ISSN 0105-8517

Name Collision in Multiple Classification Hierarchies

Jørgen Lindskov Knudsen

DAIMI PB – 250
May 1988

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*

Name Collision in Multiple Classification Hierarchies*

Jørgen Lindskov Knudsen

Computer Science Department, Aarhus University,

Ny Munkegade 116, DK-8000 Aarhus C, Denmark.

E-mail: jlk@daimi.dk

Abstract

Supporting multiple classification in object-oriented programming languages is the topic of discussion in this paper. Supporting multiple classification gives rise to one important question — namely the question of inheritance of attributes with identical names from multiple paths in the classification hierarchy. The problem is to decide how these multiple classification paths are reflected in the class being defined. One of the conclusions in this paper is, that by choosing strict and simple inheritance rules, one is excluding some particular usages of multiple classification. This leads to the notion of attribute-resolution at class definition, which means that the programmer in some cases is forced or allowed to resolve the potential ambiguity of the inherited names. The concept of attribute-resolution is managed through the identification of two conceptually different utilizations of specialization (unification and intersection), and two different attribute properties (plural and singleton) to guide the attribute-resolution.

Introduction

One of the vital issues when designing programming languages or software systems using the object-oriented perspective, is multiple classification. In this paper, we will restrict ourselves to deal with programming languages in the sense that our examples and our terminology are influenced by work done within programming language design. However, we

*Presented at the *European Conference on Object-Oriented Programming* (ECOOP'88), August 15th-17th 1988, Oslo, Norway.

will claim that the discussion is relevant in the design process of object-oriented systems, too. In fact, we find that a major part of the design of an object-oriented system is language design, bringing the following discussion into the realm of object-oriented system design. In the past there have been many proposals for programming language support for object-oriented programming with multiple classification. Some of the most notable proposals are the object-oriented extensions to Lisp: LOOPS[3] and FLAVORS[13], the proposal for multiple classification in the class hierarchy of Smalltalk-80[5], the ThingLab system[4], and the programming languages Galileo[1], Amber[6], Eiffel[10], and C++[11]. The problem of name collision has been dealt with very differently by these proposals. Some of the proposals treat name collision in the hierarchy as illegal; others treat name collisions as separate declarations of equal right, while others treat name collisions as specialization of the attribute. As it can be seen, no general agreement of the treatment of name collision has been reached yet. This paper will examine the underlying issues in order to reach a unified understanding of name collision and also to understand why there isn't *one right* treatment of name collision in class hierarchies with multiple classification. Underlying the discussion is an aim to solve as many name collisions as possible at compile-time, and to ensure the highest degree of polymorphism.

1 Discussion of Object-Oriented Programming and Multiple Inheritance

Object-oriented programming is one of the buzzwords of the eighties of which all agree without agreeing on what it is. In the following, I will shortly discuss my view of object-oriented programming to put this paper into perspective. In object-oriented programming we have shifted our attention from the program text onto the program execution. We look at the program execution as a physical model of some part of the real world. We want to structure both the program text and the actual program execution in a way that reflects this aspect of modeling. We want to be able to identify structures of the program as models of actual phenomena in the part of the world that we want to model.* This inspires us to examine

*In fact, we want to be able to model parts of some imaginary world, too, e.g. during the design of a new application without any predecessors. To ease the writing, we will only use the

the ways in which we as humans conceive and structure our knowledge of the phenomena of the world around us. Object-oriented programming is inspired by three different ways in which humans structure their knowledge. The first structuring mechanism is *classification*, where we identify that a number of different phenomena share some common characteristics. By classifying phenomena, we create concepts. A concept can be described by its name, its intension, and its extension. The *extension* of a concept is the phenomena that can be described by the concept, and the *intension* is a description of the properties which phenomena in the extension possess. Having identified some concepts, we use these concepts to create other concepts. This can be done in two different ways. Either by aggregation or by specialization. Structuring concepts by aggregation is to form a concept by describing the properties of the phenomena by means of other concepts. Specifying a concept as being an aggregate consists of specifying the sub-components and other aspects of the intension (i.e. properties of the aggregate as a whole). Classification and aggregation are not unique to object-oriented programming. In fact, almost any programming language contains language constructs for specifying classification and aggregation (e.g. type systems and record types).

The unique aspect of object-oriented programming is the language support for the structuring mechanism *specialization*. Specialization of concepts supports the specification of concepts as variants of other concepts. When we specialize a concept, we do it either by specifying further properties in the intension or by specializing one or more of the properties in the intension. Specialization of concepts gives rise to a hierarchical structure on the concepts. In object-oriented programming we utilize this hierarchical structure in the class hierarchy, and furthermore we may utilize the structure to support polymorphic programming. By polymorphic programming as part of object-oriented programming we understand the ability to utilize the hierarchical structure of concepts in the specification of e.g. parameters. Let us assume that we have a parameterized program fragment, where one of the parameters is specified as class A. Then the program fragment can be compile time checked with respect to the legal usages of this parameter, since the manipulations of the parameter is legal, as long as it is manipulated according to the specification given in class A. Since specialization is property preserving (i.e. if an

term "part of the world" instead of "part of the world or part of some imaginary world".

instance of class A has property x, then instances of all specializations of A will have the same property x) it is now possible to use instances of all specializations of class A as actual parameter to the program fragment, since they will possess at least the properties described in class A. We do not say that specialization is semantics preserving, since most language constructs for specialization do not necessarily preserve the semantics of the attribute (especially of actions), although this is the ultimative goal. For a more detailed discussion of this approach to object-oriented programming, see [8] and [12].

