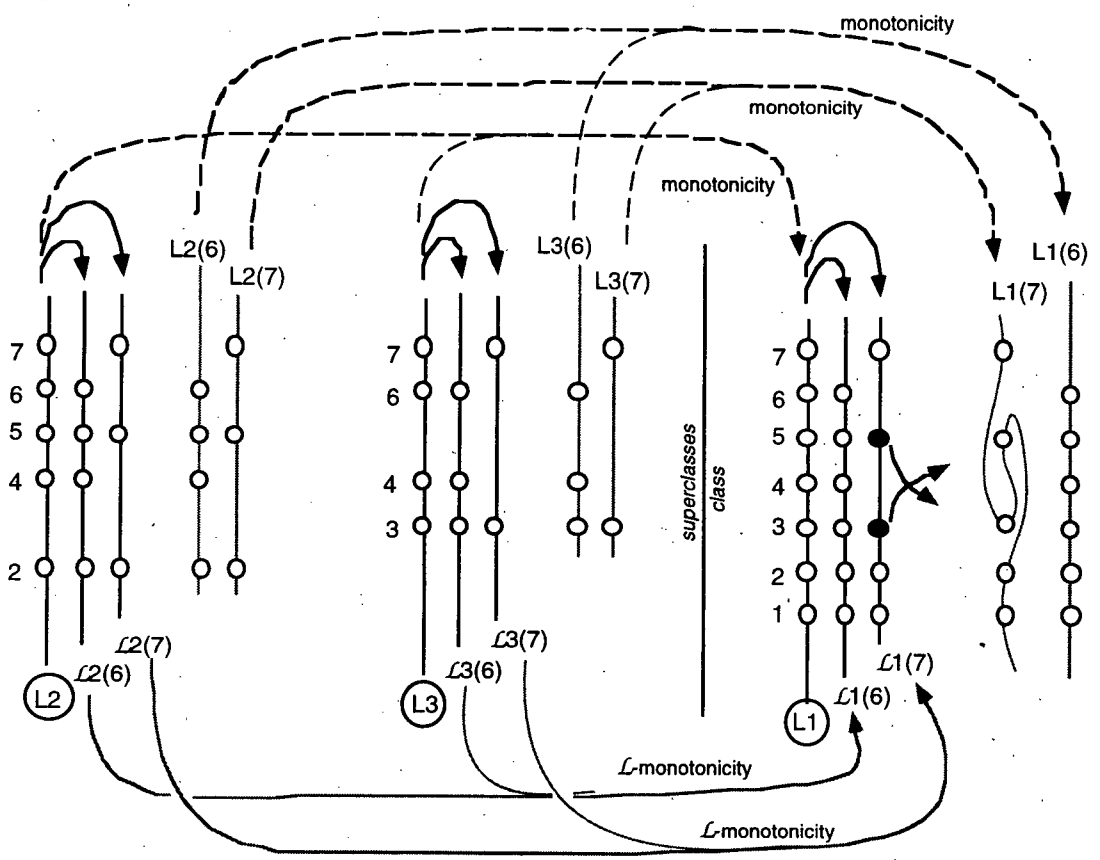



Figure 41.

## 9.5 Partial conclusion

If a solution was to be retained for its efficiency, this would certainly be the third one. It has interesting properties from a prediction point of view, even if may be "surprising" for some subhierarchies if one wants to directly predict the result of their linearization. This solution appears interesting to experiment for a cognitive evaluation. The second solution may seem more satisfactory from a cognitive point of view, but it is far less efficient (many linearizations to do). Whatever the choice, our class inheritance algorithms cope with both.

# 10. RELATED WORK

From our best knowledge, the work reported here seems to be unique in its category. As already expressed, it generalizes the approach of LISP-based OO languages using linearization (Flavors, LOOPS, CLOS [2], ...). Here, linearization is done in a multidimensional space. Is also of interest the important concept of meta-object protocol for enabling variations around the basic language [10]. The concept of "properties" [1] was reused and adapted.

