é

A Meta-modeling Semantics of Multiple Inheritance

Roland Ducournau [a,*], Jean Privat [a,b]

[a] *LIRMM – CNRS and Université Montpellier II, 161 rue Ada, 34000 Montpellier, France*
[b] *Dép. d'Informatique, UQAM, 210, avenue du Président-Kennedy, Montréal, QC H2X 3Y7, Canada*

Abstract

Inheritance affords to object-oriented programming its great powers of reusability. When inheritance is *single*, its specifications are simple and everybody roughly agree with them. On the contrary, *multiple inheritance* yields ambiguities that have provoked long-lasting debates and there are no two languages which agree on its specifications.

In this paper, we present a semantics of multiple inheritance based on meta-modeling. A metamodel is proposed, which distinguishes the 'identity' of properties from their 'values' or 'implementations'. It yields a clear separation between syntactic and semantic conflicts. The former can be solved in any language at the expense of a common syntactic construct, namely full name qualification. On the contrary, semantic conflicts require a programmer's decision and the programming language must provide some help to the programmer. The paper surveys the approach based on *linearizations*, which has been studied in-depth, and proposes some extensions. As it turns out that only static typing takes full advantage of the metamodel, the interaction of multiple inheritance and static typing is also considered, especially in the context of virtual types. The solutions proposed by the various languages with multiple inheritance are confronted with the metamodel and implementation issues are finally examined. All the paper long, the difficulties entailed under the *open world assumption* are stressed.

Key words: linearization, meta-modeling, mixins, multiple inheritance, object-oriented programming, overloading, overriding, redefinition, static typing, virtual types

1. Introduction

> Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage. Taivalsaari (1996)

Class specialization and inheritance represent key features of object-oriented programming and modeling. Introduced in the Simula language (Birtwistle et al., 1973), they have been related to the Aristotelian syllogistic (Rayside and Campbell, 2000b,a; Rayside and Kontogiannis, 2001) and they greatly contribute to the way the object-oriented approach meets such software engineering requirements as *reusability*.

In spite of the Taivalsaari's quotation above, inheritance is relatively simple when it is *single*, i.e. when a class cannot have more than one *direct superclass*—hence, the class specialization hierarchy is a tree (or a forest). This is however a strong limitation and, from the very beginning of object-oriented programming, attempts have been made to soundly specify *multiple inheritance* in the pioneer object-oriented languages,

* Corresponding author.
Email addresses: ducour@lirmm.fr (Roland Ducournau), privat@cs.purdue.edu (Jean Privat).

Flavors (Weinreb and Moon, 1980), Smalltalk (Borning and Ingalls, 1982) and Simula (Krogdahl, 1985). It quickly appeared that multiple inheritance was not as simple as single inheritance—*conflicts* may occur, that make the behaviour hard to specify and give to the quotation its full meaning. Different trends have divided the object-oriented programming community, each one advocating for a preferred policy.

(i) A few production languages, mainly Smalltalk (Goldberg and Robson, 1983), are in pure single inheritance; their key feature is *dynamic typing*.

(ii) Several production languages like C++ (Stroustrup, 2000), Eiffel (Meyer, 1992, 1997) or Clos (DeMichiel and Gabriel, 1987; Steele, 1990) propose full multiple inheritance. They have been designed in the 80s and Python (van Rossum and Drake, 2003) is one of the few from the 90s; all of them are widely used but they have been strongly discussed, e.g. (Baker, 1991; Snyder, 1991; Cargill, 1991; Waldo, 1991; Sakkinen, 1992) and they all behave in a different way in face of multiple inheritance.

(iii) Some research languages—Scala (Odersky et al., 2008) is the most representative among recent ones—and a very few production languages like Ruby (Flanagan and Matsumoto, 2008) are based on *mixins* (Bracha and Cook, 1990).

(iv) In the *static typing* setting, a major trend has been inaugurated by Java *interfaces* (Grand, 1997)—classes are in single inheritance but *multiple subtyping*, as a class can implement several unrelated interfaces; many recent languages, e.g. C# (Microsoft, 2001) and .Net languages, follow this trend.

Besides these programming languages, the main (or even single) modeling language, Uml (OMG, 2004), includes multiple inheritance without any precise specification.

The need for multiple inheritance gave also rise to a long-lasting debate. However, the fact that few statically typed languages use pure single inheritance, i.e. *single subtyping*, strongly indicates the importance of multiple inheritance. The rare counter-examples, such as Oberon (Wirth, 1988; Mössenböck, 1993), Modula-3 (Harbinson, 1992) or Ada 95 (Barnes, 1995), result from the evolution of non-object-oriented languages. Furthermore, the absence of Java-like multiple inheritance of interfaces was viewed as a deficiency of the Ada 95 revision, and this feature was incorporated in the next version (Taft et al., 2006). The requirement for multiple inheritance is less urgent in the dynamic typing framework—for instance, all Java multiple subtyping hierarchies can be directly defined in Smalltalk, by simply dropping all interfaces. Conversely, statically typing a Smalltalk hierarchy only involves adding new interfaces for *introducing* the methods which are *introduced* by more than one Smalltalk classes [1] .

Overall, despite the numerous works that have been dedicated to it, multiple inheritance remains an open issue. From this standpoint, there is no satisfactory language. As we will see, even languages based on multiple subtyping are flawed. In this paper, we propose a semantics of class specialization and inheritance which is 'natural'—one might say 'Aristotelian'. This semantics is based on *meta-modeling*, i.e. reifying the entities which are concerned, namely classes and properties. The proposed metamodel is the simplest metamodel that models classes and properties in such a way that all names in the program code can denote a single instance of the metamodel. It allows us to get rid from names and their associated ambiguities and instead just consider reified entities. The first benefits drawn from this metamodel is to precisely distinguish the 'identity' of a property from its 'value' (or 'implementation') in a given class. In turn, it yields a strong distinction between two kinds of conflicts which should not be confused but are actually confused in all known languages with multiple inheritance. The first kind of conflicts involves the property names and occurs when a class inherits two properties with the same name or signature, which have been however introduced in unrelated classes. The second kind of conflicts occurs when a class cannot choose between different implementations for the same property. A variant concerns the case where several implementations must be combined. These two categories are quite different, and require different answers. The first kind is purely syntactical and a simple unambiguous denotation would provide a solution—it would apply to Java. The second kind of conflicts involves the program semantics—the solution cannot rely on some syntactic feature but the languages should offer to the programmer some help to manage them. An approach has been studied in depth, namely *linearizations* (Ducournau and Habib, 1987, 1991; Huchard et al., 1991; Ducournau et al., 1992, 1994, 1995; Barrett et al., 1996; Ernst, 1999; Forman and Danforth, 1999). These two inheritance

[1] The 'introduction' term is crucial here and will be more formally defined—a class *introduces* a method when it defines a method with a new signature which is not already defined in any of its superclasses.

levels were originally identified in (Ducournau et al., 1995) but, besides the fact that the paper was written in French, the lack of metamodel hinder us from achieving the analysis.

The approach proposed in this paper would apply to all object-oriented programming languages with multiple inheritance, multiple subtyping or even mixins. However, it turns out that *static typing* is required to take full advantage of the metamodel. Therefore, the paper focuses on static typing languages. Without loss of generality, this proposal is cross-cutting usual types theories and object calculi. Usual type theories, e.g. record types (Cardelli, 1988), are based on names and substituting reified properties to names does not change the considered type theory. However, multiple inheritance conflicts and method combination have special effects on types when redefinition is not type-invariant. So the paper presents an abstract analysis on the way static typing and multiple inheritance interact.

Overall, a major problematics runs across the paper, namely whether the *open world assumption* (OWA) holds or not. It concerns both the design level and the runtime systems. We consider that the object-oriented philosophy is the best expressed under the OWA—each class must be designed and implemented ignoring how it will be reused, especially whether it will be specialized in single or multiple inheritance. This is the essence of reusability. However, under the *closed world assumption* (CWA), when the class hierarchy is known as a whole, detecting and fixing conflicts is easy and implementation can be as efficient as with single inheritance. In contrast, under the OWA, e.g. in a dynamic loading setting, it is not possible to foresee the conflicts which might occur when defining future classes and this may affect the implementation efficiency.

The article is organized as follows. Section 2 presents the metamodel (i.e. an UML model) of classes and properties, and analyzes the meaning of class specialization and property inheritance. The metamodel is formalized in a simple set-theoretical way. Section 3 examines both kinds of conflicts and the different ways to solve them. Section 4 reviews the main results on the linearization approach and proposes some new extensions. Section 5 examines multiple inheritance from the standpoint of static typing, with non-invariant parameter and return types. Method combination, virtual types and parametric classes are also considered. In Section 6, the specifications of the most commonly used object-oriented languages are compared with the metamodel and the *mixin* alternative is examined. Multiple inheritance is also a question of implementation and object layout. The next section states the point and surveys the various solutions, depending on whether the OWA or CWA holds. Finally, the last section concludes the paper by presenting the current perspectives together with the known limitations of our proposal.

