Proposal for a Monotonic Multiple Inheritance Linearization †

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

M. Habib, M. Huchard, M.L. Mugnier
LIRMM,
161 rue Ada, 34392 Montpellier Cedex 5, France,
email: name@lirmm.fr

Abstract

Previous studies concerning multiple inheritance convinced us that a better analysis of conflict resolution mechanisms was necessary. In [DHHM92], we stated properties that a sound mechanism has to respect. Among them, a monotonicity principle plays a critical role, ensuring that the inheritance mechanism behaves "naturally" relative to the incremental design of the inheritance hierarchy. We focus here on linearizations and present an intrinsically monotonic linearization, whereas currently used linearizations are not. This paper describes the algorithm in detail, explains the design choices, and compares it to other linearizations, with LOOPS and CLOS taken as references. In particular, this new linearization extends CLOS and LOOPS linearizations, producing the same results when these linearizations are sound.

†Please send correspondence concerning this paper to M. Habib.

OOPSLA 94- 10/94 Portland, Oregon USA

1 Introduction

1.1 Outline

Inheritance concept appears to be one of the most important contributions of object-oriented systems and gives rise to numerous promising debates concerning its meaning, use and implementation. *Multiple* inheritance, as compared to single inheritance, allows better code sharing and structuring. It is now offered by popular object-oriented languages such as C++ [Str86], CLOS [Ste90], BETA [KMMPN87] or EIFFEL [Mey88]. However, difficulties arise with its use, in particular the awkward problem of solving conflicts. The ways of solving conflicts can be divided into two main classes, depending on whether they request the designer to solve the conflict or proceed automatically. The first alternative is chosen by C++ (explicit designation) or EIFFEL (renaming clauses). The representatives of the second class are the linearization techniques, mainly used in languages built upon LISP —*e.g.* Old and NewFlavors [Moo86] [SB86], CommonLOOPS [BKK+86], LOOPS [SB86], CLOS [Ste90], *Y3* [Duc91], [DH91].

This paper concerns multiple inheritance mechanisms dealing with conflicts and essentially dedicated to dynamic inheritance of properties —*i.e.*

conflicts are solved at run time. We require the mechanism to be *sound, efficient* (in a time and space complexity context), entirely *automatic* and as *simple* as possible for the user. This mechanism is a basis for more complex computing —*e.g.* method combination. It can also be used as a default mechanism when no specific information on the semantics of properties is available.

Until now, the only known automatic methods have been based on *linearizations*. They also fulfill the efficiency requirement, but they are reproached for producing unclear and sometimes contra-intuitive results [Bak91], [BCK89], [Roy91]. In previous studies [HMHD91] [DHHM92], we defined properties that prevent conflict resolution methods —notably linearizations— from incoherent behaviors. The central property is called *monotonicity*. Since studied linearizations do not respect the monotonicity property, we determined conditions on the inheritance graph for these linearizations to be monotonic. After publication of these results, several OOP developers asked us for such a monotonic algorithm. We now propose a new linearization, which is monotonic, but also very close to used linearizations —with those of LOOPS and CLOS taken as references. However, the approach differs somewhat from existing linearizations: the linearization of a class is incrementally built from the linearizations of its superclasses, following the incremental building of the inheritance graph.

The paper is organized as follows: the rest of this part introduces the problem and the proposed solution using an example. Part 2 is devoted to the construction of a sound mechanism. We first recall the monotonicity property and related properties. It is shown that these properties are not fulfilled by currently used linearizations; however, it is also proved that there is no ideal answer. Finally, we present our approach and explain how our algorithm "extends" the known linearizations. Part 3 develops implementation issues.

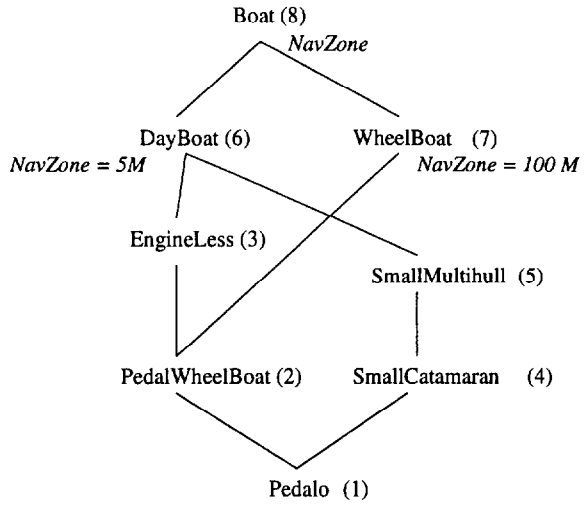
1.2 Inheritance allows Incremental Building of Classes

Let us first specify our framework. The global inheritance graph is denoted by H. The inheritance relation is intended to be reflexive, *i.e.* a class C is a subclass/superclass of itself, and may inherit from itself. Given any classes C_1 and C_2, we note $C_1 <_H C_2$ if C_1 is a subclass of C_2 and $C_1 \neq C_2$. H_C is the *hierarchy* of a class C, *i.e.* the restriction of H to the superclasses of C. We consider here the simplest form of inheritance: a *uniform inheritance mechanism* maps a class-property pair (C, P) to a class C', if any exists, such that C' is a superclass of C which possesses P. P is said to be inherited by C from C'. Then the value of P inherited in C is the value of P in C'.