Looking at existing specialization hierarchies reveals that very many of them cannot be described by tree-structured hierarchies, since they contain concepts which are specialized from at least one general concept but along two (or more) paths in the hierarchy. The language support for specialization in many programming languages allows only for tree-structured classification hierarchies, thus limiting the expressive power of the language. The concept of multiple inheritance (or multiple classification) is *one* approach to loosen this limitation.

2 Definition of Terms

In order to ease the following discussion, a few terms need to be clarified. In fact, these terms are used in the above discussion in accordance with these definitions. By the term *hierarchy*, we will mean structures that can be described by acyclic, directed graphs.

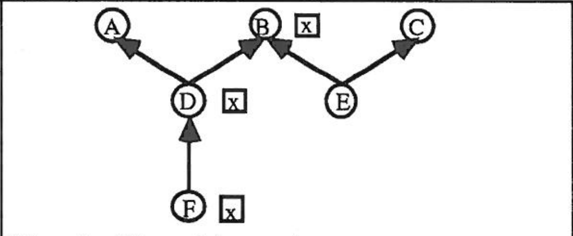


Fig. 1: Class hierarchy

The term *class hierarchy* will be used to cover the hierarchy of all the classes in a given system and their sub/super-class relationships. As described above, the class hierarchy can be used both for specifying inheritance of properties and for polymorphic programming.

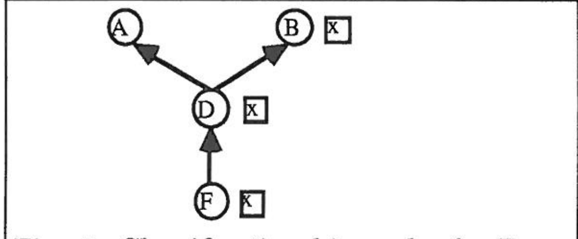


Fig. 2: Classification hierarchy for F

The term *classification hierarchy* will be used to cover that part of the class hierarchy which is involved in the classification of one particular class. That is, there is one class hierarchy in a system, but several classification hierarchies (one for each class in the system). In fact, the class hierarchy is the union of all classification hierarchies.

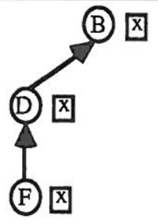


Fig. 3: Inheritance hierarchy w.r.t. x

The term *inheritance hierarchy* of a class with respect to a particular attribute is the part of the classification hierarchy that covers the inheritance paths of that attribute.

Name collision can arise in two different ways depending on whether the collision is a result of the presence of more than one super-class, or whether the collision arises because of ambiguities between the class itself and its super-classes. It clarifies the discussion if these two types of name collision are separated.

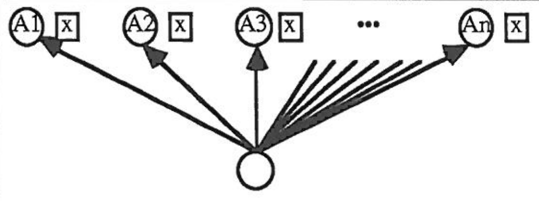


Fig. 4: Horizontal name collision

The term *horizontal name collision* will be used to cover name collisions, where a class inherits several attributes with the same name from different super-classes.

Note that horizontal name collision cannot arise in tree-structured hierarchies.

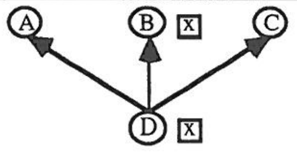


Fig. 5: Vertical name collision

The term *vertical name collision* will be used to cover name collisions, where a class defines an attribute with the same name as one (or more) attributes, inherited from one of its super-classes.

Please note, that both horizontal and vertical name collision might be involved in a particular name collision, if e.g. in fig. 5, class A and/or class C has an x-attribute, too.

3 Issues of Name Collision

Essentially, there are three different views on the consequences of a name collision. We say that a name collision is *intended* if different attributes with the same name describe the same phenomenon. We say that a name collision is *casual* if different attributes with the same name describe different phenomena. And we say that a name collision is *illegal* if the relation between attributes, names and phenomenon must be unique. In the following sections, we will discuss these views in more detail.

3.1 Intended Name Collision

When a name collision is regarded as an intended name collision, we are actually dealing with one attribute (defined by the name; i.e. the relation between attribute name and phenomenon is unique). The attribute will have several specifications (one for each inherited attribute, and possibly one in the class itself) which together must constitute the full specification of the unique attribute. In a programming language where all name collisions are regarded as intended, the phenomenon is modeled by one attribute with multiple specifications. This means that, in order to be able to ensure the polymorphic property of the classification hierarchy, the specifications must not be in conflict. If there are conflicts, it is impossible to unify these separate specifications into one specification for the phenomenon. It is not obvious in which situations we are able to ensure the polymorphic property. Let us consider an example:

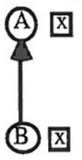


Fig. 6a: Intended vertical name collision

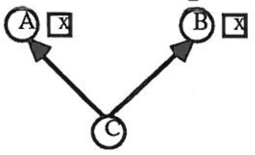


Fig. 6b: Intended horizontal name collision

If we have an intended vertical name collision (as in fig. 6a), we can ensure the polymorphic property if we know that the specification of B.x is a specialization of the specification of A.x. This specialization property can be ensured by classification hierarchies on the specifications. This approach to specifications is taken by the Beta language[9]. Another example is type-hierarchies as exemplified in the language Amber[6]. The situation is more complex, when we consider intended horizontal name collision (as in fig.6b). At least four different situations might arise:

1. The domains of A.x and B.x might be *disjunct.*
   This might happen if A.x is a variable of type integer in the range: 1–99, and B.x is a variable of type integer in the range: 200–1000.