2. Class and Property Metamodel

In this section, we present first informally the key notions of specialization and inheritance, by coming back to the Aristotelian syllogistic. Then, we propose a formalization in a simple set-theoretic setting, with an UML meta-model.

2.1. *Specialization and inheritance*

The *de facto* standard object model is the *class-based* model, as opposed to other approaches like *actors* or *prototypes* (Masini et al., 1991). It consists mainly of:

- *classes*, organized in a *specialization* hierarchy;
- *objects*, that are created as instances of these classes by an instantiation process: the set of instances of a class constitutes its *extension*;
- each class is described by a set of properties, *attributes* for the state of its instances and *methods* for their behavior: the set of properties of a class constitutes its *intension*; moreover, a property defined in a class may be *overridden* (or *redefined*) in a subclass;
- applying a method to an object, called the *receiver*, follows the metaphor of *message sending* (also called *late binding*): the precise method is selected according to the class of the receiver.

This is the core of the model and it suffices to state the point of the specialization semantics.

Though novel in computer science, specialization has quite ancient roots in the Aristotelian tradition, in the well known syllogism: *Socrates is a human, humans are mortals, thus Socrates is a mortal.* Here *Socrates*

is an instance, *human* and *mortal* are classes. The interested reader will find in (Rayside and Campbell, 2000b; Rayside and Kontogiannis, 2001) an in-depth analysis of the relationships between object orientation and Aristotle syllogistic. Anyway, we use here 'Aristotelian' as an anchor to this bimillenial tradition which is at the basis of our modern common sense understanding of the real world.

According to the Aristotelian tradition, as revised with the computer science vocabulary, one can generalize this example by saying that *instances of a class are also instances of its superclasses*. More formally, $\prec$ is the specialization relationship ($B \prec A$ means that B is a subclass of A) and *Ext* is a function which maps classes to the sets of their instances, their *extensions*. Then:

$$B \prec A \Longrightarrow Ext(B) \subseteq Ext(A) \tag{1}$$

This is the essence of specialization and it has a logical consequence, namely inclusion of intensions, i.e. inheritance. When considering the properties of a class, one must remember that they are properties of instances of the class, factorized in the class. Let B be a subclass of A: instances of B being instances of A, have all the properties of instances of A. One says that subclasses inherit properties from superclasses. More formally, *Int* is a function which maps classes to the sets of their properties, their *intensions*. Then:

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B) \tag{2}$$

In the Aristotelian terminology, B is defined by *genus* (A) and *differentia*—$Int(B) \backslash Int(A)$.

The intuition of what are specialization and inheritance is well captured by the Aristotelian syllogistic. However, key points remain to examine. What is a property? What are the relationships between classes and properties? Moreover, property redefinition and late binding are not covered by Aristotelian syllogistic.

2.2. *The* Uml *model*

We present now a metamodel for classes and properties in object-oriented languages. Here, we only consider properties that (i) are described in a class but dedicated to its instances; (ii) depend on the *dynamic type* (aka *run-time type*) of the considered object, i.e. the class which has instantiated it. Hence, the considered properties are concerned by *late binding*. They could be qualified as *virtual* in C++ jargon, or even tagged by the `virtual` keyword in the method case [2]. Class properties, i.e. properties which only concern the class itself, not its instances, are excluded from the scope of the metamodel. We do not consider either *static* methods and variables, the fact that a class may be *abstract* (aka *deferred*). In Clos, however, we would also consider slots declared with `:allocation :class` because they are dedicated to instances though shared by all of them.

This metamodel is intended to be both intuitive and universal. It is likely very close to the intuition of most programmers, when they think of object-oriented concepts. It is universal in the sense that it is not dedicated to a specific language and it is very close to the specifications of most statically typed object-oriented languages, at least when they are used in a simple way. Hence, it can be considered as an implicit metamodel of most languages, even though none strictly conforms to it. However, this metamodel has never been explicitly described in any programming language nor even in Uml (OMG, 2004). In the following, we successively present abstract requirements on meta-modeling, an Uml model which provides an informal idea of the metamodel, then a more formal set-theoretical definition. The section ends by considering the resulting run-time behaviour. The analysis of multiple inheritance conflicts will be tackled in Section 3.

[2] 'Virtual' comes from Simula and has several usages—virtual functions (Simula, C++), virtual types and classes (Simula, Beta (Madsen and Møller-Pedersen, 1989; Madsen et al., 1993)) and virtual multiple inheritance (C++). It can be understood as 'redefinable', in the sense of 'redefinition' in the present paper—hence, submitted to late binding and depending on the dynamic type of some *receiver*. However, 'virtual' is also used in the sense of 'abstract', i.e. for a non-implemented method or a non-instantiable class—e.g. in (Pugh and Weddell, 1990; Igarashi and Pierce, 1999). Recently, Ocaml has substituted 'abstract' to 'virtual'— see for instance different versions of (Hickey, 2006). Though both meanings are related—'abstract' implies that something is 'virtual'—they are different enough to require different terms and we only consider the former usage.

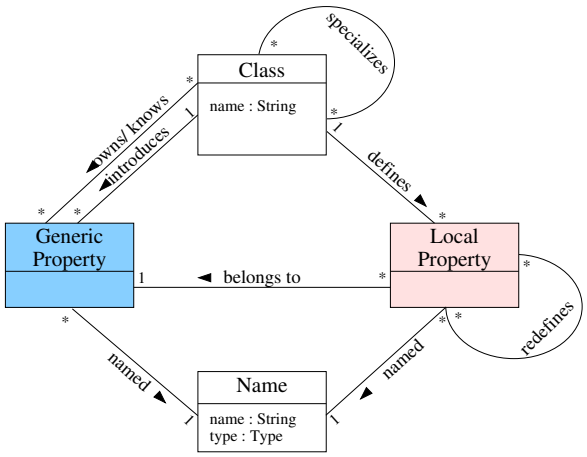


Fig. 1. Metamodel of Classes and Properties

2.2.1. *Object-oriented meta-modeling of object-oriented languages*

In an object-oriented setting, meta-modeling is a powerful tool which supports intuition and serves as an operational semantics when some meta-object protocol is added. A nice example is the analysis of classes, metaclasses and instantiation made in the ObjVlisp model [3] (Cointe, 1987). Meta-modeling some part of an object-oriented programming language amounts to defining an object model—i.e. entities like classes, associations, attributes, methods, etc.—for modeling the considered concepts of the language [4]. To this basic specification, we add the following requirement for unambiguity—actually, a metamodel should always be a specification of the meaning of names in a program.

Requirement 2.1 (Mapping from syntax to model) *In the modeled program, any occurrence of an identifier denoting a modeled entity must unambiguously map to a single instance of the metamodel. In other words, names represent a mapping from the syntax tree to the model.*

Accordingly, meta-modeling allows to get rid of names when considering the modeled entities. This will prove to be of great value, as most difficulties yielded by object-oriented programming lie in the interpretation of names. Besides the formalization of basic object-oriented concepts which follows in the next sections, the expected advantage of such a metamodel is that all programming tools could just consider reified entities instead of names. Actually, these reified entities represent the 'reality'—the *ontology*—of object-oriented programs and names should remain at the human-computer interface.

Conversely, following Occam's Razor, meta-modeling should aim at minimality. As a counter-example, the Clos reflective kernel includes, but is far larger than, the ObjVlisp model. It may be necessary for fully implementing Clos, but it is useless for conceptually modeling classes, superclasses and instantiation. Models must remain partial—this is of course a truism, since a complete model is the real world itself.

2.2.2. *Classes and properties*

The metamodel consists of three main kinds of entities: *classes*, *generic properties* [5] and *local properties*—Fig. 1. The former is natural, but the two latter are original key features of the model. It follows from Requirement 2.1 that late binding (aka message sending) implies the definition of exactly two categories

[3] We take ObjVlisp as an example because it focuses on its object—namely, classes and metaclasses, in their purest form. Actually, the reflective kernel of ObjVlisp is also present in Clos and in other works on reflection, e.g. (Forman and Danforth, 1999). In contrast, it is not compatible with the Smalltalk reflective kernel (Goldberg and Robson, 1983).

[4] Here, we adopt a very restrictive meaning of the term 'metamodel', which is much more general—we only consider *reflective object-oriented* metamodels, i.e. a model of the object-oriented model in itself.