To evaluate or design an inheritance strategy, we have to consider what the user does and what result he expects. An important part of "object programming" involves defining objects (classes or prototypes, according to the terminology of the language; we choose here the vocabulary of class-based languages). In this process, inheritance allows *incremental building* of classes, by creating a new class from existing ones —its *direct* superclasses.

Figure 1 describes part of a taxonomy for boats. A very general class *Boat* is created, then a class *DayBoat* is needed for the boats rented on the beach, and so on. The definition of *Boat* includes a property $NavZone$, which represents the navigation zone, *i.e.* the maximal distance to harbour, given in miles. *DayBoat* and $WheelBoat$ both specialize $NavZone$, by fixing a specific value, respectively 5 miles ($5M$) and 100 miles ($100M$).

When using *multiple inheritance*, several classes can compete for the definition of a class, *e.g.* $PedalWheelBoat$. Here, *DayBoat* and $WheelBoat$ both transmit the $NavZone$ property to $PedalWheelBoat$ causing a conflict for this latter class. The inheritance mechanism has to give a solution in accordance with the intuition of the user. Notice in this example that, according to the semantics of the $NavZone$ property, the conflict should be solved by taking the minimal value



$CLOS(2) = LOOPS(2) = \mathcal{L}^*_{LOOPS}(2) = (2\ 3\ 6\ 7\ 8)$
$CLOS(4) = LOOPS(4) = \mathcal{L}^*_{LOOPS}(4) = (4\ 5\ 6\ 8)$
$CLOS(1) = LOOPS(1) = (1\ 2\ 3\ 7\ 4\ 5\ 6\ 8)$
$\mathcal{L}^*_{LOOPS}(1) = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8)$

Figure 1: Partial taxonomy for boats

among the conflicting values. However, let us forget this correct but specific conflict resolution: we are only interested here in a generic default mechanism.

Let us suppose that *PedalWheelBoat* inherits *NavZone* from *DayBoat*. *SmallCatamaran* obviously inherits *NavZone* from *DayBoat* too. Now consider the class *Pedalo*. *Pedalo* being a direct subclass of *PedalWheelBoat* and *SmallCatamaran*, the designer's goal is to mix the behavior of *PedalWheelBoat* and *SmallCatamaran* (and also to refine it). He expects here that *Pedalo* inherits *NavZone* as *PedalWheelBoat* or *SmallCatamaran*, thus inevitably with value $5M$ (Figure 2). Unfortunately, this result is not ensured by known linearizations, as shown in next section.

1.3 Currently used Linearizations are not Monotonic

A *linearization* (say L) is a mapping which, with every class C, associates a list of its superclasses (denoted by $L(H_C)$ or $L(C)$) —called *class precedence list* in CLOS. This list is then interpreted as

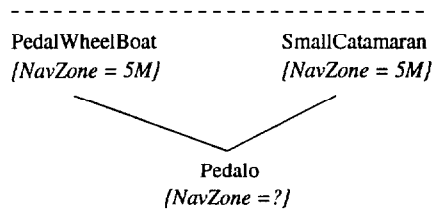


Figure 2: What value for *NavZone* should be inherited in *Pedalo*?

a *single inheritance* hierarchy, with the value of a property P in C given by the first class in that list which defines P. The list is obtained by a traversal of H_C, usually in a more or less depth-first manner. The linearizations used respectively in CLOS [DG87], [BDG+88], [Ste90] and LOOPS [SB86] are the two prevalent linearizations. For short we simply call them *CLOS* and *LOOPS*.

Let us recall the example in Figure 1. *CLOS* and *LOOPS* produce the same bad result. With *PedalWheelBoat* is associated the list (*PedalWheelBoat EngineLess DayBoat WheelBoat Boat*). In this list, *DayBoat* is the first class possessing *NavZone*. Thus *PedalWheelBoat* inherits *NavZone* from *DayBoat*, with value $5M$. The list associated with *Pedalo* is (*Pedalo PedalWheelBoat EngineLess WheelBoat SmallCatamaran SmallMultihull DayBoat Boat*). *Pedalo* inherits *NavZone* from *WheelBoat*, with value $100M$. As already mentioned, this result may surprise the user, since the direct superclasses of *Pedalo* both inherit *NavZone* from *DayBoat*, with value $5M$. Such a situation may be critical for navigational security!

1.4 An Incremental Algorithm Computing a Monotonic Linearization

We state a *monotonicity* principle to protect the inheritance mechanism from these defective behaviors. According to this principle, if a class C inherits a property P from another class C', then one of its *direct* superclasses must also inherit P from C'. In other words, a property value cannot skip a generation (*Pedalo* inherits *NavZone* with the same value as *PedalWheelBoat* or *SmallCatamaran*).

However, as shown later, it is not possible to define an "acceptable" linearization that is monotonic on all hierarchies. [DHHM92] studied already used algorithms, such as $CLOS$, and characterized the conditions of their monotonicity. We define here a new mechanism which is intrinsically monotonic. It is unable to process some hierarchies, but it processes hierarchies for which $CLOS$ is not monotonic. As suggested by the incremental aspect of the monotonicity, we propose to jointly build the inheritance hierarchy and a monotonic linearization.