Concerning linearization algorithms as such, the work reported in [7] is a major reference : it proposes an incremental linearization algorithm satisfying monotonicity. As announced here, [5] is about a new linearization algorithm, also monotonic, of general interest yet conceived with our approach in mind.

# 11. CONCLUSION

The goal of the paper was to show how the (traditional OOP) concept of class inheritance meshes with a hierarchy of multidimensional class graphs. We explained in a fairly detailed way how transitions as well as implementation items (memory representations, pre- and post-methods) are inherited in this context. The most complex case to deal with concerns the inheritance of methods : the basic idea consists in linearizing dimension per dimension the subhierarchies of classes that impact each dimension. To simplify the "invocation sequence diagram" that result from these linearizations, a condition (termed "regularity") and a rule (termed "prevalence of combined dimensions") are proposed : they make the invocation sequence diagram a tree-like structure. This being obtained, it is easy to combine the methods using two combinations (abstractly named the [combination] and the {combination}). Interestingly enough, sophisticated method combinations (using qualifiers like :before, :after, ...) are possible in a pure declarative way (the OOP classical imperative anti-modular *send-super* construct may be thrown away). To make inheritance easy to predict, a concept of congruency was proposed. Because congruency and monotonicity are antagonistic for any blind linearization algorithm, a refined study was made. Three solutions were proposed with different consequences on predictability and efficiency.

Two important observations may be drawn from a general point of view:

**a)** in this paper and —with much more details— in the research report [4], class inheritance rules (valid in a hierarchy of class graphs) are shown to be progressively developed from the simple and intuitive local inheritance rules (valid in a single class graph). This observation is valid for transitions as well as for implementation items. Contrasting with the imposition of magic rules extracted out of a hat like a bunny rabbit, this development thus fully explains all inheritance rules ;

**b)** from a conceptual point of view, the interpretation of inheritance in terms of dimensions was found quite illuminating. It will probably triggers new researches and bring new and important results due to a new mathematical point of view. Intuitively, it appears that merging all the points (coordinates) that exist on one dimension, as it is implicitly done in traditional OOP, certainly makes inheritance difficult to deal with.

From a much narrower point of view, the multidimensional aspect of inheritance developed here may certainly be a good basis for the development of an object-oriented system organized around statecharts.

# Bibliography


**[1]** America. *"Designing an Object-Oriented Programming Language with behavioral subtyping"*. Lecture Notes in Computer Science #489. 1990.

**[2]** Bobrow, DeMichiel, Gabriel, Keene, Kiczales & Moon. *"CLOS"*. X3J13 Document 88-002R. 1988.

**[3]** Borron. *"Colored-Object Programming : Concrete and Abstract Implementations"*. GL'95 Proceedings. (Huitièmes Journées Internationales sur le Génie Logiciel et ses Applications.) 1995.

**[4]** Borron. *"Colored-Object Programming : Inheritance by dimensions"*. October 1995. Revised, April 1996. RR 2878.

**[5]** Borron. *"A two-pass effcient and monotonic linearization algorithm"*. In preparation. INRIA Sophia-Antipolis. 1996.

**[5b]** Borron. *"Three efficientmonotonic linearization algorithms"*. In preparation. INRIA Sophia-Antipolis. 1996.

**[6]** Ducournau, Habib, Huchard & Mugnier. *"Monotonic conflict resolution for inheritance"*. OOPSLA'92 pp. 16-24.

**[7]** Ducournau, Habib, Huchard & Mugnier. *"Proposal for a monotonic multiple linearization"*. OOPSLA'94. pp. 164-175.

**[8]** Harel & Gery. *"Executable object modelling with statecharts"*. Proc. 18th Conf. Soft. Eng., pp. 246-257. 1996.

**[9]** Keene. *"Object-oriented progamming in Common Lisp. A programmer's guide to CLOS"*. Addison-Wesley. 1989.
**[10]** Kiczales, Rivières & Bobrow. *"The Art of the Metaobject Protocol"*. MIT Press, 1992.