[5] The term 'generic property' has been coined on the model of Clos *generic functions*, which is closely related, apart from method dispatch. Zibin and Gil (2003) use the term 'method family' or 'implementation family' in a close though informal sense. 'Genus' is of course the Latin word for 'family'. In (Ducournau et al., 2007), we used 'global' instead of 'generic', for a simple technical reason—the metamodel were also applied to the class-module level, and 'generic class' would have been confused with *parametrized classes* (aka *generics*). In our minds, there is no difference at all between 'generic' and 'global'.

of properties. *Local properties* correspond to the attributes and methods as they are defined in a *class*, independently of other possible definitions of the 'same property' in superclasses or subclasses. *Generic properties* are intended to model this idea of the 'same property' in different related classes. They correspond to messages that the instances of a class can answer—in the case of methods, the answer is the invocation of the corresponding local property of the dynamic type of the receiver. Each local property *belongs to* a single generic property and is *defined* in a single class. Finally, a last kind of entities represent names which are crucial and can be less simple than usual symbols.

Generic and local properties must in turn be specialized into several specific kinds, according to the values which are associated with the properties—data for attributes, functions for methods or even types for *virtual types*. Attributes and methods are present in all languages but the main complication involves the methods—indeed, attributes are usually quite simpler, though languages such as CLOS or EIFFEL accept full attribute redefinition. Moreover, a proper distinction between attributes and methods may be not straightforward since it is sound to accept, as in EIFFEL, that a method without parameters can be redefined by an attribute. However, there is no need to detail this here. Besides attributes and methods, a third kind of properties must also be considered, namely *virtual types*. Whereas attributes and methods are respectively associated with data and functions, virtual types involve associating a property with a type which depends on the dynamic type of the receiver. Virtual types (Torgersen, 1998; Thorup and Torgersen, 1999; Igarashi and Pierce, 1999) represent a combined genericity and covariance mechanism first introduced in BETA (Madsen and Møller-Pedersen, 1989; Madsen et al., 1993) and somewhat similar to EIFFEL *anchored types* (Meyer, 1997). The fact that functions and types are part of the metamodel does not imply that they must be first-class objects in the considered programming languages. Anyway, in the following, 'property' stands for all three kinds and a partial but more intuitive translation of our terminology is possible—generic (resp. local) properties stand for methods (resp. method implementations). Furthermore, in this section, the specific kind of property does not matter and the case of virtual types will be examined in Section 5.

A class definition is a triplet consisting of a class name, the name of its superclasses, presumably already defined, and a set of local property definitions. The specialization relation supports the inheritance mechanism—i.e. classes inherit the properties of their superclasses. When translated in terms of the meta-model, this yields two-level inheritance. First of all, the new class *inherits* from its superclasses all their generic properties—this is *generic property inheritance*. The class *knows* all the generic properties *known* by its superclasses. Then each local property definition is processed. If its name is the same as that of an inherited generic property, the new local property is attached to the generic property. If there is no such inherited generic property, a new generic property with the same name is *introduced* in the class.

Local property inheritance takes place at run-time—though most implementations statically precompile it. A call site `x.foo(args)` represents the invocation of the *generic property* named `foo` of the *static type* (say A) of the receiver `x`. The static typing requirement appears here—in a dynamic typing framework, for instance in SMALLTALK, there is no way to distinguish different generic properties with the same name while satisfying Requirement 2.1. At run-time, this call site is interpreted as the invocation of the **single** *local property* corresponding to both the *generic property*, and the *dynamic type* of the value bound to `x`, i.e. the class which has instantiated it. Therefore, when no such local property is defined in the considered class, a local property of the same generic property must be inherited from superclasses. Static typing ensures that such a local property exists.

Static overloading slightly complicates the point. In many statically-typed languages, such as C++, JAVA or C#, a class may have several properties with the same name and different parameter types and numbers. This implies distinguishing the property names in the language and in the meta-model. In the language, the names can be overloaded but, in the meta-model, they are made locally unambiguous by considering the tuple formed by the name and the parameter types—this is known as *mangling* in the compiler jargon and the reason for explicitly reifying names. Hence, names are signatures and two overloaded properties correspond to different generic properties. If the name '`foo`' is overloaded in A, i.e. if A knows several generic properties named `foo`, which differ by their parameter types or number, a **single** one, i.e. the most specific according to formal and actual parameter types, must be first selected *at compile-time*. A conflict may occur when the most specific property is not unique, e.g. when there is multiple inheritance between parameter types, or with multiple contravariant parameters. Such conflicts can be easily solved by simply

```
1  class A {
2     public void foo()
      {...}
   }

3  class B extends A {
4     public void foo()
      {...}
5     public void bar()
      {...}
   }

   ...

   {...
1   A x
3   B y
6   x.foo()
7   y.bar()
    ...}
```

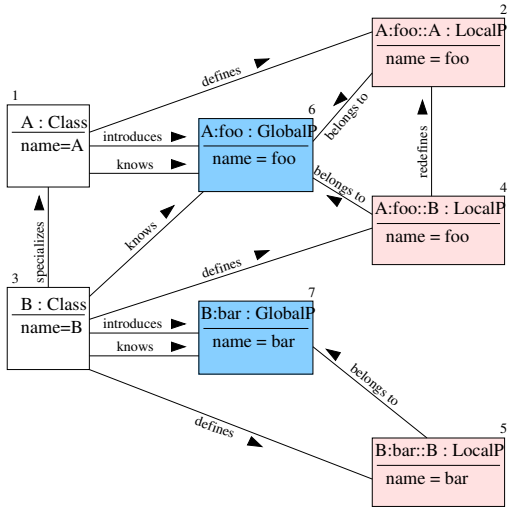


Fig. 2. A simple JAVA example and the corresponding instance diagram. For the sake of simplicity, names are no longer reified.

making actual parameter types more precise, i.e. paradoxically more general, at the call site. Anyway, *at run-time*, the call site remains interpreted as the invocation of the **single** *local property* corresponding to both this **single** statically selected *generic property*, and the *dynamic type* of the value bound to `x`.

2.2.3. *Example*

The JAVA example in Figure 2 defines seven entities of our metamodel—two classes, `A` and `B`; three local properties, the method `foo` defined in `A` and the methods `foo` and `bar` defined in `B`; two generic properties, respectively introduced as `foo` in `A` and as `bar` in `B`. The corresponding numbered instance diagram specifies the unambiguous mapping between the code and the instances of the metamodel (Req. 2.1).

The instantiation of the metamodel proceeds as follows, as the code is read:

- `A` class is first created; it does not inherit any explicit generic property—in practice, it would inherit all generic properties introduced in the hierarchy root, `Object`;
- a method named `foo` is defined in `A`: the corresponding local property (1) is created and, since `A` does not know any generic property with this name, a generic property (4) `foo` is introduced;
- `B` class is then created as a specialization of `A`; it then inherits all explicit generic properties from `A`, especially the generic property `foo`;
- a method named `foo` is defined in `B`: the corresponding local property (2) is created and attached to the generic property (4) `foo` inherited from `A`—this is a redefinition;
- a method named `bar` is defined: the corresponding local property (3) is created and, since `B` does not know any generic property named `bar`, then a generic property (5) `bar` is introduced;
- in the following code sequence, `foo` (resp. `bar`) is understood as the single generic property named `foo` (resp. `bar`) which is known by the static type `A` (resp. `B`) of the receiver `x` (resp. `y`); changing the static type of `x` from `A` to `B` would not change the mapping but doing the converse for `y` would yield a static type error—`A` class does not know any generic property named `bar`.
- finally, at run-time, the invocation of `foo` will call the local property defined in `A` or in `B`, according to the actual dynamic type of the value of `x`—this is late binding, as usual.

Figure 2 clearly suggests how a development tool like Eclipse could allow the programmer to easily navigate between the source code and the model instances.

2.3. *Formal Definitions*

This section first defines a model in a static way—i.e. its components and their relationships—then describes the protocols (i) for instantiating it, and (ii) for late binding.

Notations

Let E, F and G be sets. 2^E denotes the power set of E, $|E|$ is the cardinality of E, and $E \uplus F$ is the union of the disjoint sets E and F. Given a function $foo : E \to F$, the function $foo^{-1} : F \to 2^E$ maps $x \in F$ to the set $\{y \mid foo(x) = y\}$. Function notations are extended to powersets and Cartesian products in the usual way: $\forall G \subseteq E$, $foo(G) = \{foo(x) \mid x \in G\}$ and $\forall R \subseteq E \times E$, $foo(R) = \{(foo(x), foo(y)) \mid (x, y) \in R\}$. Finally, $(E, \leq)$ denotes the graph of a binary relation $\leq$ on a set E. It is a *poset* (partially ordered set) iff $\leq$ is reflexive, transitive and antisymmetric. Then, $\max_{\leq}$ (resp. $\min_{\leq}$) denotes the *maximal* (resp. *minimal*) elements of a subset of E.