2. The domains of A.x and B.x might be *inconsistent.*
   This might happen if A.x and B.x both model temperature but the domain of A.x is Fahrenheit, whereas the domain of B.x is Celsius.

3. A.x and B.x might be of *different nature.*

This might happen if A.x is a variable of some type (e.g. integer), whereas B.x is an operation.

4. The classes of A.x and B.x might have *a common superclass.* In this case, the two attributes are to some extent related, and it might therefore be plausible to consider them as different views on the same attribute.

3.2 Casual Name Collision

When a name collision is considered casual, we are allowing several attributes with the same name but with different, and not necessarily related specifications. In this situation it is important to be able to distinguish between the different attributes by some means other that their names. The most immediate solution is to *qualify* attribute names with the name of the class from which it is inherited (this qualified name is unique). That is, in fig. 6a it must be possible to denote both A.x and B.x, whereas in fig. 6b it must be possible to denote both A.x, B.x, and C.x. In the case of casual name collision, it is useful to use horizontal and vertical overwriting. *Horizontal overwriting* means that in class B in fig. 6a it is possible to state that the x-attribute of B is e.g. A.x. *Vertical overwriting* means that in class C in fig. 6b it is possible to state that the x-attribute of C is e.g. B.x. Horizontal and vertical overwriting does not exclude the possibility of denoting the other x-attributes by qualification — it is merely a short-hand.

3.3 Illegal Name Collision

When a name collision is considered illegal, the relation between names, attributes and phenomena must be unique. This means that name collisions will always give rise to compile-time errors, and not run-time errors.

3.4 Summary of Name Collision

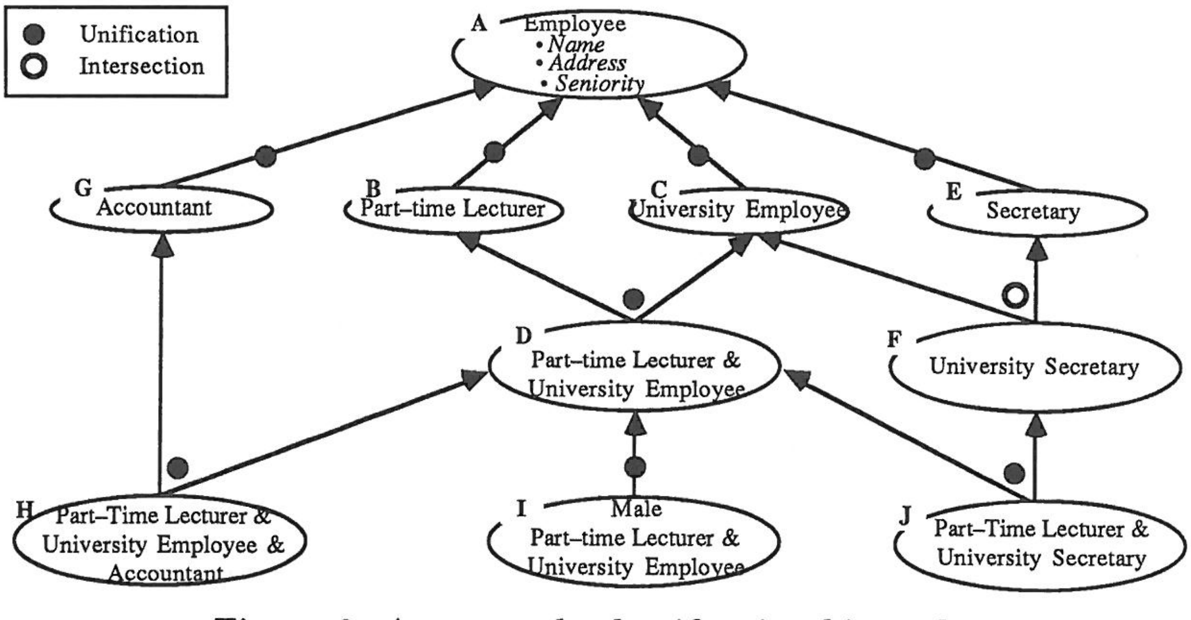
The above discussion can be summarized by the following table:

	Intended	Casual	Illegal
Horizontal	• Disjunct • Inconsistent • Different nature • Specializations	• Qualification • Horizontal overwriting	
Vertical	• Specializations	• Qualification • Vertical overwriting	

Figure 7: Summary of issues of name collision

4 The Need for Programmer Control

By examining a selected classification hierarchy, we find that all three views on name collision are useful, and each corresponds to different aspects of programming and modeling, and that choosing one particular interpretation will result in the inability to express certain structures. That is, using one particular view in connection with either vertical or horizontal name collision is a matter of choosing to express one particular relationship between the inherited attributes, and the specified attribute (if there is such one).