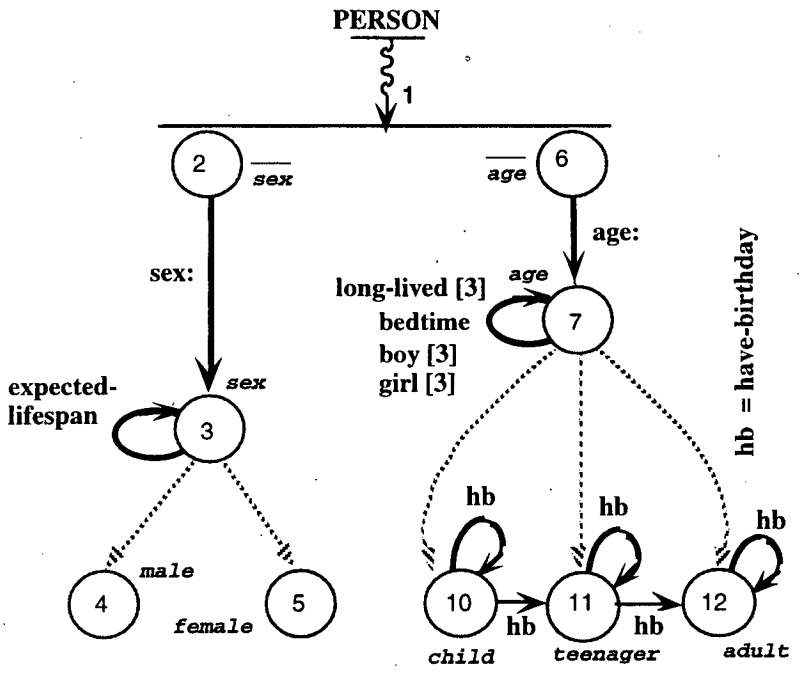



Figure 54.

As reported by references [14] and [15], the insideness vs. connectedness choice has been the subject of a debate for expressing hierarchical relationships in the early seventies : Nassi-Schneiderman (1973) and Jackson (1975) had both their adherents at that time. We are not going to aliment this debate : for the moment, we consider that a realistic system should support both styles depending on the user's choice. If a serious cognitive study based on actual observations were to draw definitive conclusions, these will be taken into account.

### b) Important properties of higraphs valid for class graphs

In his papers, Harel insists much on partitioning as well as on hyperedges. For example, [Harel et alii, 1987] (p. 54) lists the reasons why *"people working on the design of really complex systems have all but given up on the use of conventional FSM's and their state diagrams"* :

(1) *"State diagrams are flat"* ;

(2) *"State diagrams are uneconomical when it comes to transitions"* ;

(3) *"State diagrams are extremely uneconomical, indeed quite infeasible, when it come to states"* ;

(4) *"State diagrams (...) do not cater naturally for concurrency"*.

Then, about the idea of depth (i.e. the hyperedges), the same paper states : *"This simple idea, when applied to large collection of states in a multi-level manner, overcome points 1 and 2 above."* (p.55). About the partitioning, it explains : *"If the orthogonality construct is used often, and on many levels, difficulties 3 and 4 above are overcome in a reasonable way"* (p. 55).

As noted previously, the decomposition construct in class graphs is equivalent to the partitioning in higraphs, while transitions to ephemere nodes in class graphs are equivalent to hyperedges in higraphs. If we consider a given class higraph, its class graph equivalent can be built using this correspondence. Concerning this equivalent, the above claims are obviously maintained. They are also maintained for class graphs in general.

## 4.2.2 The McGregor-Dyer proposal

Concerning our proposal for mixins, [22] seems to be the work which, by many aspects, is the closest to ours.

### a) Generalities

The cited reference is about a work *"to better understand the relationship between a state machine representation of a class and similar representations for its subclasses"* (p. 61). It refers to the so-called "strict inheritance model" for inferring three implications (pp. 64-65) :

(1) *"A child class can not delete a state of any of its parent classes"* ;

(2) *"Any new state introduced in a child class is wholly contained in a existing state of one of the parent classes"* ;

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine - Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes - IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes - 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de recherche INRIA Rocquencourt - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 6399