Definition 2.1 (Class hierarchy) *A* model of a hierarchy*, i.e. an instance of the metamodel, is a tuple* $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, name_{\mathcal{H}}, gen_{\mathcal{H}}, intro_{\mathcal{H}}, def_{\mathcal{H}} \rangle$*, where:*

- $X^{\mathcal{H}}$ *is the set of* classes*;*
- $\prec^{\mathcal{H}}$ *is the* class specialization *relationship, which is transitive, antisymmetric and anti-reflexive;* $\preceq^{\mathcal{H}}$ *(resp.* $\prec^{\mathcal{H}}_d$*) denotes the reflexive closure (resp. transitive reduction) of* $\prec^{\mathcal{H}}$ *and* $(X^{\mathcal{H}}, \preceq^{\mathcal{H}})$ *is a poset;*
- $G^{\mathcal{H}}$ *and* $L^{\mathcal{H}}$ *are the disjoint sets of* generic *and* local properties*;*
- $N^{\mathcal{H}}$ *is the set of* identifiers *(names) of* classes *and* properties*;*
- $name_{\mathcal{H}} : X^{\mathcal{H}} \uplus G^{\mathcal{H}} \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ *is the naming function of classes and properties; its restriction to* $X^{\mathcal{H}}$ *is injective;*
- $gen_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ *associates with each local property a generic property;*
- $intro_{\mathcal{H}} : G^{\mathcal{H}} \to X^{\mathcal{H}}$ *associates with a generic property the class introducing it;*
- $def_{\mathcal{H}} : L^{\mathcal{H}} \to X^{\mathcal{H}}$ *associates with a local property the class where it is defined.*

The notations are supplemented by the following set of equations and definitions (3–11).

The sets $X^{\mathcal{H}}$, $G^{\mathcal{H}}$ and $L^{\mathcal{H}}$ correspond to the three classes in the metamodel, whereas the total functions $gen_{\mathcal{H}}$, $intro_{\mathcal{H}}$ and $def_{\mathcal{H}}$ correspond to the three functional associations—all six elements form the metamodel upper triangle. The 'specializes' association is represented by $\prec^{\mathcal{H}}$ and all other associations, such as 'knows' and 'redefines', are not primitive. On the other hand, $N^{\mathcal{H}}$ and $name_{\mathcal{H}}$ represent relationships between the metamodel and names which are used in the program text. So far, the formalization is a straightforward translation of the UML diagram in Figure 1. The definition of a *legal model* is achieved by a set of constraints which ensure that: (i) the triangular diagrams commute at the instance level, (ii) names in the program text are unambiguous or can be disambiguated.

The metamodel is generic, as all its components are parametrized by $\mathcal{H}$. However, in the rest of the paper, parameter $\mathcal{H}$ will remain implicit for the sake of readability. The parameter must be explicit when several hierarchies are considered, e.g. as in (Ducournau et al., 2007). All proofs are trivial and left to the reader.

2.3.1. *Generic Properties*

Given a class $c \in X$, G_c denotes the set of generic properties *known* by c. Generic properties are either *inherited* from a superclass of c, or *introduced* by c. Let $G_{\uparrow c}$ and G_{+c} be the two corresponding subsets. Hence, all G_{+c} are disjoint and

$$G_{+c} \triangleq intro^{-1}(c) \ , \tag{3}$$

$$G_{\uparrow c} \triangleq \bigcup_{c \prec_{\mathrm{d}} c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'} \ , \tag{4}$$

$$G_c \triangleq G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \ , \tag{5}$$

$$G = \bigcup_{c \in C} G_c = \biguplus_{c \in C} G_{+c} \ . \tag{6}$$

The definitions and equations (3-5) formally define *generic property inheritance.*

Names of newly introduced generic properties are constrained:

Constraint 2.1 (Locally unambiguous names) *When a generic property is introduced, its name must be unambiguous, hence for all $c \in X$, (i) the restriction of name on G_{+c} is injective and (ii) inherited and introduced properties cannot have the same name:*

$$name(G_{+c}) \cap name(G_{\uparrow c}) = \emptyset \ .$$

This constraint is actually implied by the constraints concerning local properties (Section 2.3.2). It follows from it that we do not consider here the questionable possibility of introducing a new generic property with the same name as an inherited one, as with the `reintroduce` keyword in Turbo Pascal. Moreover, the function $gid : G \to X \times N$ that maps a generic property $g \in G$ to the pair

$$gid(g) \triangleq (intro(g), name(g)) \tag{7}$$

is injective. Besides this constraint, all $\langle X, \prec, G, name, intro\rangle$ tuples that satisfy Definition 2.1 are legal.

There is a generic property conflict when a class knows two distinct generic properties with the same name. Constraint 2.1 implies that such a conflict is always caused by multiple inheritance:

Definition 2.2 (Generic property conflict) *Given a class $c \in X$ and two distinct generic properties $g_1, g_2 \in G_{\uparrow c}$, a* generic property conflict *occurs between g_1 and g_2 when*

$$name(g_1) = name(g_2) \wedge intro(g_1) \neq intro(g_2) \ .$$

Moreover, this implies that there are classes $c', c_1, c_2 \in X$, such that:

$$(c \preceq c' \prec_{\mathrm{d}} c_1, c_2) \wedge (g_1 \in G_{c_1} \backslash G_{c_2}) \wedge (g_2 \in G_{c_2} \backslash G_{c_1}) \ .$$

When there is no generic property conflict, for instance in a legal model in single inheritance, the restriction of the function $name : G \to N$ to $G_{\uparrow c}$ is injective. Moreover, in the same condition, Constraint 2.1 implies that the restriction of *name* to G_c is also injective. Therefore, in the context of a class c, the identifier of a generic property is unambiguous. Of course, *name* is not constrained to be injective throughout its domain G—hence it must be disambiguated by the context, i.e. the static type of the receiver. Generic property conflicts will be examined in Section 3.1.

2.3.2. *Local Properties*

Given a class c, L_c denotes the set of local properties *defined* in c and, conversely, the function $def : L \to C$ associates with each local property the class where it is defined:

$$L = \biguplus_{c \in C} L_c \qquad \text{with} \qquad L_c \triangleq def^{-1}(c) \ . \tag{8}$$

The correspondence between local and generic properties must be constrained in order to close the metamodel triangles, i.e. to make the diagrams commute. First, the correspondence is based on property names:

Constraint 2.2 (Name triangle) *The function $gen : L \to G$ associates with each local property a generic property, such that both have the same name:*

$$\forall l \in L, name(gen(l)) = name(l) \ .$$

Moreover, it does not make sense to define more than one local property for some generic property in the same class, hence:

Constraint 2.3 (Single local definition) *For all $c \in X$, the restriction of gen to L_c is* injective. *Equivalently, for all $g \in G$, the restriction of def to $gen^{-1}(g)$ is injective.*

Therefore, if there is no generic property conflict, the restriction of *name* to L_c is also injective. Thus determining the generic property associated with a local one is unambiguous when the name of the generic property is unambiguous.

$$\forall l \in L_c, \forall g \in G_c, name(l) = name(g) \Longrightarrow gen(l) = g \ . \tag{9}$$

Of course, *gen* and *name* are not injective over their whole domain L—hence local property names must be disambiguated by the context, i.e. the enclosing class, at compile-time, and local properties must be selected by the late binding mechanism at run-time (see below).

A last constraint closes the upper triangle and achieves the definition of a *legal model*:

Constraint 2.4 (Class triangle) *The generic property associated with a local one must be known by the defining class and all generic properties must have been introduced by a local property definition:*

$$\forall c \in X, G_{+c} \subseteq gen(L_c) \subseteq G_c \ .$$

If a property is considered as *abstract* (aka *deferred*) in its introduction class—i.e. if it has no default implementation—an abstract local property must still be provided.

Definition 2.3 (Legal model) *A* legal model *of a* class hierarchy *is a model which satisfies all Constraints 2.1 to 2.4.*

It follows from Constraint 2.3 that, in a legal model, the function $lid : L \to G \times X$ that maps a local property $l \in L$ to the pair

$$lid(l) \triangleq (gen(l), def(l)) \tag{10}$$

is injective.

Finally, one can formally define *property redefinition*. A local property belongs to an inherited or newly introduced generic property. Let $L_{\uparrow c}$ and L_{+c} be the corresponding sets.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{with} \begin{cases} L_{\uparrow c} & \triangleq L_c \cap gen^{-1}(G_{\uparrow c}) \ , \\ L_{+c} & \triangleq L_c \cap gen^{-1}(G_{+c}) \ . \end{cases} \tag{11}$$

Moreover, *gen* is a one-to-one correspondence between L_{+c} and G_{+c}.