● Unification
○ Intersection
A Employee • Name • Address • Seniority
G Accountant
B Part–time Lecturer
C University Employee
E Secretary
D Part–time Lecturer & University Employee
F University Secretary
H Part–Time Lecturer & University Employee & Accountant
I Male Part–time Lecturer & University Employee
J Part–Time Lecturer & University Secretary

Figure 8: An example classification hierarchy

To guide the discussion of the various possibilities involved in dealing with name collisions in multiple classification hierarchies, we will examine the classification hierarchy in figure 8. Let us assume that we are in the process of developing an accounting system for the university administration. Currently we are focusing on the structures for handling the data concerning employees (name, address, job-category, salary, etc.). Assume that we are utilizing an object-oriented system, and that some part of the classification hierarchy is developed outside our organization (that is, we cannot make changes to parts of the system — only expand those parts). There is one important requirement; namely that an employee must only be represented as one employee-object in the system.

We have only identified three attributes of the employee class to support the discussion. Further attributes may be specified in both the employee class and in the shown specializations (e.g. accountant). In the hierarchy there are four examples of multiple classification, namely classes D, F, H and J. Now looking at the attributes *Name*, *Address*, and *Seniority* there is no doubt that throughout the entire hierarchy, the attributes *Name* and *Address* are singleton (i.e. any instance of any class in the hierarchy will only have one *Name* and one *Address* attribute). The question is more subtle when we consider the *Seniority* attribute. In that case we often find that a single person is employed at the same university in more that one position at a time. As an example, we may find a person who is professor at one department, but at the same time part-time lecturer at another department. And he may even be accountant of some foundation, administrated by the university in question. The *Seniority* attribute is concerned with the seniority of the person as employed in the particular job-category. Since we know that any employee is employed in at least one job-category, it is meaningful to specify that any employee-object has a *Seniority* attribute. But the seniority of one particular person is dependent on whether we consider his/her seniority as e.g. accountant or as e.g. part-time lecturer. This might lead to specifying that the *Seniority* attribute should be inherited down the hierarchy with duplicates when multiple classification is involved. This is, however, erroneous since the class F models secretaries employed at the university, and as such secretary instances should only have one *Seniority* attribute. In fact, the following table indicates the intended distribution of *Seniority* attributes in this small class hierarchy:

	A	B	C	D	E	F	G	H	I	J
Number of Seniority attributes	1	1	1	2	1	1	1	3	2	2

Figure 9: Table showing number of *Seniority* attributes

The problem is how do we obtain the situation where some name collisions are treated as intended, others as casual, and yet others as illegal? It is obvious that choosing one particular view on name collisions (e.g. casual name collision) will not result in the above table.

5 Discussion of Solutions

Looking at the classification hierarchy in figure 8 and the table in figure 9, one can see that the specialization taking place in the specification of class D and H is different from the specialization of class C and E to class F. If we look at (G,D) $\Rightarrow$ H and (C,E) $\Rightarrow$ F, we will see that the number of *Seniority* attributes in H resp. F will be either 3 resp. 2, or 1 resp. 1. This inspires to look closely at the underlying semantics of the classes H and F. Class H models employees who at the same time are employed as part-time lecturer, university employee, and accountant; that is, holding three job-positions, whereas class F models employees who are secretaries at a particular university; that is, holding only one job-position. This gives us the motivation for introducing two different specialization methods. The first specialization method (called *unification*) takes care of the kind of specialization where the specialized class is supposed to model the unification of all the classes in its classification hierarchy; that is, if a horizontal name collision should occur, it should be treated as a casual horizontal name collision, giving rise to multiple attributes with the same name. We call such a class an unification class.[†] The second specialization method (called *intersection*) takes care of the kind of specialization where the specialized class is supposed to model the intersection of all the classes in its classification hierarchy; that is, if a horizontal name collision should occur, it is treated as an intended horizontal name collision if the attribute for all the immediate superclasses is inherited from a common superclass. We call such a class an intersection class. To motivate this rule, let us look at figure 8 and assume that class D is an intersection specialization. Then the name collision of the two *Seniority* attributes from B resp. C is treated as an intended horizontal

