Monotonic Conflict Resolution Mechanisms for Inheritance †

R. Ducournau
Sema Group,
16-18 rue Barbès,92126 Montrouge Cedex, France,
email: ducour@sema-taa.fr

M. Habib, M. Huchard, M.L. Mugnier
Dpt. Informatique Fondamentale, LIRMM,
860 rue de Saint-Priest, 34090 Montpellier, France,
email: name@crim.fr

Abstract

The main topic of this paper is multiple inheritance and conflict resolution methods in Object Oriented Programming. Our aim is to develop sound mechanisms easily understandable to any user. For this purpose, coherent behaviors of conflict resolution methods for multiple inheritance (such as supporting incrementality-monotonicity and stability under link subdivision) are introduced. We present interesting examples in which multiple inheritance known linearization algorithms (such as in CLOS [2] and LOOPS [19]) behave badly. Then we carefully study the conditions (on the inheritance graph) which assure good linearizations. We end with some suggestions for an incremental inheritance algorithm.

1 Introduction

1.1 Inheritance in object oriented systems

Inheritance is among the most interesting concepts of object oriented systems. However, it is an important motive for divergence. In this paper we adopt the general definition of inheritance as a specialization relation close to subtyping. Similarly, we use the general term "property" of an object, which may be a method, a datum, or both, depending on what specific language one refers to. Let us specify that our experience is essentially based on the hybrid language Y3 [7] and on object oriented systems based upon Lisp.

An inheritance strategy deals with answering the question: given a class x and a property P, what does x inherit concerning P?

One refers to *simple inheritance* when a class inherits directly from one class and *multiple inheritance* when a class may inherit directly from several classes.

When inheritance is simple, so is the strategy. The following masking rule applies: when a property P is defined in a class x and refined in a subclass of x, say y, then the value of P in y hides —or masks— the value of P in x, for y and its subclasses. Consequently, the P value for some x inheriting P is provided by the most specific superclass of x defining P.

With multiple inheritance, given a class x which inherits a property P, several superclasses of x may be in conflict when defining P. One has to

† This work was partially supported by the *Greco de Programmation*.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-539-9/92/0010/0016...$1.50

OOPSLA'92, pp. 16-24

choose between them. In addition, there can be clarity problems of the inheritance mechanism or even intelligibility. These stumbling blocks have led some people to exclude multiple inheritance [5]. However, there are strong arguments in its favour: modelisation closer to represented objects or better sharing of code [21], [19]. Besides, multiple inheritance is a feature offered by most object oriented systems.

1.2 Conflicts solving

We distinguish between two kinds of conflicts: conflicts with different values of the same property, called *value conflicts*, and conflicts between two distinct properties with the same name, called *name conflicts* [11]. In this paper, we are concerned with value conflicts. Indeed, we consider name conflicts to be software accidents which cannot be solved specifically by the inheritance mechanism.

The ways of solving conflicts differ with respect to several parameters. Examples given, the moment of the conflict resolution —at compile or run time— the kind of answer given to the user, the number of inherited occurrences [13]... However, strategies are divided into two main classes, depending on whether they request an interaction with the user or not. The first class includes explicit designation (C++ [20], extended Smalltalk [4], Trellis/Owl [17]), renaming (Eiffel [14]), exclusion (CommonObjects [18]), point of views (Objlog [10], Rome [6]). The only representatives of the second class are the linearization techniques.

1.3 Linearization techniques

The linearization techniques are mainly used for inheritance in object oriented languages based upon LISP —Old and NewFlavors [19], [15], CommonLOOPS [3], CLOS [2], $Y3$ [8], [7], [9], LOOPS [19]. They are also used in frame languages for reading and writing mechanisms with reflexes [22]. Their main advantage is the run-time conflict solving without human intervention. This specifically allows dynamic creation/destruction of objects. This technique appears to be a good default mechanism, used when no precise information on the object and its properties are available. Besides, in the above systems it is the basis of more sophisticated techniques. For instance, the numerous modes of method combination in Flavors [15].

On the other hand, the linearization algorithms are reproached for being non intuitive [5], [1]. Their implicit priority rules are indeed a cause of difficulties for the programmer to understand. Our work focusses on this stumbling block. We base it on the concept of monotonicity, previously introduced in [12], which corresponds to incremental computing of a linearization.