Definition 2.4 (Property redefinition) Property redefinition *(aka* overriding*) is defined as the relationship $\ll^{\mathcal{H}}$ (or $\ll$ for short) between a local property in $L_{\uparrow c}$ and the corresponding local properties in the superclasses of c:*

$$l \ll l' \overset{\Delta}{\Longleftrightarrow} gen(l) = gen(l') \wedge def(l) \prec def(l') \ .$$

This is a strict partial order and $\ll_d$ denotes its transitive reduction.

2.3.3. *Class Definition and Model Construction*

The model is built by successive class definitions.

Definition 2.5 (Class definition) *A* class definition *is a triplet* ⟨`classname`, `supernames`, `localdef`⟩, *where* `classname` *is the name of the newly defined class,* `supernames` *is the set of names of its direct superclasses—they are presumed to be already defined—and* `localdef` *is a set of local property definitions.*
A local property definition involves a property name—in the general meaning, i.e. a signature if static overloading is considered—and other data, e.g. code, which are not needed here.

Let $\mathcal{H}$ be a legal class hierarchy, then a class definition in $\mathcal{H}$ will produce another hierarchy $\mathcal{H}'$. The operational semantics of the metamodel is given by the *meta-object protocol* which determines how this class definition is processed. We informally sketch this 4-step protocol as follows.

(i) *Hierarchy update:* a *new* class c with name `classname` is added to X—i.e. $X' = X \uplus \{c\}$. For each name $n \in$ `supernames`, a pair $(c, name^{-1}(n))$ is added to $\prec_d$. Of course, the names of all considered classes are checked for correction—existence and uniqueness. Moreover, `supernames` is a set—this means that it does not make sense to inherit more than once from a given class—and it should also be checked against transitivity edges, that should not be added to $\prec_{\mathrm{d}}$ [6] .

(ii) *Generic property inheritance:* $G'_{\uparrow c}$ is computed (4) and generic property conflicts are checked (Def. 2.2).

(iii) *Local definitions:* for each definition in `localdef`, a *new* local property is created, with its corresponding name—this yields L'_c. $L'_{\uparrow c}$ is determined by (11). Then, G'_{+c} is constituted as the set of *new* generic

[6] In both cases, this is contrary to C++ and Eiffel behavior. Anyway, transitive edges would have no effect on the constructed model. See also Section 4.2.

properties corresponding to each local property in $L'_{+c} = L'_c \backslash L'_{\uparrow c}$. L'_c and G'_{+c} are then respectively added to L and G—i.e. $L' = L \uplus L'_c$ and $G' = G \uplus G'_{+c}$. Here again, the names of all local properties are checked for correction—existence and uniqueness. Ambiguities resulting from generic property conflicts are discussed in Section 3.1.

(iv) *Local property inheritance:* finally, the protocol proceeds to local property inheritance and checks conflicts for all inherited and not redefined properties, i.e. $G'_{\uparrow c} \backslash gen(L'_{\uparrow c})$. Conflicts are discussed in Section 3.2.

In the protocol, each occurrence of 'new' denotes the instantiation of a class in the metamodel.

The metamodel is complete, in the sense that all components in Definition 2.1, together with Constraints 2.1 to 2.4 and equations (3–11), are sufficient to characterize a legal model as long as there is no generic property conflict. All such legal models could be generated by Definition 2.5—given a legal class hierarchy $\mathcal{H}$, for all c in X ordered by $\preceq$, $\langle name(c), \{name(c') \mid c \prec_d c'\}, name(L_c)\rangle$ forms a legal class definition.

2.4. *Local property inheritance and method invocation*

So far, we have presented the static model, which considers classes and properties at compile-time. These classes behave as usual at run-time, i.e. they have instances which receive and send messages according to the Smalltalk metaphor. Run-time objects and their construction are not explicit in the model—for instance, we have not merged the previous metamodel within the ObjVlisp kernel. This is actually not required since message sending can be modelled at the class level—it only depends on the *receiver's dynamic type*, which is some class. The precise receiver does not matter and all direct instances of a given class are equivalent—this explains why method invocation can be compiled and efficiently implemented (see Section 7). Constructor methods are no more explicit in the metamodel because there is no room for them—the construction itself is a hidden black box and the so-called constructors are ordinary initialization methods which do not require any specific treatment [7].

Local property inheritance is a matter of selection of a single local property in a given generic property, according to the dynamic type of the receiver. Though it non-trivially applies only to methods in most languages, it can also be used for attributes as in Clos or Eiffel. We only detail the case concerning methods. Method invocation usually involves two distinct mechanisms, namely late binding (aka message sending) and call to `super`. It implies some auxiliary functions—$loc, spec : G \times X \to 2^L$, $sel : G \times X \to L$, $cs : G \times X \to 2^X$ and $supl : L \to 2^L$—defined as follows. All functions *loc*, *spec*, *sel* and *cs* are partial and only defined on $g \in G$ and $c \in X$ when $g \in G_c$.

2.4.1. *Late binding*

Given a class $c \in C$—the dynamic type of the receiver—and a generic property $g \in G_c$, *late binding* involves the selection of a local property of g defined in the superclasses of c (including c), hence in the set

$$loc(g, c) \triangleq \{l \in gen^{-1}(g) \mid c \preceq def(l)\} \ . \tag{12}$$

Thus, the selection function *sel* must return a local property such that $sel(g, c) \in loc(g, c)$. If c has a local definition for g—i.e. if $L_c \cap loc(g, c) = \{l\}$—then $sel(g, c) = l$. On the contrary, c must inherit one from its superclasses—this constitutes the second level of inheritance, namely *local property inheritance*. In single inheritance, *sel* returns the *most specific* property, i.e. the single element in

$$spec(g, c) \triangleq \min_{\ll}(loc(g, c)) \ , \tag{13}$$

[7] 'Constructor' is a misleading term. As a local property, it just denotes an *initializer*, which corresponds to Clos `initialize-instance`. At a constructor call site, i.e. as a generic property, the construction role is ensured by some hidden mechanism—similar to Clos `make-instance`—which makes the instance before calling the initializer. Anyway, the programmer must usually declare that a given method can be used as a constructor, i.e. as a *primary initializer* directly called at an instantiation site. Normally, any method can be used as a secondary initializer, i.e. when indirectly called by a primary initializer, though an odd specification makes it false in C++ since, in this language, late binding is impossible within a constructor.

but its uniqueness is only ensured in single inheritance. In multiple inheritance, a *local property conflict* may occur:

Definition 2.6 (Local property conflict) *Given a class $c \in X$ and a generic property $g \in G_{\uparrow c}$, a* local property conflict *occurs when $|spec(g,c)| > 1$. The* conflict set *is defined as the set*

$$cs(g,c) \triangleq def^{-1}(spec(g,c)) = \min_{\preceq}(def^{-1}(loc(g,c))) \tag{14}$$

which contains all superclasses of c which define a local property for g and are minimal according to $\preceq$.

We shall examine, in Section 3.2, the case of multiple inheritance, local property conflicts and the reasons for this definition.

2.4.2. *Call to* `super`

Call to `super` allows a local property l to call another one, say l', which is redefined by l, i.e. $l \ll l'$. This can be ensured by another selection function which selects an element in the set

$$supl(l) \triangleq \{l'' \mid l \ll l''\} \ . \tag{15}$$

Note that we keep the term '`super`' used in SMALLTALK and JAVA as it is the most popular, but we take it with a slightly different meaning. It is here closer to EIFFEL `Precursor`. The point is that l and l' belong to the same generic property, i.e. $gen(l) = gen(l')$, while SMALLTALK and JAVA accept `super.foo` in method `bar`[8]. Usually, like late binding, call to `super` involves selection of the most specific property, i.e. the single element in $\min_{\ll}(supl(l))$, which is the single local property l' which satisfies $l \ll_d l'$. However, the uniqueness of l' is only ensured in single inheritance, where it is determined by the uniqueness of c' such that $def(l) \prec_d c'$—then $l' = sel(gen(l), c')$.

The bottom-up call to `super` is not the only way of combining methods. A top-down mechanism has also been proposed in LISP-based object-oriented languages (FLAVORS, CLOS) under the name of `:around` methods (aka *wrappers*) and in BETA under the name of `inner`. Goldberg et al. (2004) propose their integration in a single language. As these mechanisms are exactly symmetric to `super`, they present similar problems in case of multiple inheritance. We shall examine them in Section 3.2.