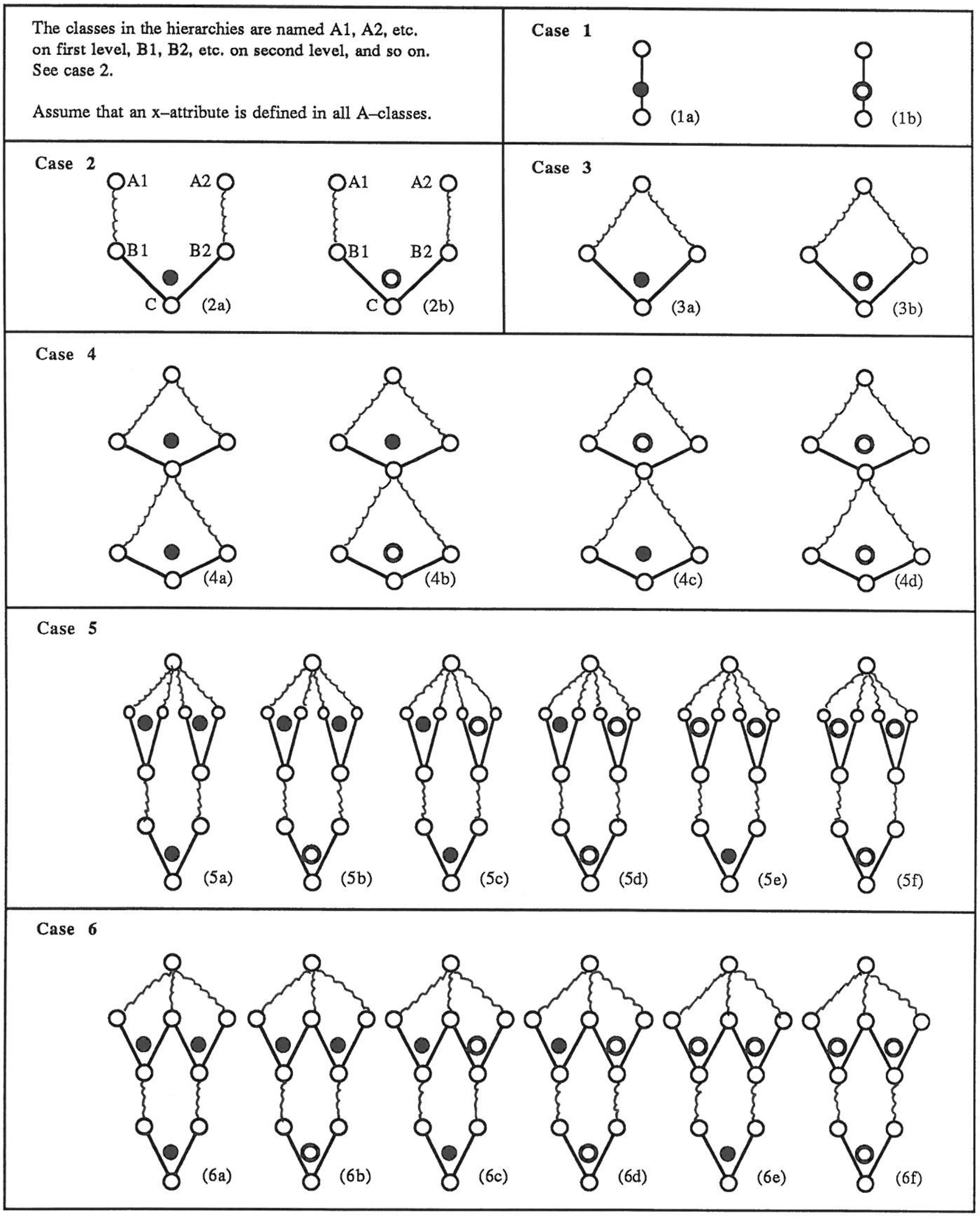
[†]Please note, that unification is conceptually different from aggregation, since a unification of classes A, B and C specifies that the unified class can be approached from three different perspectives (namely those perspectives that are defined by the classes A, B and C). This is called subtyping by combination in [7]. An aggregation of classes A, B and C specifies that the aggregated class is composed of an instance of class A, an instance of class B, and an instance of class C. This is called subtyping by composition in [7], and part hierarchy in [2].

name collision, since the attribute originates from class A which is a common superclass of both B and C. If classes B and C, on the other hand, both had an x-attribute (not inherited from class A), the name collision in D would be treated as a casual horizontal name collision. Now, applying the above rule to the hierarchy in figure 8 will give us the intended distribution of the *Seniority* attribute, if we assume that classes D and H are unification classes, and class F is an intersection class. If we however look at the *Name* and *Address* attributes, we do not get the intended distribution, since there will be multiple copies in the classes D and H. This is highly undesirable, since it may give rise to inconsistencies in the contents of these different copies of this semantically identical attribute. It is therefore not sufficient to device two different specialization methods — we have to specify inheritance properties of individual attributes, too. We will therefore introduce the concept of *singleton* attributes with the semantics that they may only exist in one copy in any of the future specializations of the class. All other attributes are said to be *plural.* The singleton property is associated with the attribute in the class that initially declared the attribute. Looking at figure 8, the attributes *Name* and *Address* must be specified as singleton in class A in order to obtain the desired distribution of attributes.

Of course, it may be possible to specify the singleton property on a class as a whole implying that all attributes of the class are given the singleton property. This is merely a shorthand for the common case where the whole class is shared by all subclasses in the classification hierarchy. Singleton classes are very similar to virtual classes in the proposal for multiple inheritance in C++[11].

6 Discussion of Unification and Intersection Inheritance

The detailed properties of unification and intersection inheritance can be discussed in detail by examining the cases outlined in figure 10. These cases illustrate the various possible types of hierarchies that may arise in multiple classification hierarchies. In the following, I will give some comments on the most important cases in order to justify the formal rules for inheritance in multiple classification hierarchies that are given in section 7.


The classes in the hierarchies are named A1, A2, etc. on first level, B1, B2, etc. on second level, and so on. See case 2.

Assume that an x–attribute is defined in all A–classes.

Figure 10: Important Multiple Classification Hierarchies

Case 1: Single Inheritance

If class B inherits from class A using single inheritance, name collision is dealt with using the well-known rules from tree-structured classification. We will not discuss this case further in this article.

Case 2: Disjoint Multiple Inheritance

The case of disjoint multiple inheritance is the place where we decide to consider some name collisions as being illegal.