1.4 Outline of the paper

The paper is organized as follows:

Section 2 specifies the basic notations on inheritance graphs. Section 3 defines what a sound conflict resolution mechanism should be. We introduce the principles of uniformity, masking, monotonicity-incrementality and stability. In section 4, we restrict previous principles to the specific case of linearization. In section 5 and 6, we describe the linearization techniques, and particularly two linearization algorithms that we think are representative of their family, namely: LOOPS-$Y3$ and CLOS algorithms. We study their behavior with respect to the above criteria. Section 7 is devoted to our main results. We formalize an intuitive inheritance relation and prove that it is computed by the LOOPS-$Y3$ and CLOS algorithms on a very large class of graphs. On these graphs both algorithms are monotonic and stable. Furthermore, we actually characterize graphs on which the LOOPS-$Y3$ and CLOS algorithms are monotonic and stable.

2 Basic definitions and linearizations

The inheritance graphs we consider $H = (X, U)$ with X being the classes and U being the inheritance links, are by definition directed, simple (without multiple edges), loopless and without transi-

tivity edges. Usually H has no directed cycles and has a greatest element, denoted by ω. Therefore, to such a graph one may associate a strict order relation $<_H$ (or simply $<$ when non ambiguous) defined as follows: for $x, y \in X, x < y$ if and only if there is a path from x to y in H. In OOP terminology, x (resp. y) is said to be a *subclass* (resp. *superclass*) of y (resp. x). When $xy \in U$, y is said to be a *direct superclass* of x (resp. x is a *direct subclass* of y).

The inheritance hierarchy of a class $x \in H$, denoted by H_x, is the subgraph of H induced by x and its superclasses. For our study, we are only concerned by hierarchies.

A *linear extension* or topological sorting of $H = (X, U)$, is a total ordering of its vertices τ such that $x <_H y \Rightarrow x <_\tau y$. For commodity τ is also considered as a permutation (word of length $|X|$) on X.

A *linearization* is a mapping L which associates with every inheritance graph $H = (X, U)$, a linear extension of $<$ denoted by $L(H)$.

3 Sound conflict resolution mechanisms

In our model, a multiple inheritance mechanism is a mapping M, which associates to a class x of an inheritance graph H, and a property P, a class $y \in H_x$ which possesses P. Let us denote this class by $M(x, H, P)$ (or simply $M(x, P)$ when non ambiguous). Therefore x inherits from $M(x, P)$ the value of property P. Of course if P is refined in x, then $M(x, P) = x$.

> **Uniformity principle:** The mechanism is said to be *uniform* when, for all classes x and for all properties P and Q —P and Q being defined on the same set of classes, $M(x, P) = M(x, Q)$.

In other words, the inheritance mechanism is uniform if it is independent of the semantics of the property.

Let us denote by $CS(x, P)$ (for conflict set) the set of minimal classes in H_x which possesses P. Conflicts appear when $|CS(x, P)| > 1$. Conflict resolution mechanisms must choose one superclass as an answer. Generalization of the masking property defined in section 1 for simple inheritance is:

> **Masking principle:** For every x and every property P, $M(x, P) \in CS(x, P)$.

In other words, the P value inherited by x has to be chosen from the most specialized occurrences of P in superclasses of x. More sophisticated ways of computing the inherited value can be based on this concept. They are typically processed by combining several values of the conflict set, taking the infimum or the supremum in a lattice, or others techniques (for a description see [9]). We do not consider these extensions here.

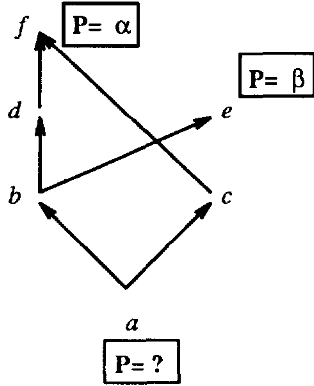
Let us now introduce new principles, monotonicity and stability, which we consider must be followed by every reasonnable inheritance mechanism. Moreover they are generalizations of previous observations: [12], [5] and [1].

3.1 Monotonicity-Incrementality

A good conflict resolution mechanism M must support abstraction or incrementality. In other words, for any property P and for any class x inheriting P, the value of P for x must be one of the P values inherited or defined by a direct superclass of x. Thus the user can easily add new classes as specializations of previously defined ones, by simply checking the behavior of direct superclasses. This allows incremental conception.

In our formalism this gives:

> **Monotonicity-Incrementality principle**
>
> For every inheritance graph H and for every property P defined in H, for every $x \in X$, either $M(x, P) = x$ or there is at least one direct superclass y of x, such that $M(x, P) = M(y, P)$.