Let us glance at our algorithm, we call $\mathcal{L}^*_{LOOPS}$ (the origins of this name are explained in part 2.4). Referring to Figure 1, suppose that we are adding *Pedalo*. Instead of traversing the hierarchy of *Pedalo* to compute its associated list as linearization algorithms usually do, we only consider the lists associated with the direct superclasses of *Pedalo*, namely *PedalWheelBoat* and *SmallCatamaran*. These two lists are merged to form the list for *Pedalo*, ensuring that classes are ordered similarly in all the lists. Two points should be raised: such a merging is not always possible and there are clearly different ways of doing it. These are discussed in the following sections.

Suppose in Figure 1 that the lists previously associated with *PedalWheelBoat* and *SmallCatamaran* are the same as in $CLOS$: $\mathcal{L}^*_{LOOPS}(PedalWheelBoat) = (PedalWheelBoat\ EngineLess\ DayBoat\ WheelBoat\ Boat)$ and $\mathcal{L}^*_{LOOPS}(SmallCatamaran) = (SmallCatamaran\ SmallMultihull\ DayBoat\ Boat)$. Then, for *Pedalo*, $\mathcal{L}^*_{LOOPS}$ builds the list $(Pedalo\ PedalWheelBoat\ EngineLess\ SmallCatamaran\ SmallMultihull\ DayBoat\ WheelBoat\ Boat)$. *Pedalo* inherits *NavZone* with the same value as *PedalWheelBoat*.

2 Towards a Sound Inheritance Mechanism

2.1 What *Sound* Means

2.1.1 Masking

Since inheritance relation is generally interpreted as a specialization relation, almost all systems agree that an occurrence of a property P in a class C' *masks* all other occurrences of P in superclasses of C', *i.e.* if C inherits P from C', then C' is one of the most specialized superclasses of C where P occurs. A linearization respects this property if and only if it produces a list compatible with the inheritance relation, *i.e.* is a *linear extension* —or topological sorting— of the inheritance hierarchy. That is, in this list a class always comes before its superclasses. The main currently used linearizations have this property, notably $CLOS$ and $LOOPS$. Thus, we will hereafter restrict the meaning of linearization to a mechanism which, with each class, associates a linear extension of its hierarchy. As a consequence, a linearization offers an always sound inheritance mechanism when there is no conflict.

2.1.2 Monotonicity

An inheritance hierarchy may accept a great number of linear extensions, and the question is what characterizes a sound one. As previously explained, the incremental conception of an inheritance graph implies that the inherited behavior of a new class should be easily understandable from the behavior of its *direct* superclasses. This can be translated by the following general principle:

> **Monotonicity principle:** If C inherits P from C', with $C \neq C'$, then there is a direct superclass of C which also inherits P from C'.

For linearizations, monotonicity can be defined in the following way:

> **Definition (Monotonic linearization)** L is monotonic for H_C if, for all C' superclass of C, $L(C')$ is a sublist of $L(C)$.

That is the restriction of $L(C)$ to the classes of $H_{C'}$ is exactly $L(C')$. From an incremental viewpoint, the list of C can be built without traversing C hierarchy, by a conservative merging of the lists of its direct superclasses —*i.e.* a merging retaining the list orders.

2.1.3 Consistency with the local precedence order

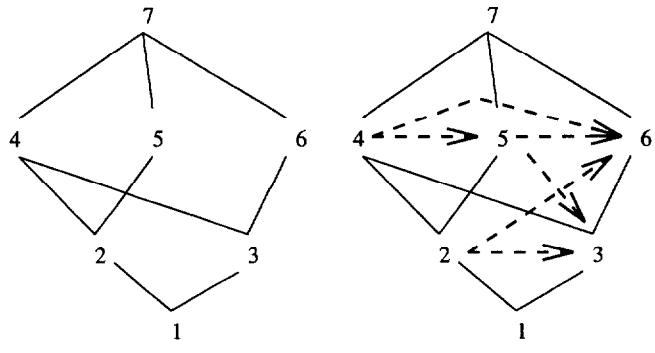
Linearization-based strategies use an implicit or explicit order on the direct superclasses of each class —the *local precedence order* of CLOS. This order is usually given by the user, and corresponds to the machine representation of the graph. It becomes inseparable from the hierarchy. In figures, when not explicitly mentioned, it increases from left to right. We note $prec_C$ the local order for a class C and *prec* the union of these local orders for a hierarchy. Given C_1 and C_2, we note $C_1 <_{prec} C_2$ if there is a C, with $C_1 <_{prec_C} C_2$. Here *prec* is supposed to be acyclic. *prec* also induces an order (say $<_{path}$) on paths leaving a class. Let $\lambda_1 = (\Pi\ C\ C_1\ ...\ C_p)$ and $\lambda_2 = (\Pi\ C\ C_2\ ...\ C_q)$ be two paths, with common prefix (ΠC); $\lambda_1 <_{path} \lambda_2$ if $C_1 <_{prec_C} C_2$.

2.1.4 Consistency with the extended precedence graph

In [HMHD91], we defined an extension of *prec*, called $<_e$, which provides an additional guide for building a sound linearization. It is based on the following observation: let us consider two classes (say a and b) which are uncomparable relative to $<_H$; if a and b have a common *direct* subclass (say c), they are ordered by $prec_c$; otherwise, one intuitively looks at a "closest" common subclass of a and b (say c), and orders a and b by *extending* the local order $prec_c$.