2(a): Unification

When disjoint hierarchies are combined using unification inheritance, we consider name collisions as being casual, and allow duplicate instances of attributes having the same name. The reason is, that we want to combine two independent hierarchies. An example is combining a hierarchy concerning job type (teacher, secretary, trucker, etc.) with a hierarchy concerning nationality (Danish, Swedish, American, etc.). If there is an attribute X in both hierarchies, then this attribute will not be considered as being the same attribute (i.e. the two hierarchies use the same name X by coincidence).

2(b): Intersection

When hierarchies are combined using intersection inheritance, we are stressing that the involved hierarchies are considered as mutually contributing to the full specification of the new class. That is, the new class is created by merging attributes. In the case of a name collision, we have to consider whether it makes sense to merge the attributes. If the attributes are not defined in a common superclass (that is, the attributes have each their own defining statement), then there is no way to ensure, that the attributes are related in any way (see section 3.1), and any automatic rule must consider name collisions in disjoint intersection inheritance as *illegal* name collisions.

Case 3: Simple Multiple Classification

In this simple case of multiple inheritance, the classification hierarchies of the superclasses share a common superclass in which the X attribute is defined (and no multiple inheritance is involved in the superclass hierarchies).

3(a): Unification

The same as disjoint unification above, giving rise to two X attributes in class C.

3(b): Intersection

In this case, the superclass hierarchies share a common superclass in which the X attribute is defined, and it is therefore possible to assure that the inherited X attributes are related and thus it makes sense to merge them into one attribute.

Case 4: Chained Multiple Classification

Chained multiple inheritance is similar to simple multiple inheritance, so only two sub-cases need to be commented on:

4(a): Chained Unification

When a unification class (e.g. C) is a common super-class in another unification class (e.g. E), all attributes are inherited along all available paths in the resulting inheritance hierarchy for E w.r.t. X. In this case giving rise to four X attributes in E. It may be argued, that class E only contains two X attributes, and the formal rules in section 7 can relatively easy be modified to reflect such a decision.

4(c): Unification after intersection

Having intersected an attribute does not imply that the attribute is made singular (i.e. further specializations may contain duplicates of the attribute), but merely that at this level in the hierarchy, the attribute is unique. If this attribute is further inherited multiple times in a unification, it will give rise to multiple instances of the attribute. In this case, class E will possess two X attributes.

Case 5: Two-level Multiple Classification

The case of two-level multiple classification deals with the case where an attribute is defined in a common superclass, and that class is (independently) specialized into several classes. These specialized classes are then involved in separate multiple classifications that in turn are classified into one class using multiple classification. None of those cases needs further comments.

Case 6: Two-level Mixed Multiple Classification

This case is similar to case 5, except that the specialized classes are involved in multiple classifications that are overlapping. Here cases 6(a), 6(c), and 6(e) need to be commented on:

6(a): Two-level Mixed Multiple Unification
This case is very similar to case 4(a), leading to four X attributes in class E. However, similar to case 4(a), it may be argued, that class E only contains three X attributes, and the formal rules in section 7 can relatively easy be modified to reflect such a decision.

6(c): Merging
In this case, the X attribute which is inherited along the path A-$B2$-$C1$-$D1$ is merged into the X attribute inherited along path A-$B2$-$C2$-$D2$ since the origin (class $B2$) is involved in both class $D1$ and $D2$.

6(e): Unification after overlapping Intersection
With respect to the number of X attributes, this case is identical to case 4(c).

7 Formal Specification of Inheritance Rules

After the above discussion of the objectives for inheritance rules, it is time to state them more formally. The formal rules will be followed by the rationale for the intrinsics of the rules.

7.1 Notation

$X_B^{singleton}$	$\equiv$ The set of all paths in the inheritance-hierarchy for class B w.r.t. the singleton attribute X. If X is not a singleton attribute of B, $X_B^{singleton} = \emptyset$.
X_B^{plural}	$\equiv$ The set of all paths in the inheritance-hierarchy for class B w.r.t. the plural attribute X. If X is not a plural attribute of B, $X_B^{plural} = \emptyset$.
X_B^{merged}	$\equiv$ The set of merged attributes named X in class B. Each element in the set is a set containing all the paths, contributing to the particular merged X. If X is not a plural attribute of B, $X_B^{merged} = \emptyset$.
$super(B)$	$\equiv$ The set of all superclasses of class B.
$X_{super(B)}^{singleton}$	$\equiv \bigcup_{A \in super(B)} X_A^{singleton}$
$X_{super(B)}^{plural}$	$\equiv \bigcup_{A \in super(B)} X_A^{plural}$
$X_{super(B)}^{merged}$	$\equiv \bigcup_{A \in super(B)} X_A^{merged}$
$\bigcup X_{super(B)}^{merged}$	$\equiv \left\{ p \in S \mid S \in X_{super(B)}^{merged} \right\}$

Comments on Notation

The sets $X_B^{singleton}$, X_B^{plural} and X_B^{merged} contain the information necessary to decide the number of X-attributes in B, divided into three categories, depending on whether the particular X-attribute is *singleton*, *plural* or *merged*. When an instance of class B is created, the sets are used in the following way:

For each element in X_B^{plural}, an X-attribute is instantiated with the qualification given by the path in the classification hierarchy for B specified by that element. Intuitively, this rule states that a plural attribute is to be instantiated with the full qualification along its (unique) inheritance path in the classification hierarchy for its class.