Assume that $M(b,P) = M(c,P) = f$. In f the P value is α. Let us imagine the surprise of the programmer when adding a, he discovers that the P value of a is β. This may occur with linearization techniques (see Figure 4).

Figure 1: Non Incrementality

3.2 Stability under arc subdivision

In many cases when developing an application, one notices that some class x has properties which are general enough to be pushed up in the hierarchy and have to be separated from x. These properties can thus be reused by classes which are not necessarily specializations of x. Such an operation might simply locally modify the inheritance. In graph terms, such an operation is called an arc subdivision, since we can understand it as follows: an extra class $x - y$ has been added on the arc from x to y. From this we obtain:

Stability principle

For every inheritance graph $H = (X, U)$, for every link $xy \in U$, if H_{xy} denotes the graph obtained from H by subdividing the arc xy, then:

for every $z \in X$ and every property P, if $M(z, H, P) \neq x$ then $M(z, H_{xy}, P) = M(z, H, P)$, otherwise $M(z, H_{xy}, P) \in \{x, x - y\}$ (depending on the definition of $x - y$).

4 Specific case of linearization

A linearization yields a natural conflict resolution mechanism. Since it totally orders the ancestors of a class x in H, we can apply the simple inheritance mechanism (for inheriting P, take the first class in H_x where P is defined along this order). For a class x and a property P, let us denote this class by $L(x, H, P)$.

A linearization is obviously uniform and respects masking principle. It is not hard to be convinced that the following property ensures the respect of the above incrementality/monotonicity principle.

Monotonic linearization

For every inheritance graph $H = (X, U)$ and for every $x, y \in X$ such that $x < y$, then $L(H_y)$ is a subword of $L(H_x)$.

This property can be understood as being a monotonic behavior of the mapping L among inheritance subgraphs: $x < y \rightarrow H_y \subset H_x \rightarrow L(H_y)$ subword of $L(H_y)$ and 'to be a subword' is a partial order.

An alternative way of defining a monotonic behavior is related to the possibility of incrementally building $L(H_x)$ from linear extensions of x direct superclasses. Let $x_1 ... x_p$ be the direct superclasses of x. Then $L(H_x)$ can be obtained by merging $L(H_{x_1})$ ... $L(H_{x_p})$.

Similarly, our stability principle becomes in the specific case of linearizations:

Stable linearization

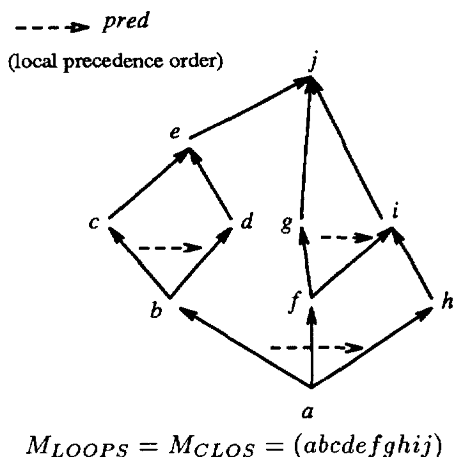
For every inheritance graph $H = (X, U)$, for every link $xy \in U$, then $L(H)$ is a subword of $L(H_{xy})$.

Let us now examine in full details some available linearizations.

5 Known linearization techniques

All systems, for their inheritance mechanisms, benefit from the local ordering yielded by the list of the

direct superclasses of a class (called local precedence order in CLOS, or multiplicity in $Y3$). In order to formalize this idea, we define a *representation* of an inheritance graph together with its predecessor ordering, *i.e.* $R(H) = (X, U, pred)$ with $H = (X, U)$ and $pred \subset U \times U$, a total order on the edges with same origin. See Figure 2.



$M_{LOOPS} = M_{CLOS} = (abcdefghij)$

Figure 2: Representation of an inheritance graph

We define $pred_X \subset X \times X$ a natural extension of *pred*: if $(xy, xz) \in pred$, then $(y, z) \in pred_X$. *pred* also provides an order, denoted by $pred*$, on the paths leaving a single class. Let $\alpha = (Pxx_i...y)$ and $\beta = (Pxx_j...z)$ be two paths having a common prefix (Px). $\alpha <_{pred*} \beta$ if $xx_i <_{pred} xx_j$. β is also said to be *righter* than α, with reference to the natural drawing of $R(H)$, edges being drawn from left to right in an increasing order of *pred*.