Definition ($<_e$) Given two classes a and b, $a <_e b$ if there are two paths $\lambda_1 = (c\ ...\ a)$ and $\lambda_2 = (c\ ...\ b)$ such that c is a maximal common subclass of a and b, and $\lambda_1 <_{path} \lambda_2$ —a common subclass of a and b, say c, is maximal if no superclass of c is also a common subclass of a and b.

Definition (Extended precedence graph T) T_C is the graph obtained from H_C by adding the edges of the relation $<_e$ (the notation T stands for "total", or "tournament" in a graph theory sense, since there is at least one edge between any two vertices in the transitive closure).



With 2 *and* 3 *are respectively associated the linear extensions* (2 4 5 7) *and* (3 4 6 7)
$CLOS(1)$ = (1 2 3 4 6 5 7)
CLOS is monotonic but not consistent with T_1
$LOOPS(1)$ = (1 2 5 3 4 6 7)
LOOPS is consistent with T_1 *but not monotonic*
$\mathcal{L}^*_{LOOPS}(1)$ = (1 2 3 4 5 6 7)
$\mathcal{L}^*_{LOOPS}$ *is monotonic and consistent with* T_1

Figure 3: Extended precedence graph

Note that $<_e$ may contain cycles since the paths leaving a (maximal) common subclass may be shuffled. In addition the union of $<_{H_C}$ and $<_e$ may also be contradictory (Figure 3: T_1 has a cycle (3 4 5 3)). But, when T_C is acyclic, it totally orders the classes of H_C; furthermore, every linearization computing this ordering is monotonic on H_C.

Definition (Consistency with T) A linearization L is consistent with T_C if $L(H_C)$ is included in the transitive closure of T_C.

In other words, for any uncomparable classes a and b, if $a <_e b$, and there is no cycle in T_C going through a and b, then a comes before b in $L(H_C)$: the unambiguous part of T_C is included in $L(H_C)$. Let us consider Figure 3. The drawing of the hierarchy and *prec* suggest that class 5 should appear before class 6 in the list of class 1. The extended precedence graph explains this intuition: $5 <_e 6$ and we do not have $6 <_{T_1} 5$ (*i.e.* there is no path from 6 to 5 in T_1). Assume that L is monotonic, with $L(2) = (2\ 4\ 5\ 7)$ and $L(3) = (3\ 4\ 6\ 7)$. There are two possible linear extensions of H_C, which respect the local order $(2 <_{prec_1} 3)$: $(1\ 2\ 3\ 4\ 5\ 6\ 7)$ or $(1\ 2\ 3\ 4\ 6\ 5\ 7)$. The first one is consistent with T_1 (it is computed by $\mathcal{L}^*_{LOOPS}$), while the second one is not, because 6 comes before 5 (it is produced by $CLOS$).

2.2 Currently used Linearizations are not Sound

As previously noted, $CLOS$ and $LOOPS$ do not guarantee the monotonicity property (Figure 1: both linearizations order $WheelBoat$ and $DayBoat$ differently for $PedalWheelBoat$ and $Pedalo$).

[DHHM92] studied the relationships between $CLOS$ and $LOOPS$. Let us recall one important result: when $LOOPS$ is monotonic, then $CLOS$ is monotonic (actually, $CLOS = LOOPS$). The reciprocal is not true (see for example Figure 3). Concerning the extended precedence graph, the following holds:

> **Result** [HHM93] For all hierarchy H_C, $LOOPS$ is consistent with T_C. $CLOS$ is not always consistent with T_C, even if monotonic (Figure 3).

> **Result** [DHHM92] If T_C is acyclic, then $LOOPS(H_C) = CLOS(H_C) = T_C$

In summary, $CLOS$ is monotonic on more graphs than $LOOPS$, but $LOOPS$ is intrinsically consistent with T_C, while $CLOS$ is not. Both algorithms produce the same result when the extended precedence graph is acyclic.

2.3 No hope for a Totally Monotonic Linearization

A priori, one could imagine several intrinsically monotonic linearizations, which are not related to the structure of a given hierarchy. An example of an extreme case would be a linearization based on a lexicographical order over the class names. Then, changing the name of a class could change the behavior of its subclasses. These observations led us to define what an *acceptable* candidate is. A weak requirement for the linearization mechanism is to produce the same lists for hierarchies with the same "structures".

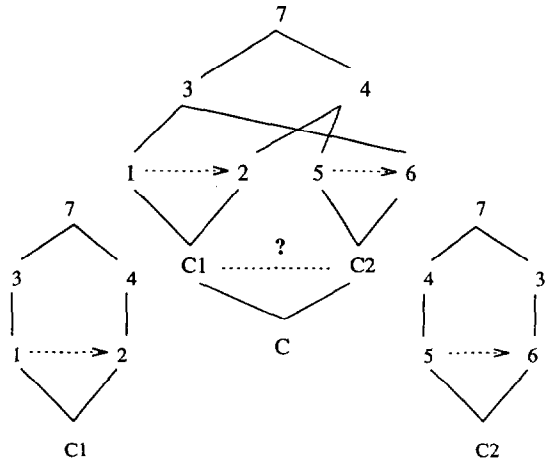
> **Acceptability Criterion :** An acceptable linearization must produce similar results on hierarchies which are similar when taking *prec* into account.