For each element in X_B^{merged}, an X-attribute is instantiated with a qualification that is given by the path in the classification hierarchy for B, specified by the *longest common path* (LCP) of that element. LCP is the longest common path of the set. Intuitively, this rule states that a merged attribute cannot always be instantiated with the full qualification along its inheritance path in the classification hierarchy for its class since that path is not necessarily unique. A merged attribute may be inherited along multiple paths in the classification hierarchy (as with the *Seniority* attribute in class J in figure 8). Since these multiple paths may have conflicting qualifications for the attribute, the only qualification that can be ensured for the attribute is the qualification which all inheritance paths for that particular instance of the attribute agree upon, namely that given by the nearest branching node in the inheritance hierarchy (defined by LCP).

If $X_B^{singleton} \neq \emptyset$, one attribute is instantiated with a qualification given by $LCP(X_B^{singleton})$.

The inheritance paths are represented in the sets in the following way: Upper-case letters indicate names of classes, lower-case letters indicate sub-paths. The classes in the path are separated by either '.', '†' or '‡' where '$A.B$' indicates that B inherits from A using single inheritance, '$A\dagger B$' indicates that B inherits from A using unification inheritance, and '$A\ddagger B$' indicates that B inherits from A using intersection inheritance. In path expressions, '$\star$' is taken to denote any of '.', '†' or '‡'. For any path p, $|p|$ represents the length of the inheritance path (i.e. the number of classes along the path).

7.2 Plural and Singleton definition in B

$$X_B^{plural} = \{B\} \cup X_{super(B)}^{plural}$$

$$X_B^{singleton} = \{B\} \cup X_{super(B)}^{singleton}$$

Comments on Plural and Singleton Definition Rules

The rules for plural and singleton definitions state that defining a plural attribute in a class gives rise to one more potential attribute of that name in all subclasses of that class. Defining a singleton attribute implies the presence of at least that instance of the attribute. In both cases, the definitions from the superclasses are retained.

7.3 Single inheritance in B

$$
\begin{aligned}
X_B^{singleton} &= \left\{ s.B \mid s \in X_{super(B)}^{singleton} \right\} \\
X_B^{plural} &= \left\{ p.B \mid p \in X_{super(B)}^{plural} \right\} \\
X_B^{merged} &= \left\{ \{ s.B \mid s \in S \} \mid S \in X_{super(B)}^{merged} \right\}
\end{aligned}
$$

7.4 Unification inheritance in B

$$
\begin{aligned}
X_B^{singleton} &= \left\{ s\dagger B \mid s \in X_{super(B)}^{singleton} \right\} \\
X_B^{plural} &= \left\{ p\dagger B \mid p \in X_{super(B)}^{plural} \wedge \left(\nexists a\ddagger s \in \bigcup X_{super(B)}^{merged} : p = a\star p' \wedge |a| > 0 \right) \right\} \\
X_B^{merged} &= \Big\{ \{ s\dagger B \mid s \in S \} \cup \\
&\qquad \{ a\ddagger p\dagger B \mid a\star p \in X_{super(B)}^{plural} \wedge (\exists a\ddagger s \in S : |a| > 0) \} \mid S \in X_{super(B)}^{merged} \Big\}
\end{aligned}
$$

Comments on Unification Inheritance Rules

The case of unification inheritance is complex. Let us first comment on the rule for X_B^{plural}. This rule states that the set of X-attributes in B is those attributes inherited from the superclasses that have not been merged in any of the superclasses. The exact rule states that a plural attribute is turned into a merged attribute, iff the plural attribute is inherited along the same path as one of the paths of the merged attribute up to the node in the classification hierarchy where the attribute is made merged. Intuitively this implies that name collisions are considered casual unless the name collision is between a plural and a merged attribute with a common subpath.

The first part of the rule for X_B^{merged} states that all merged attributes in the superclasses are inherited unconditionally. The second part of the rule states the actual merging of a plural attribute into a merged attribute. Exactly this is done by injecting the (slightly modified) path of the plural attribute into the element of X_B^{merged} containing the common subpath. Intuitively this implies that the specification of that particular instance of the merged attribute is extended to include the specification given by the plural attribute. This implies that the name collision is considered intended.

7.5 Intersection inheritance in B

$$\text{if} \quad \exists a : \Big(|a| > 0 \wedge \forall p \in X^{plural}_{super(B)} \cup \bigcup X^{merged}_{super(B)} : p = a{\star}p' \vee p = a\Big)$$

$$\text{then } X^{singleton}_{B} = \left\{ s{\ddagger}B \mid s \in X^{singleton}_{super(B)} \right\}$$

$$X^{plural}_{B} = \emptyset$$

$$X^{merged}_{B} = \left\{ \left\{ s{\ddagger}B \mid s \in X^{plural}_{super(B)} \cup \bigcup X^{merged}_{super(B)} \right\} \right\}$$

else **illegal name collision on X in B**

Comments on Intersection Inheritance Rules

The conditional part of the rule states that intersection is legal iff all inherited plural attributes share a common subpath in their inheritance path. If not, the name collision is considered illegal, since the origin of the different plural attributes is distinct parts of the hierarchy with no connections in the inheritance path of attributes.