In the following, we assume that $pred_X$ has no cycle and the respect of U and $pred_X$ is not contradictory. A linear extension for $R(H)$ is a linear extension of H which is compatible with *pred* order. A *linearization* of $R(H)$ is a mapping L which associates, for every $R(H) = (X, U, pred)$, a linear extension of $R(H)$ denoted by $L(R(H))$. A linearization must at least produce same results on isomorphic representations, *i.e.* satisfy the following property: if there is an isomorphism θ from $R_1 = (X_1, U_1, pred_1)$ onto $R_2 = (X_2, U_2, pred_2)$, then for every $x \in X_1$, for every $a, b \in X_1$, then $(a <_{L(R_1)} b)$ if and only if $\theta(a) <_{L(R_2)} \theta(b)$.

6 The behavior of most popular linearizations

We have extracted two algorithms out of the systems LOOPS, Y3 and CLOS. Rather than providing their official definitions, we propose equivalent definitions which enable easier comparisons of their features. Both algorithms have a linear time complexity in the size of the examinated subgraph.

6.1 LOOPS-$Y3$ algorithm

The linearization algorithm of LOOPS [19], and the historical basis of the algorithm of $Y3$ [7], [9] have been defined independently but produce the same result. Let us call this mechanism M_{LOOPS}.

M_{LOOPS} builds a linear extension of H_x, for some x, as follows: successively choose a H_x-minimal object among the remaining objects. A H_x-minimal object is an object whose all subclasses in H_x have been taken. When several objects are available, take a H_x-minimal direct superclass of the object the most recently taken which possesses such a superclass (*i.e.* backtrack as less as possible). If this object has several H_x-minimal superclasses, take the first H_x-minimal superclass in $pred_X$ order.

In figure 2, $M_{LOOPS}(H_a) = (abcdefghij)$.

One can find other definitions of M_{LOOPS} in [9].

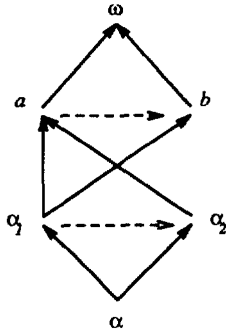
6.2 CLOS Algorithm

The following definition of M_{CLOS} is equivalent to the official one [2].

M_{CLOS} builds a linear extension of $R(H_x)$ as follows: successively choose a $R(H_x)$-minimal object among the remaining objects. A $R(H_x)$-minimal object is an object whose all subclasses in H_x and all predecessors by $pred_X$ have been taken. When several objects are available, take the $R(H_x)$-minimal direct superclass of the object the most recently taken which possesses such a superclass (*i.e.* backtrack as less as possible).

In Figure 2, M_{CLOS} behaves as M_{LOOPS}, but it is not true in general.

6.3 The behavior of M_{LOOPS} and M_{CLOS}



$M_{LOOPS}(H_\alpha) = (\alpha\ \alpha_1\ b\ \alpha_2\ \alpha\ \omega)$

Figure 3: M_{LOOPS} does not respect $pred_X$

M_{LOOPS} is always stable. It may be non monotonic, and furthermore it may contradict the order given by $pred_X$ —see Figure 3.

In general, M_{CLOS} is non monotonic, and non stable —see Figure 4.

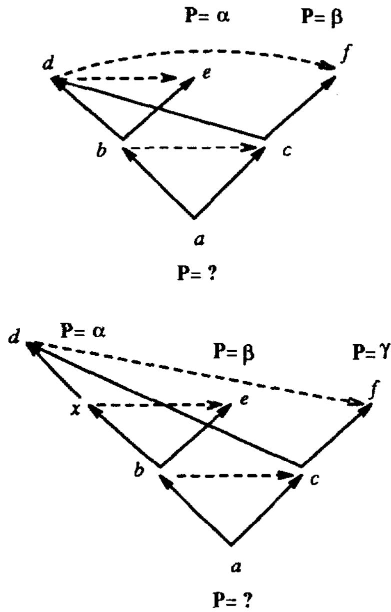


Figure 4: **Bad behaviour of** M_{LOOPS} **and** M_{CLOS}

Figure 4 —up— $M_{LOOPS}(H_a) = (abecdf)$ $M_{CLOS}(H_a) = (abcdfe)$. M_{LOOPS} is not monotonic: it does not respect $pred_X$ between d and e. M_{CLOS} is monotonic but not stable (See next Figure 4 —bottom—). Its behaviour is surprising: it puts f before e and so a inherits P with value β.