More formally: Let H_{C_1} and H_{C_2} be any hierarchies provided with precedence orders $prec_{H_1}$ and $prec_{H_2}$ respectively. Suppose there is an isomorphism Φ from H_{C_1} to H_{C_2}, such that, for all classes a and b of H_{C_1}, $a <_{prec_{H_1}} b$ *iff* $\Phi(a) <_{prec_{H_2}} \Phi(b)$. Then, for any acceptable linearization L, $L(H_{C_2}) = \Phi(L(H_{C_1}))$, where $\Phi(L(H_{C_1}))$ denotes the list obtained from $L(H_{C_1})$ by replacing each class with its image by Φ (see Figure 4). All currently used linearizations are acceptable. We will hereafter restrict the meaning of linearization to an acceptable linearization.

> **Result** [Mug92] There is no acceptable linearization which is monotonic for all graphs: see Figure 4.

2.4 The Proposed Approach: Incremental Monotonicity

The solution we propose builds the linearization and the hierarchy at the same time (and then incrementally), so that the linearization is always monotonic. The algorithm is applied when inserting a class C in the hierarchy, (*e.g.* $C = Pedalo$ in Figure 1). The lists associated with the superclasses



H_{C_1} and H_{C_2} are isomorphic subgraphs of H_C. Acceptable linearizations associated to classes C_1 and C_2 inevitably order 3 and 4 in reverse order, causing trouble when C is inserted.

Figure 4: No monotonic linearization on all graphs

of C are supposed to be already computed by some linearization, say L. The method consists of trying to compute a linear extension of H_C —C list— that is compatible with the union of the lists associated with the direct superclasses of C. Such a total order exists *iff* one can merge the lists associated with the direct superclasses of C, *i.e.* there is no circuit $(a_1 \ldots a_q \ a_{q+1} = a_1)$, where, for all $1 \leq i \leq q$, there is a list in which a_i comes before a_{i+1}.

2.4.1 Linearization graph

The union of these lists is represented by a graph, we call *linearization graph of* C, noted $H_L(C)$. Classes of H_C form the vertex set of $H_L(C)$. Each list associated with a direct superclass C_i of C is represented as a path leading from the first class (C_i) to the last class (the most general class). In addition, there is an edge from C to each of its direct superclasses. See figures 5, 6 and 7 for examples (where L may be indifferently $CLOS$, $LOOPS$ or $\mathcal{L}^*_{LOOPS}$).

Definition (Linearization graph H_L) Given a linearization L, the linearization graph associated with a class C, denoted by $H_L(C) = (V_C, E_C)$, is a directed graph, with vertex set V_C and edge set E_C. V_C is exactly the vertex set of H_C. E_C is $\{(C, C_i)$ *s.t.* C_i is a direct superclass of $C\} \cup \{(a, b)$ *s.t.* b immediately follows a in the list of a direct superclass of $C\}$.

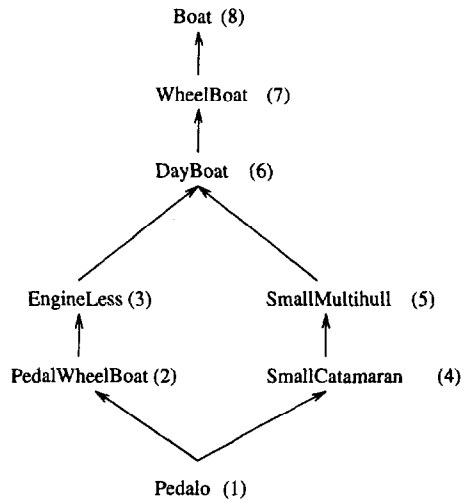
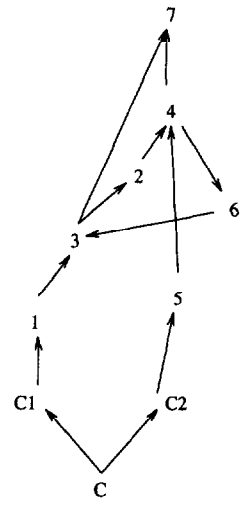


Figure 5: The linearization graph for Figure 1



There is a circuit (3 2 4 6 3)

Figure 6: The linearization graph for Figure 4

Results [Mug92]

- One can conservatively merge lists associated with the direct superclasses of C *iff* $H_L(C)$ has no circuit.
- Given L a monotonic linearization on the superclasses of C, L can be extended to a monotonic linearization of H_C by computing any linear extension of $H_L(C)$. Reciprocally, any monotonic linearization L of H_C produces a linear extension of $H_L(C)$.
- When L is acceptable, computing L on $H_L(C)$ yields an acceptable linearization L' of H_C.

In addition to ensuring acceptability and monotonicity, a "good" linearization should be consistent with T_C and as close as possible to already used linearizations when these latter are sound. These requirements guided our definition of a local precedence order on $H_L(C)$ and our choice of a linearization algorithm.

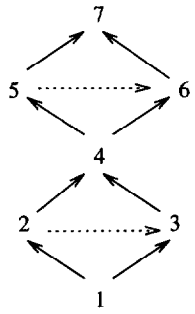


Figure 7: The linearization graph for Figure 3

2.4.2 Local precedence order for H_L

Remark. If (a, b) is an edge of $H_L(C)$, we say, according to graph theory terminology, that b is a *successor* of a —rather than a *direct superclass*, which would not be correct here.