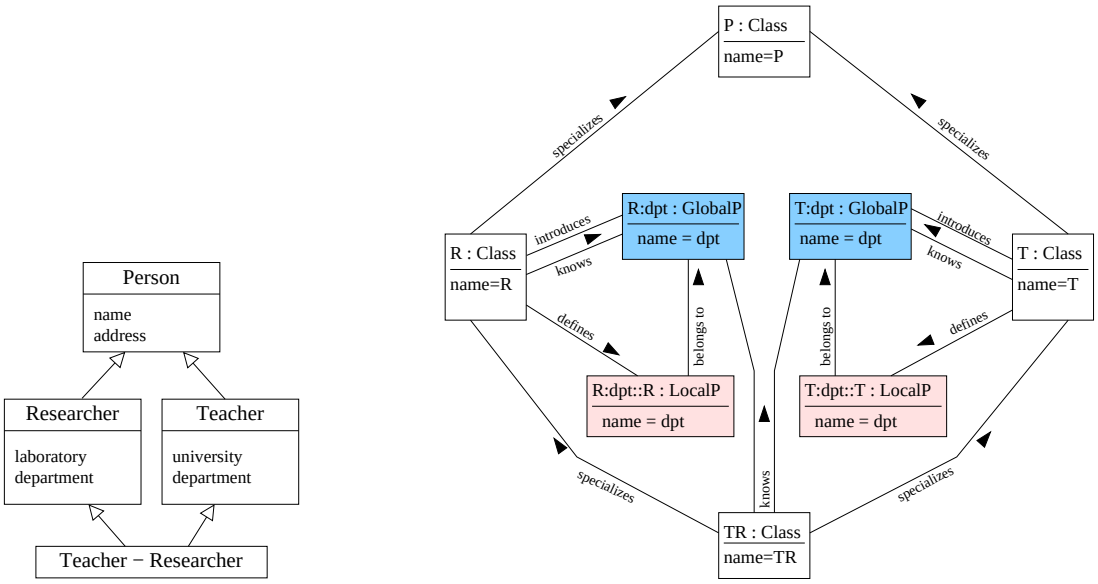
3. Multiple Inheritance Conflicts

In multiple inheritance, conflicts are the main difficulty. They are usually expressed in terms of name ambiguities. The metamodel yields a straightforward analysis in terms of reified entities and distinguishes between two kinds of conflicts which require totally different answers. The following analysis is mostly the same as in (Ducournau et al., 1995), just enhanced with the metamodel.

3.1. *Generic Property Conflict*

A *generic property conflict*—in (Ducournau et al., 1995), it was called 'name conflict'—occurs when a class specializes two classes having distinct but homonymic generic properties (Def. 2.2). Figure 3 shows two generic properties named `department`. The first one specifies a department in a research laboratory. The other one specifies a teaching department in a university. It is then expected that the common subclass `Teacher-Researcher` inherits all the different generic properties of its superclasses. However, the name `department` is ambiguous in the subclass context. Anyway, this situation is simply a naming problem. It must be solved and a systematic renaming in the whole program would solve it. Different answers are possible, which do not depend on the specific kind of properties, i.e. attributes, methods or types, since the conflict only involves names.

[8] SMALLTALK and JAVA consider that `super` denotes an object—i.e. `self` 'in the superclass'. In contrast, `Precursor` has the EIFFEL syntax for a type. Like CLOS, we consider that `super` and `call-next-method` are methods, i.e. the current generic property 'in the superclass'.



-a Class diagram -b Instance diagram

Fig. 3. Generic Property Conflict—the class diagram (a) depicts a conflict between the two properties named `department` introduced in two unrelated classes and the instance diagram (b) shows the corresponding metamodel instantiation. All entities are tagged by unambiguous fully qualified names and names are abbreviated.

Nothing, i.e. error. The language does not specify any answer to such a conflict but it signals an ambiguity. This forces the programmer to rename at least one of the two properties, but this must be done throughout the program and can be error-prone or even impossible (unavailable source code).

Fully qualified names. This simply involves an alternative unambiguous fully qualified syntax, which juxtaposes the property name with the name of a class in which the property name is not ambiguous, for instance the class that introduces the generic property. In the example, `Teacher:department` would denote the generic property known as `department` in the class `Teacher`. Such a naming would be unambiguous since, in a legal model, *gid* is injective (7). A similar solution is used for attributes in C++ with the operator :: (Stroustrup, 2000) [9] . Note that, with fully qualified names, a generic property conflict requires a solution only when the programmer wants to use the ambiguous name, in a context where it is ambiguous. Hence, this is a modular and lazy solution.

Local Renaming. Local renaming changes the designation of a property, both generic and local, in a class and its future subclasses. In the `Teacher-Researcher` class of the example, one can rename `department` inherited from `Teacher` as `teach-dept` and `department` inherited from `Researcher` as `res-dept`. Thus, `department` denotes, in `Researcher`, the same generic property as `res-dept` in `Teacher-Researcher`; conversely, in the class `Teacher-Researcher`, `res-dept` and `teach-dept` denote two distinct generic properties. This solution is used in Eiffel (Meyer, 1997). Renaming here is required even when the programmer does not use the name in a context where it is ambiguous—i.e. when redefining one of the conflicting properties or calling it on a receiver typed by the considered class. Moreover, as class hierarchies are not constrained to form lattices, i.e. there is not a single lower bound, the same conflicting properties can occur in different subclasses, with possibly different renamings.

Unification, i.e. silence. Dynamic languages, like Clos, and C++ for functions (not for attributes), consider that if two generic properties have the same name then they are not distinct. Java has the same behaviour when a class implements two interfaces which introduce a generic property with the same

[9] In C++, attributes cannot be redefined—this would be static overloading, hence a new generic property. Therefore, a single local attribute corresponds to each generic attribute and :: denotes either the local or generic one. For methods, :: corresponds to a static call—therefore, it denotes a local property.

name and signature [10] . Hence, the generic property conflict is not recognized and the multiple inheritance ambiguities are deferred to local property inheritance. So this solution is very close to the first one—it does not allow the programmer to express his/her intention of distinct generic properties, unless there is global renaming. In Figure 3, the two departments represent distinct concepts. If the programmer's intention was a single concept, then he/she should have defined a common superclass introducing a single generic property for this concept. However, silence adds an extra flaw—i.e. the programmer may remain unaware of the problem or might misunderstand it.

Naming convention. When languages deal with generic property conflicts by signalling an error or unifying conflicting properties, the situation may have no solution. A global renaming would be the only way out, but it might be impossible under the OWA, if the source code of the conflicting classes is not available, or if the conflicting classes are contractually used by other people. Therefore, it is necessary to anticipate conflicts by enacting and following a naming convention—e.g. property names would be prefixed by all or part of the introducing class name. This is however a subjective solution that relies on the programmer's discipline instead of being ensured by the language specifications. In dynamic typing, this is anyway the only solution since the metamodel cannot distinguish homonymic properties. In static typing, this convention would be redundant with static types and would make the code clumsy.

As a provisional conclusion, generic property conflicts represent a shallow problem. They could easily be solved in any programming language at the expense of a cosmetic modification—namely, *qualifying* or *renaming*—and they should not be an obstacle to the use of multiple inheritance. Both solutions need a slight adaptation of the metamodel. This is straightforward in the case of full qualification—all names in the class definition—more generally in the program text—may be simple or fully qualified. An analogous syntax is available in all languages where name-spaces are explicit, e.g. packages in Java or Common Lisp, but it does not apply at the right level. This is less simple for local renaming—the function $name_{\mathcal{H}}$ is no longer global and must take two parameters, the property and the class. Moreover, renaming clauses must be integrated in Definition 2.5. The best solution might be combining both qualification and renaming, the latter being restricted to lexical scope. Full qualification is the default solution but it might be clumsy if the fully qualified name is to be overused in a compilation unit. So, the programmer might prefer to rename it, but the renaming scope is the compilation unit—renaming is no longer inherited.

3.2. *Local Property Conflict*

3.2.1. *Conflict definition*

A *local property conflict*—in (Ducournau et al., 1995), it was called 'value conflict'—occurs when a class inherits two local properties from the same generic property, with none of them more specific than the other according to $\ll$ (Def. 2.6). Figure 4-a illustrates this situation with two classes, `Rectangle` and `Rhombus`, both redefining the method `area` which was introduced into a common superclass, `Quadrilateral`. In the common subclass `Square`, none is most specific—which one must be selected? Figure 4-c depicts the model of the example, restricted to the concerned property. To be compatible with the syntax for generic properties, we adopt here a syntax similar to that of C++ but slightly different—namely `area::Rhombus` instead of `Rhombus::area` denotes the local property `area` defined in the class `Rhombus` (Fig. 4). It is always possible in a legal model since the *lid* function (10) is injective. This is the same meaning as the C++ operator :: (see Note 9), apart from operand inversion. If the generic property name is ambiguous, full qualification can lead to `Quadrilateral:area::Rhombus`. Note however that this syntax is metalanguage and is not part of the considered programming language.

Let g be the `area` generic property introduced in `Quadrilateral`—`Q:area` for short—and c be the `Square` class, abbreviated in `S` in the diagram. Then, according to formulas (12-14),

[10] Java provides a fully qualified syntax for classes, not for properties.