Figure 4 —bottom— (obtained from Figure 4 —up— by subdivision of the edge bd) Now, $M_{LOOPS}(H_a) = M_{CLOS}(H_a) = (abxecdf)$. M_{CLOS} is not monotonic. a inherits P from e. b and c both inherit P with value α but a inherits P with value β!

After these basic remarks on M_{LOOPS} and M_{CLOS}, let us establish some results concerning their relationships and their connections with a new total relation.

7 Main results

7.1 M_{LOOPS} and M_{CLOS} are equivalent for almost all inheritance graphs

One can summarize the behavior of these algorithms in the following way. M_{LOOPS} is a depth-first linearization which respects *pred* when possible. Symmetrically, M_{CLOS} is a linearization that respects *pred* which is depth-first when possible. The next result is thus not surprising.

Result 1 [11] - M_{LOOPS} and M_{CLOS} produce the same result if and only if M_{LOOPS} respects *pred*.

When M_{LOOPS} is monotonic then it respects *pred*. An immediate consequence is: if M_{LOOPS} is monotonic, so is M_{CLOS}.

7.2 A new total relation

Consider a class a and let b and c be two superclasses of a. If b and c are comparable by the inheritance relation, then there is a unique way of ordering them in $L(H_a)$. If they have a common direct subclass, say a', then there is also a unique way of ordering them in $L(H_a)$ according to the relation *pred* on $a'b$ and $a'c$. Otherwise there is no *a priori* order.

Let us advance two principles:

- The relation between two incomparable classes is provided by their maximal common subclasses —a common subclass of b and c, say a', is *maximal* if no superclass of a' is also a common subclass of b and c.
- The relation between two incomparable classes is an extension of $pred_X$. Intuitively, if there is a path from a to b righter than a path from a to c, then b has to precede c in $L(H_a)$.

Combining these ideas, we define an extended ordering, say $<_e$, as follows:

Definition

Given two classes x and y, $x <_e y$ if there are two paths $\alpha = (zx_1...x)$ and $\beta = (zy_1...y)$ such that $zx_1 <_{pred} zy_1$, with z is a maximal common subclass of x and y.

Let us note that if $zx <_{pred} zy$, then $x <_e y$. In general, $<_e$ may contain cycles, because the paths may be shuffled. *A fortiori*, the union of $R(H)$ and $<_e$, which we denote $R(H) \cup <_e$, may be contradictory.

If $R(H) \cup <_e$ is acyclic, then it totally orders the objects: indeed, given two objects, they are either comparable by $R(H)$; or they are comparable by $<_e$ because they possess at least one common subclass. Furthermore, assuming that $R(H) \cup <_e$ is acyclic, every linearization algorithm computing $R(H) \cup <_e$ is monotonic.

Result 2 [11]-

If $R(H) \cup <_e$ is acyclic, then M_{LOOPS} and M_{CLOS} compute exactly $R(H) \cup <_e$; they are monotonic.

7.3 Conditions for M_{LOOPS} and M_{CLOS} to be monotonic

We now focus on precisely characterizing the monotonicity and stability of M_{LOOPS} and M_{CLOS}. Let us first consider M_{LOOPS}. When computing the order $L(H_a)$, M_{LOOPS} does not take into account all paths from a to b and c; the decisive paths are the *rightest* paths from a to b and from a to c respectively. This leads us to define the following restriction of $<_e$, say $<_r$ (r for right).

Definition

Given two classes x and y, $x <_r y$ if there is z as a maximal common subclass of x and y, such that: let $\alpha = (zx_1...x)$ be the rightest (greatest for $<_{pred*}$) path from z to x; let $\beta = (zy_1...y)$ be the rightest path from z to y; then $zx_1 <_{pred} zy_1$.

Definition

A linearization L is **stable* for $R(H)$ if there is no sequence $R(H) = (R_0 R_1 ... R_p)$ —where each R_{i+1} is obtained from R_i by an edge subdivision— such that L is not stable for R_p. In other words, there is no representation R_{p+1} obtained from $R(H)$ by a sequence of edge subdivisions such that $L(R(H))$ is not a subword of $L(R_{p+1})$.

Result 3 [16]-

(i) M_{LOOPS} is monotonic if and only if $R(H) \cup <_r$ is without cycles.