Let us note $prec'$ the precedence order on $H_L(C)$. The restriction of $prec'$ to the successor set of any vertex d —we note $prec'_d$— is the following total order: let e and f be successors of d; let C_i (resp. C_j) be the greatest direct superclass of C (relative to $prec_C$) which is a subclass of e (resp. f); then $e <_{prec'_d} f$ if $C_i <_{prec_C} C_j$ (Figure 7: $2 <_{prec'} 3$ and $5 <_{prec'} 6$). For C, we have $prec'_C = prec_C$. The union of these local orderings (*a fortiori* the union of these local orderings and H_L) may be contradictory —*i.e.* may contain circuits— but it is of no importance for the algorithm.

2.4.3 The Algorithm $\mathcal{L}^*_{LOOPS}$

Let us define the following generic notation $\mathcal{L}^*_L$: given a linearization L, $\mathcal{L}^*_L$ is the linearization defined as the application of L to the linearization graph $H_{\mathcal{L}^*_L}$ (and when H is restricted to one vertex, then $H_{\mathcal{L}^*_L} = H$). Of course this notation has no sense if $H_{\mathcal{L}^*_L}$ contains a circuit. Our algorithm basically consists in computing $\mathcal{L}^*_L$ with $L = LOOPS$. More precisely, let C be the newly inserted class. The algorithm first builds $H_{\mathcal{L}^*_{LOOPS}}(C)$. If $H_{\mathcal{L}^*_{LOOPS}}(C)$ has a circuit, it produces an error. Otherwise, it involves computing $LOOPS$ on $H_{\mathcal{L}^*_{LOOPS}}(C)$ —actually we use a slightly modified version of $LOOPS$, see Appendix for more details. *E.g.* with the linearization graph of Figure 5, the resulting list is (*Pedalo PedalWheelBoat EngineLess SmallCatamaran SmallMultihull DayBoat WheelBoat Boat*). With the linearization graph of Figure 7, one obtains (1 2 3 4 5 6 7).

2.4.4 Properties of $\mathcal{L}^*_{LOOPS}$

$\mathcal{L}^*_{LOOPS}$ is an acceptable and monotonic linearization which has other attractive properties:

Main results

- For all hierarchy H_C, where $H_{\mathcal{L}^*_{LOOPS}(C)}$ has no circuit, $\mathcal{L}^*_{LOOPS}(C)$ is consistent with T_C [HHM93].
- $\mathcal{L}^*_{LOOPS}$ produces the same results as $LOOPS$ and $CLOS$ *iff* $LOOPS$ is monotonic [Mug92].

In addition, $\mathcal{L}^*_{LOOPS}$ can be efficiently computed, as detailed in the next section. Remark that it would not be possible to use $CLOS$ instead of $LOOPS$ since there may be circuits in the union of $H_{\mathcal{L}^*_{LOOPS}}(C)$ and $prec'$. In addition, even when this union is cycle free, $\mathcal{L}^*_{CLOS}(C)$ is not necessarily consistent with T_C [HHM93]. The fact that $LOOPS$ does not always respect the local order — here $prec'$— is not a problem: first, $prec'$ has no direct meaning here, it is only a part of T_C; secondly, the part of $prec'$ which is not respected by $LOOPS$ is always a contradictory part of T_C.

3 Implementation Issues

3.1 Global Algorithm

The global function is given below. For short, we write L instead of $\mathcal{L}^*_{LOOPS}$.

FUNCTION $\mathcal{L}^*_{LOOPS}$ (Graph H, Class C) : returns a list
(1) $H_L(C) \leftarrow$ BuildLGraph(H, C)
{*Build $H_L(C)$ from the lists associated with the direct superclasses of C* }
(2) $L_C \leftarrow LOOPS(H_L(C))$
{*Compute LOOPS on $H_L(C)$* }
(3) **If** a superclass of C does not belong to L_C, **then** Exit with failure
---there is a cycle in $H_L(C)$---
else return L_C
End $\mathcal{L}^*_{LOOPS}$

The construction of $H_L(C)$ and the algorithm $LOOPS$ are detailed in next sections. We prove that these two steps have time and space complexity in $\mathcal{O}(size(H_C))$, where the size is the number of edges. More exactly, let p be the number of C direct superclasses, and N_C be the vertex number of H_C. Time complexity is in $\mathcal{O}(p^2 \times N_C)$ and space complexity is in $\mathcal{O}(p \times N_C)$. In comparison, $LOOPS$ and $CLOS$ have time complexity in $\mathcal{O}(size(H_C))$ and no additional space is needed, except for the computed list if it is saved.

3.2 Building the Linearization Graph ($H_L(C)$)

$H_L(C)$ is built by considering the direct superclasses of C, following $prec_C$. For a given C_i, direct superclass of C, the edge CC_i is created, then $L(C_i)$ is traversed. Let $L(C_i) = (C_i = a_1 \ ... \ a_k)$; for all $1 \leq j < k$, an edge $a_j a_{j+1}$ is created, if it does not exist yet. Edges are represented as successor lists —denoted $Succ(y)$, for any y superclass of C. Insertion of $a_j a_{j+1}$ means that a_{j+1} is added at the end of $Succ(a_j)$. Note that, the order in which the direct superclasses of C are considered, and the order in which the successor lists are built, ensure that the successor lists are ordered following the $prec'$ order defined for $H_L(C)$.