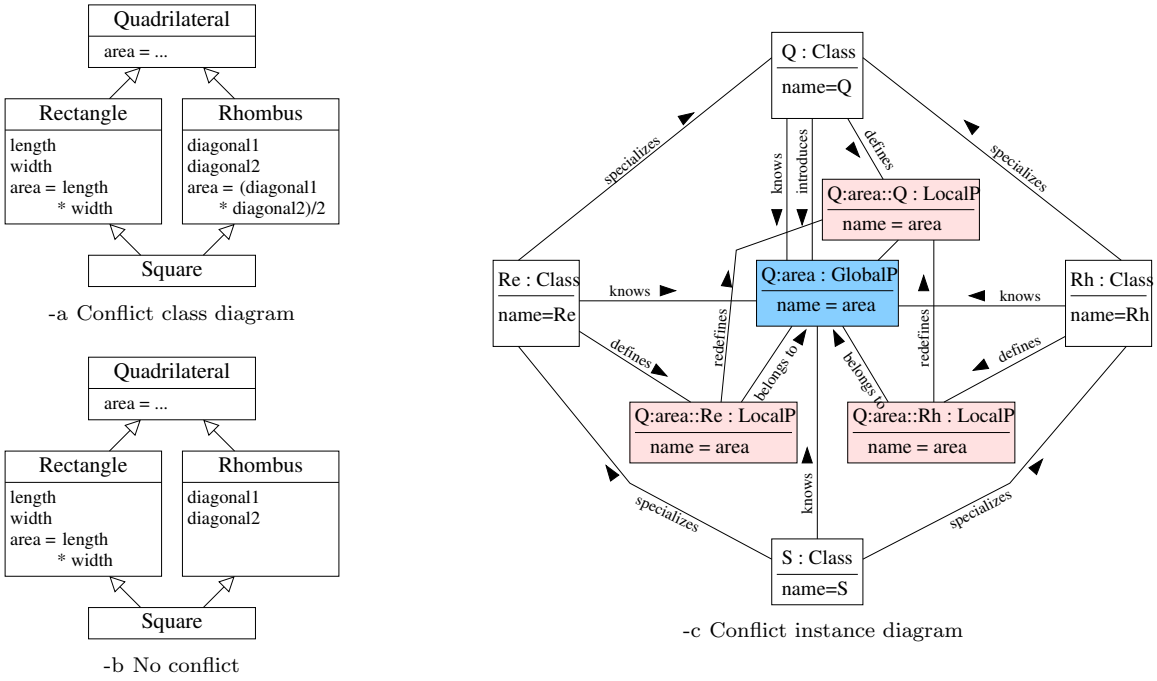


Fig. 4. Local Property Conflict—the class diagram (a) depicts a conflict between the two local properties `area` redefining the same property in two unrelated classes and the instance diagram (c) shows the corresponding metamodel instantiation. In contrast, there is no conflict in class diagram (b).

$$\begin{aligned} loc(g,c) &= \{\texttt{Q:area::Q}, \texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ spec(g,c) &= \{\texttt{Q:area::Rh}, \texttt{Q:area::Re}\},\\ cs(g,c) &= \{\texttt{Rh}, \texttt{Re}\}. \end{aligned}$$

The conflict vanishes if one removes the definition of one of the two conflicting local properties, say `Q:area::Rh`, though `Rhombus` still inherits a local property from `Quadrilateral` (Fig. 4-b). However, some languages, e.g. EIFFEL, consider that there is still a conflict in `Square`, between the property defined in `Rectangle` and the property inherited by `Rhombus`, unless the latter is abstract. Therefore, it is important to understand why we choose this conflict definition. Apart from redefinition, specialization is inherently *monotonic*—i.e. in the definition of A, what is true for an instance of A is also true for an instance of any subclass of A. This goes back to Aristotelian syllogistic—see Section 2.1. On the contrary, property redefinition entails *non-monotonicity*, in the sense of *non-monotonic* (aka *defeasible*) inheritance theories (Touretzky, 1986; Horty, 1994). A local property is a *default value* for the instances of the class which defines it. For all instances of a subclass A', the redefining property *overrides* (or *masks*) the redefined one. So property redefinition must be understood following the *masking rule*:

Requirement 3.1 (Masking rule) *Let g be a generic property, l a local property of g defined in class A. Then, l implements g for all instances of A,* unless otherwise stated, *i.e. unless the considered object is an instance of a subclass of A which redefines g. So, if $A' \prec A$ and l' redefines l in A' ($l' \ll l$), l' will* mask *l for all instances of A', direct and indirect alike.*

So in the considered example of Figure 4-b, `area::Q` implements `area` for all instances of `Q`, except instances of `Re`, and `area::Re` implements it for all instances of `Re`, including those of `Square`. This means that *defining* a property is stronger than *inheriting* it (Ducournau and Habib, 1991). In this perspective, method combination is a way to recover some monotonicity—if l' calls `super`, all instances of A' will behave like those of A, plus some extra behaviour. This encourages *behavioral subtyping* (Liskov and Wing, 1994).

3.2.2. *Conflict solutions*

Unlike the generic property conflict, there is no intrinsic solution to this problem. Consequently, either the programmer or the language must bring additional semantics to solve the local property conflict and this additional semantics may depend on the kind of properties, i.e. attribute, method or type. There are roughly three ways to do it:

Nothing, i.e. error. The considered language does not specify any answer to local property conflicts but it signals an error at compile-time. This forces the programmer to define a local property in the class where the conflict appears. In this redefinition, a call to `super` is often desired but it will be ambiguous (see below). A variant of this approach makes the class which introduces the conflict (`Square`) *abstract*, by implicitly defining an *abstract* local property [11] (`area`) in this class (Nystrom et al., 2006). This forces the programmer to define the property in all direct non-abstract subclasses.

Selection. The programmer or the language arbitrarily select the local property to inherit. In many dynamic languages, the choice is made by a *linearization* (Section 4); in Eiffel, the programmer can select the desired property with the `undefine` inheritance clause.

Combining. For some values or particular properties—especially for some *meta-properties* [12] —the conflict must be solved by combining the conflicting values. For instance, in Java, when the conflict concerns the declared exceptions of methods, it should be solved by taking the union of all declared exceptions. Another example is the return type of methods which can be covariantly redefined—the lower bound of conflicting types, if it exists, is the solution. Virtual types require a similar solution. It follows that the type system must include *intersection types*. Static typing will be examined in Section 5. Combining is also needed by Eiffel contracts with disjunction of preconditions and conjunction of postconditions. Generally speaking, *method combination* is often the solution when several methods conflict—this is examined hereafter.

So the solution is *redefining*, *selecting* or *combining*, or a mix since the redefinition can be combined with the selection. For instance, in C++, selection can be done by redefining the local property with a static call, done with the :: operator, to the selected one.

3.2.3. *Call to `super` and method combination*

Call to `super` presents a similar, but more general, kind of conflict. Suppose first that the current local property l has been determined without local property conflict. Let $g \in G$ be the considered generic property and $c \in X$ be the receiver's dynamic type. This means that $spec(g, c) = \{l\}$ and $loc(g, c) \backslash \{l\} = supl(l)$—see definitions in equations (12-15), Section 2.4.1. In other words, all other local properties that might be combined are in $supl(l)$. Therefore, the situation is exactly the same as late binding, except that the selection or combination process must now consider $supl(l)$ instead of $spec(g, c)$. Then the set $\min_{\ll}(supl(l)) = \{l' \mid l \ll_d l'\}$ may not be a singleton, thus making `super` as ambiguous as a local property conflict. This is, for instance, the case if l has been defined to solve a local property conflict. A solution is to consider that `super` is legal only when $\min_{\ll}(supl(l))$ is a singleton. This is the approach of Eiffel, with `Precursor`. If this is not a singleton, an explicit selection among the conflicting local properties is required. An alternative would be a static call, as with :: in C++. Static calls have, however, a major drawback, as they may yield *multiple evaluations* of the local property defined in the root `Quadrilateral` of the diamond example in Figure 4—consider a method `foo` with a local definition in each four classes, with each definition, except the diamond root, statically calling all local properties defined in its direct superclasses.

In a conflict case, when l has been arbitrarily selected in $spec(g, c)$, the point is that $supl(l)$ is far from including all other local properties in $loc(g, c)$—actually $spec(g, c) \backslash \{l\}$ and $supl(l)$ are disjoint non-empty sets. In a linearization framework, as in Clos, the analogue of `super`, called `call-next-method`, involves calling the local property following l in the linearization of $loc(g, c)$. So, `call-next-method` avoids multiple

[11] When considering *abstract local properties*, conflict definition must be slightly adapted. An actual conflict occurs when $spec(g, c)$ contains several non-abstract properties. If all members of $spec(g, c)$ are abstract, the inherited property in c is also abstract. Of course, the implied abstract local property must not been considered in subsequent conflicts.