If, on the other hand, the inherited plural attributes share a common subpath, they are all merged into one merged attribute. Intuitively this rule states that intersection is like unification except that all inherited plural attributes must be considered as all contributing to one particular attribute. If this is not possible, the name collision is considered illegal.

8 Resolving Conflicts

The above inheritance rules do not resolve the conflicts, but merely restrict the possibilities for conflicts. Now assume that we want to access the x-attribute of class B. If only one x-attribute exists in $X^{merged}_{B} \cup X^{plural}_{B}$, no ambiguity arises. But if more than one exist, the specification of the x-attribute must be unambiguous. We will propose the following rule: If no further specification is given, the x-attribute in X^{merged}_{B} will be selected, if the set is not empty. If the set is empty, or if one of the x-attributes in

X_B^{plural} is wanted, a qualification must be given to resolve the ambiguity. This qualification can be given by specifying a subpath of the inheritance path of the attribute in the inheritance hierarchy for the attribute from the present class and to a node, where the attribute is unique (according to the above rule). This implies that the above mentioned node must be either the defining node of the attribute, or a node where the attribute has taken part of an intersection inheritance.

9 Horizontal and Vertical Overwriting

Instead of resolving conflicts (as described above) each time it may arise, it might be easier to allow for horizontal or vertical overwriting. If this is allowed, the above rule for resolving conflicts must be extended to check for the existence of an overwriting, in which case the qualification can be considered unambiguous. Naturally, further qualification is allowed, and sensible, in the case where the overwriting is not what is wanted in the present situation.

9.1 Explicit Inheritance

In section 7, the default rules for inheritance of attributes in a multiple classification hierarchy are given. The reasons for the rules are based on the recognition of the classification hierarchy as a conceptual hierarchy. There might be cases in which the expected inheritance does not conform with the conceptual hierarchy. This might be the case in situations where the classification hierarchy is slightly ill-suited for the specific application, but where reorganization for some reason is not applicable.

In these situations some restricted usage of horizontal overwriting can be allowed. In fact, in this situation we are dealing with intended name collision which cannot be dealt with by the default rules. Returning to section 3.1, four situations might arise. The case of the inherited attributes being disjunct cannot be remedied by horizontal overwriting. The next two cases can be remedied by horizontal overwriting by means of language constructs for specifying transformations between two domains, or to a lesser extent by language constructs for interchanging variable denotations with operation denotations. The case of the inherited attributes having a common superclass is the most easily remedied case, since it only requires abilities similar to the abilities already used in intersection inheritance and further described in [12].

Vertical overwriting is the usual method redeclaration in Smalltalk-80, or the virtual declaration in languages like Simula67, C++ or Beta. The most interesting version of virtual declaration is the one in Beta, since virtual declarations in Beta to some degree ensure that inherited properties are not invalidated by the vertical overwriting.

10 Conclusion

The prime result here is the recognition of the need for supporting all three views on name collision in one programming language, unless one accepts to be unable to express certain structures. As stated in the introduction, the discussion in this paper is in itself valuable as many proposals have been put forward, and in many cases the arguments and the discussion of the alternatives aren't given. This paper offers such a discussion relieved from the burden of promoting one particular solution at the same time. It will thus hopefully be a source of inspiration for future designers of programming languages with support of classification hierarchies with multiple classification.

References

1. A. Albano, L. Cardelli & R. Orsini: *Galileo: A Strongly-Typed, Interactive Conceptual Language*, ACM TODS, **10**(2), 230–260 (June 1985).

2. E. Blake & S. Cook: *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Paris, France, June 1987.

3. D.G. Bobrow & M.J. Stefik: *Loops — Data and Object-Oriented Programming for Interlisp*, Discussion papers, Proceedings of the European Conference on AI, Orsay, France, July 1982.

4. A. Borning: *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM TOPLAS, **3**(4), 353–387 (October 1981).

5. A. Borning & D.H.H. Ingalls: *Multiple Inheritance in Smalltalk-80*, Proceedings of the National Conference on AI, Pittsburgh, PA, 1982.
6. L. Cardelli: *Amber*, AT&T Bell Labs Technical Memorandum, 11271–84092–410TM, 1984.
7. D.C. Halbert & P.D. O'Brien: *Using Types and Inheritance in Object-Oriented Languages*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Paris, France, June 1987.
8. J. Lindskov Knudsen & K. Stougård Thomsen: *A Conceptual Framework for Programming Languages*, Computer Science Department, Aarhus University, DAIMI PB-192, 1985.
9. B. Bruun Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen & K. Nygaard: *The Beta Programming Language*, in B.D. Shriver & P. Wegner (eds.): *Research Directions in Object-Oriented Programming*, MIT Press, 1987.
10. B. Meyer: *Eiffel: Programming for Reusability and Extendibility*, ACM Sigplan Notices, **22**(2), 85–94 (February 1987).
11. B. Stroustrup: *Multiple Inheritance for C++*, Proceedings of the Spring '87 EUUG Conference, Helsinki and Stockholm, May 1987.
12. K. Stougård Thomsen: *Inheritance on Processes, Exemplified on Distributed Termination Detection*, International Journal of Parallel Programming, **16**(1), 17–52 (1987).
13. D. Weinreb & D. Moon: *Flavors: Message Passing in the Lisp Machine*, MIT AI Memo No. 602, November 1980.