FUNCTION BuildLGraph (Graph H, Class C) : returns a graph
(1) $H_L(C)$ vertex set $\leftarrow$ {C superclasses}
For all vertex y of $H_L(C)$,
Initialize $succ(y)$ to the empty list
(2) **For all** C_i direct superclass of C
---according to $prec_C$ order---
Insert C_i at the end of $Succ(C)$
$CurrentClass \leftarrow C_i$
While $CurrentClass$ is not the last element of $L(C_i)$
Let $NextClass$ be the class following $CurrentClass$ in $L(C_i)$
If $NextClass \notin Succ(CurrentClass)$ **then insert** $NextClass$ **at the end of** $Succ(CurrentClass)$
$CurrentClass \leftarrow NextClass$
EndWhile
EndFor
(3) Return $H_L(C)$
End BuildLGraph

Complexity

Suppose that the global inheritance graph H is encoded by a vector of size N, where N is the number of classes of H, with indexes corresponding to classes. Each element of the array is a pointer to a linked list encoding the direct superclass lists of the corresponding class. This structure is space linear in the size of H.

Suppose that $H_L(C)$ is encoded by a pointer array as H, replacing direct superclasses lists by successor lists. This array also has size N. It is used

for any class C. However, for a given C, only indexes corresponding to C superclasses are actually used in $\mathcal{L}^*_{LOOPS}(H, C)$.

$H_L(C)$ and H_C have same vertex set. Initializing the successor lists can be done in traversing H_C, that is in time $\mathcal{O}(size(H_C))$. $H_L(C)$ cannot have more than $\sum_{i=1}^{p} \mid L(O_i) \mid$ edges, where p is the number of direct superclasses of C. Since a vertex of $H_L(C)$ has p successors at most, the time complexity of the second step can be roughly bounded by $p \times \sum_{i=1}^{p} \mid L(C_i) \mid$ or $p^2 * N_C$, where N_C is the number of classes of H_C, since $\sum_{i=1}^{p} \mid L(C_i) \mid < p \times N_C$. We can reasonably assume that the number of direct superclasses of any class is bounded by a low constant and thus the second step is linear in the size of H_C.

Thus, the time complexity of `BuildLGraph(`H`,` C`)` is $\mathcal{O}(size(H_C))$. And the space complexity of `BuildLGraph(`H`,` C`)` is $p \times N_C$, which can be considered to be in $\mathcal{O}(size(H_C))$.

3.3 Computing $LOOPS$ on $H_L(C)$

First, we define $LOOPS$ as applied to an inheritance hierarchy, say H_C. It is straightforward to apply it to $H_L(C)$ instead of H_C.

$LOOPS$ can be recursively defined as follows [Duc91]: $LOOPS(H_C) = (C)$ if C has no direct superclass. Otherwise, let $(C_1 \ldots C_p)$ be the direct superclass list of C ordered by $prec_C$. Then $LOOPS(H_C) = (C + L(C_1) + \ldots L(C_p))$, where "+" denotes the concatenation operator, and, for all $1 \leq i \leq p$, $L(C_i)$ is $LOOPS(H_{C_i})$ restricted to the classes that do not belong to any H_{C_j}, $j <_{prec_C} i$.

$LOOPS$ can be implemented with a linear time and space complexity relatively to the size of H_C. The following is a possible implementation. For the needs of the algorithm, we define an N size vector of positive integers indicating, for each class, the current number of its direct subclasses. Let us call it $SUBS$.

```
FUNCTION LOOPS (Graph H, Class C)
                    : returns a list
(1) For all i superclass of C
        Initialize SUBS[i] to the number of
        direct subclasses of i in H_C
(2) Return STEP(C)
End LOOPS

FUNCTION STEP (Class C) : returns a list
(1) For all y direct superclass of C
    ---according to prec_C order---
        Decrement SUBS[y]
    Endfor
(2) L ← (C)
    For all y direct superclass of C
    ---according to prec_C order---
        If SUBS[y] = 0 then L ← L + STEP(y)
    Endfor
(3) Return L
End STEP
```

Complexity

The recursive function $STEP$ clearly has a linear time complexity relative to the size of H_C, since each edge of H_C is considered only once. The vector $SUBS$ can be initialized by traversing the structure encoding H, considering only the classes of H_C. Thus, the time and space complexity of $LOOPS$ is linear in the size of H_C.

When considering $H_L(C)$ instead of H_C, replace parameter H by H_L, and "direct superclass" by "successor". The previous reasoning about complexity also holds for $H_L(C)$.

4 Perspectives

$\mathcal{L}^*_{LOOPS}$ provides a monotonic linearization which extends known linearizations. The list associated with a class is not built by traversing its hierarchy but by merging the lists of its direct superclasses.

One problem arises when these lists are not consistent, *i.e.* it is not possible to incrementally build a monotonic linearization (remark that the local precedence order of the new class is never the cause of a circuit in the linearization graph). Two main kinds of solutions are imaginable, depending on whether it is allowed to modify the inheritance

graph or not. The first alternative settles the problem of reorganizing the hierarchy. This may imply modifying some local orders or adding new classes. The second alternative sacrifices monotonicity and gives priority to some classes.