(ii) M_{CLOS} is monotonic and *stable if and only if $R(H) \cup <_r$ is without cycles.

Of course, $<_r$ does not have a semantic justification. Instead, the user bases his analysis on the leftest (first) paths from a to b and c. This might define a new relation we denote by $<_l$. $<_e$ is in a way the conciliation of $<_l$, corresponding to the intuition, and $<_r$, explaining the behavior of known algorithms.

8 Conclusion

Figure 5 captures most of our results concerning existing algorithms. We have shown that they differ only on a very particular class of graphs.

Out of this study the notion of monotonicity brings new ideas to build incremental algorithms. The main issue we are now considering is to define

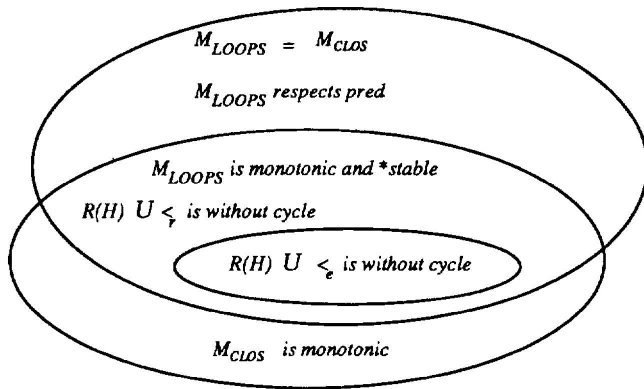


Figure 5: **Summarizing our results**

an algorithm computing a monotonic linearization when possible, and as compatible with M_{CLOS} as possible. Experimental results on existing hierarchies have shown that the non-monotonicity cases are fortunatly not frequent and therefore in these cases the system could automatically direct the user toward good or better *pred* orders, if any exists.

References

[1] H. G. Baker. CLOStrophobia : Its Etiology and Treatment. *OOPS Messenger*, 2(4), 1991.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13 document 88-002R. *Sigplan Notices*, 23, 1988.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops : Merging Lisp and Object-Oriented Programming. *OOPSLA'86 Proceedings*, 1986.

[4] A. H. Borning and D. H. Ingalls. Multiple Inheritance in Smalltalk 80. *AAI'82 Proceedings*, 1982.

[5] H. Brettauer, T. Christaller, and J. Kopp. Multiple versus Single Inheritance in Object Oriented Programming. *Arbeistpapiere der GMD 415*, 1989.

[6] B. Carré and J. Geib. The Point of View notion for Multiple Inheritance. *ECOOP/OOPSLA'90 Proceedings*, 1990.

[7] R. Ducournau. *Y3, version 3.* Sema Group, 1988.

[8] R. Ducournau and M. Habib. On some Algorithms for Multiple Inheritance in Object Oriented Programming. *ECOOP'87 Proceedings*, 1987.

[9] R. Ducournau and M. Habib. Masking and Conflicts *or* To Inherit Is Not To Own! In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 223–244. John Wiley and Sons Ldt, Chichester, West Sussex, 1991.

[10] P. Dugerdil. *Contribution à l'étude de la représentation des connaissances fondées sur les objets. Le langage OBJLOG.* PhD thesis, Université d'Aix-Marseille II, 1988.

[11] M. Huchard. *Sur quelques questions algorithmiques de l'héritage multiple.* PhD thesis, Université Montpellier II, 1992.

[12] M. Huchard, ML. Mugnier, M. Habib, and R. Ducournau. Towards a Unique Multiple Inheritance Linearization. *EurOOp'e 91 Proceedings*, 1991.

[13] J. L. Knudsen. Name Collision in Multiple Inheritance Hierarchies. *ECOOP'88 Proceedings*, 1988.

[14] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[15] D. A. Moon. Object-Oriented Programming with Flavors. *OOPSLA'86 Proceedings*, 1986.

[16] ML. Mugnier. PhD thesis, Université Montpellier II, to appear, October 1992.

[17] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilport. An introduction to Trellis/Owl. *OOPSLA'86 Proceedings*, 1986.

8 ✓

[18] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA'86 Proceedings*, 1986.

[19] M. Stefik and D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 6(4), 1986.

[20] B. Stroustrup. Multiple Inheritance for C++. *EUUG Spring Conference Proceedings*, 1987.

[21] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1), 1990.

[22] P. H. Winston. *Artificial Intelligence*. Addison–Wesley, 1977.

9