Another open question is how to prevent future incompatibilities. One possibility is to maintain a "global linearization graph". This solution is close to Baker's proposition in [Bak91]. Briefly, Baker's proposition involves computing a linear extension (say L_H) of the global inheritance graph, then the list associated with each class C is obtained by restricting L_H to C superclasses. *E.g.* in Figure 4, suppose that C_1 is created before C_2. Then 3 comes before 4 for all future specializations of 3 and 4, notably for C_2. However, a linearization based on this principle is not acceptable (according to the criterion of 2.3).

The system should also be able to propose a "good" local *prec* for a new inserted class, for example a *prec* such that *CLOS* is sound, if any exists, or a *prec* that allows future extensions as far as possible.

A Appendix: More about CLOS and LOOPS

Let us first explain why we use a slightly modified version of *LOOPS*, say *ModLOOPS* (see algorithm in 3.3), instead of *LOOPS* itself. The linearization *ModLOOPS* is exactly *LOOPS* if the inheritance graph has no transitivity edges —an edge xy is a transitivity edge if there is another path from x to y. We have shown [HMHD91] [Huc92] that *LOOPS* does not behave correctly in the presence of transitivity edges. Indeed, a transitivity edge is from our viewpoint a redundant edge that should be ignored by the linearization. Then, the linearization must produce the same result on a graph with transitivity edges as obtained on the latter graph when all transitivity edges are deleted. This property is not fulfilled by *LOOPS* —but is fulfilled by *ModLOOPS* and *CLOS*.

One general method for building a linear extension (of H_C) is to successively choose a minimal class among the remaining classes —a *minimal* class means that all its subclasses in H_C already have been taken. *ModLOOPS* specifies it by applying a *depth-first* principle: it takes a minimal direct superclass of the class, the most recently taken, which possesses one minimal direct superclass (*i.e.* it backtracks as little as possible). When this class (say C') has several minimal direct superclasses, *ModLOOPS* chooses the least class according $prec_{C'}$ order.

CLOS builds a linear extension of H_C, which necessarily respects *prec*; the union of these two relations must have no cycles. *CLOS* successively chooses a $H_C \cup prec$-minimal class among the remaining classes —a $H_C \cup prec$-minimal class is a class whose all subclasses in H_C and all predecessors by *prec* already have been taken. When several classes are available, it takes the $H_C \cup prec$-minimal direct superclass of the class, the most recently taken, which possesses such a superclass (*i.e.* it backtracks as little as possible).

Briefly, *CLOS* is depth-first when possible, *i.e.* when the depth-first principle does not bypass *prec*. In comparison, *ModLOOPS* respects *prec* when possible, *i.e.* when this respect does not bypass the depth-first principle. See [DHHM92] [Huc92] [Mug92] for detailed studies of these algorithms.

References

[Bak91] H. G. Baker. CLOStrophobia : Its Etiology and Treatment. *OOPS Messenger*, 2(4), 1991.

[BCK89] H. Brettauer, T. Christaller, and J. Kopp. Multiple versus Single Inheritance in Object Oriented Programming. *Arbeistpapiere der GMD 415*, 1989.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13 doc-

ument 88-002R. *Sigplan Notices*, 23, 1988.

[BKK+86] 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.

[DG87] L.G. DeMichiel and R.P. Gabriel. The Common Lisp Object System : An Overview. *ECOOP Proceedings*, 1987.

[DH91] 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.

[DHHM92] R. Ducournau, M. Habib, M. Huchard, and ML. Mugnier. Monotonic Conflict Resolution for Inheritance. *OOPSLA'92 Proceedings*, 1992.

[Duc91] R. Ducournau. Y3 : le langage à objets, version 3.6. Technical report, Sema Group, Montrouge, France, 1991.

[HHM93] M. Habib, M. Huchard, and M.L. Mugnier. An incremental and monotonic graph construction. Technical Report 93-049, LIRMM, Montpellier, France, 1993.

[HMHD91] M. Huchard, M.L. Mugnier, M. Habib, and R. Ducournau. Towards a Unique Multiple Inheritance Linearization. *EurOOp'e 91 Proceedings*, 1991.

[Huc92] M. Huchard. *Sur quelques questions algorithmiques de l'héritage multiple.* PhD thesis, Université Montpellier II, France, 1992.

[KMMPN87] B.B. Kristensen, O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. The Beta Programming Language. In B.D. Shriver and P.Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Englewood Cliffs NJ: Prentice Hall, 1988.

[Moo86] D. A. Moon. Object-Oriented Programming with Flavors. *OOPSLA'86 Proceedings*, 1986.

[Mug92] M.L. Mugnier. *Contributions algorithmiques pour les graphes d'héritage et les graphes conceptuels.* PhD thesis, Université Montpellier II, France, 1992.

[Roy91] JC. Royer. A propos des concepts de CLOS. *Proc. JFLA: Langages Applicatifs*, 1991.

[SB86] M. Stefik and D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 6(4), 1986.

[Ste90] G. L. Steele. *Common Lisp : The Language*, chapter 28. Digital Press, 2 edition, 1990.

[Str86] Bjarne Stroustrup. *The C++ programming language.* Addison–Wesley, 1986.