[12] Local properties are objects, hence 'composed' of some meta-properties. Hence, solving the conflict amounts to combining the conflicting objects and, for each meta-property, selecting or combining. This is for instance quite explicit in the Clos *slot-definition* specifications.

evaluation risks. Finally, call to `super` can also occur in a local property which has been invoked by a call to `super`, not by a 'primary' late binding, but this does not yield any complication.

Finally, there are four possibilities:

(i) `call-next-method` and linearizations, discussed hereafter;
(ii) a constrained keyword `super`, which behaves like `Precursor` in EIFFEL, i.e. only sound when there is no conflict;
(iii) static call, with a fully qualified syntax such as `foo::C`;
(iv) a qualified use of `super`, which allows the programmer to explicitly reference the class when there is a conflict, e.g. `super⟨C⟩`.

Among these options, we exclude static calls because they explicitly mention the property name, like `super` in SMALLTALK and JAVA. The three others are all acceptable and `area::Rhombus` would be replaced by `super⟨Rhombus⟩`, but only within the code of an `area` method. Moreover, all three mechanisms are compatible with each other. A language can provide all of them—linearizations are more flexible, whereas unqualified `super` has a restricted use and its qualified version can lead to multiple evaluation.

Finally, `call-next-method` should also be considered in a top-down fashion. In BETA, `inner` is restricted to single inheritance but CLOS wrappers are integrated with `call-next-method`—actually, the same `call-next-method` function works top-down in wrappers and bottom-up in ordinary methods. In contrast, constrained or qualified `super` cannot work top-down, since there is no way to deal with conflicting subclasses. So, we shall not develop a top-down version for `call-next-method`, since it is exactly like the bottom-up mechanism, in reverse order.

4. Linearizations

Linearizations have been introduced in the early 80s in FLAVORS, an object-oriented extension of LISP (Weinreb and Moon, 1980). They have been widely used in many dynamically typed object-oriented languages such as LOOPS and COMMON LOOPS (Stefik and Bobrow, 1986; Bobrow et al., 1986), CLOS (DeMichiel and Gabriel, 1987), YAFOOL (Ducournau, 1991), POWER-CLASSES (ILOG, 1996), DYLAN (Shalit, 1997), PYTHON (van Rossum and Drake, 2003), etc. To our knowledge, their only use in statically typed object-oriented languages with full multiple inheritance concerns constructors and destructors in C++ (Huchard, 2000). They are, however, also used in statically typed mixin-based languages, such as GBETA (Ernst, 1999) and SCALA (Odersky et al., 2008)—see Section 6.2.

4.1. *Principle*

The linearization principle involves computing, for each class $c \in X$, a total ordering on the set of superclasses of c, i.e.

$$supc(c) \triangleq \{c' \mid c \preceq c'\}. \tag{16}$$

This ordering is called *class precedence list* in CLOS and *method resolution order* in PYTHON.

Definition 4.1 (Class linearization) *Given a class hierarchy $\mathcal{H}$, a* class linearization *is defined as a function $clin^{\mathcal{H}} : X \to (X \to \mathbf{N})$ (clin for short) such that $clin(c)$ (noted hereafter $clin_c$ for the sake of readability) is a bijective function $clin_c : supc(c) \to 0..|supc(c)| - 1$ (aka a* permutation*). It yields a total order $(supc(c), \leq_{clin(c)})$, whereby $x \leq_{clin(c)} y \overset{\Delta}{\iff} clin_c(x) \leq clin_c(y)$. Moreover, $clin_c(c) = 0$ for all c.*

An alternative notation is the following: $clin(c) = (c_0, c_1, ..., c_k)$, with $c = c_0$, $supc(c) = \{c_i \mid i \in 0..k\}$ and $clin_c(c_i) = i$ for all $i \in 0..k$.

Class linearizations only involve the poset $(X, \preceq)$ and they are just dedicated to the solution of local property conflicts. Therefore class linearizations must be mapped from classes to local properties, i.e. from the poset $(supc(c), \preceq)$ to the poset $(loc(g, c), \ll)$, for each generic property $g \in G_c$.

Definition 4.2 (Local property linearization) *Given a class hierarchy $\mathcal{H}$, equipped with a class linearization clin, a* local property linearization *is defined by the function $llin : G \times X \to (L \to \mathbf{N})$ such that*

llin$(g,c) = (l_0, l_1, ..l_m)$, *with* $loc(g,c) = \{l_i \mid i \in 0..m\}$ *and* $0 \leq i < j \leq m \Rightarrow clin_c(def(l_i)) < clin_c(def(l_j))$. *It yields a total order* $(loc(g,c), \leq_{llin(g,c)})$ *analogous to* $(supc(c), \leq_{clin(c)})$.

In this framework, the selection function *sel* selects the first property in this ordering—i.e. $sel(g,c) = l_0$—and the call to `super` is carried out by the `call-next-method` mechanism.

Definition 4.3 (Call next method) *The* call next method *mechanism relies on the partial function* $cnm : X \times L \rightarrow L$ *such that, with the previous definition notations,* $cnm(c, l_i) = l_{i+1}$ *when* $i < m$, *and* $cnm(c, l_m)$ *is undefined.*

So, when used for combination, linearizations avoid possible multiple evaluations which may occur with static calls or qualified `super`. However, it is essential to note that, in the expression $cnm(c, l_i)$, c is not the class which defines l_i, i.e. $def(l_i)$, but the receiver's dynamic type [13], hence $c \preceq def(l_i)$. Single inheritance ensures that $l \ll cnm(c,l)$ but this is no longer verified as soon that there is a local property conflict.

4.2. *Requirements*

Several theoretical studies have determined what should be a 'good' linearization. We review here their main conclusions. All proofs can be found in the referenced papers.

Linear extensions. In order to ensure that the selection respects the *masking rule* (Req. 3.1), i.e. that no other property would be more specific—the total order must be a *linear extension* [14] (Ducournau and Habib, 1987). This means that

$$c \preceq c' \preceq c'' \Longrightarrow clin_c(c') \leq clin_c(c'') \tag{17}$$

or, equivalently, that the restriction $\preceq / supc(c)$ is a subset of $\leq_{clin(c)}$, for all $c \in X$. This implies that the selected property is taken from the *conflict set* (Ducournau et al., 1995)—i.e. $def(sel(g,c)) \in cs(g,c)$. This requirement is easy to meet and is satisfied in most recent languages—it was actually satisfied in mid-80s languages (Stefik and Bobrow, 1986; Bobrow et al., 1986; Moon, 1986; DeMichiel and Gabriel, 1987)—with the notable exception of Python (for 'classic classes' only). When used for selection, linear extensions have the desired behaviour when there is no local property conflict—they select the single most specific local property. Hence, when there is a conflict, linearizations represent only a default selection mechanism and the programmer can switch it off anyway by simply redefining the property to solve the conflict. From now on and unless otherwise stated, we shall consider that all linearizations are linear extensions.

Monotonicity, i.e. linearization inheritance. Another important requirement is that the class linearization should be *monotonic*—i.e. the total ordering of a class extends that of its superclasses (Ducournau et al., 1992, 1994; Barrett et al., 1996; Ernst, 1999). This amounts to inheriting linearizations—i.e. $\leq_{clin(c')}$ is a subset of $\leq_{clin(c)}$, for all $c \prec c'$ in X—or, equivalently, it means that:

$$c \preceq c' \preceq c'', c''' \Longrightarrow \big(clin_c(c'') \leq clin_c(c''') \iff clin_{c'}(c'') \leq clin_{c'}(c''')\big) \tag{18}$$

Since $clin_c(c) = 0$, a monotonic linearization is always a linear extension. Moreover, when the linearization is a linear extension, (18) is obviously verified for all $\preceq$-related pairs c'', c'''—therefore, monotonicity only constrains $\preceq$-unrelated pairs.

When the linearization is used for combination, monotonicity makes the order of method invocations preserved by inheritance—of course, *llin* is also monotonic. Furthermore, monotonicity implies a nice property when the linearization is used for selection—namely, a class always behaves like at least one of its direct superclasses or, equivalently, *inheritance cannot skip a generation.*

However, the need for monotonicity is not as easy to meet as that for linear extension. Actually, given a class hierarchy $(X, \preceq)$ equipped with a monotonic linearization *clin*, and two classes $c_1, c_2 \in X$, it may

[13] So, the existence of this 'next' method cannot always be statically—i.e. when compiling c—ensured and an auxiliary function `next-method-p` allows the programmer to check it at run-time. However, in the present framework, when the linearization is a linear extension (see hereafter), this run-time check is only required when the method has been declared *abstract* in superclasses.

[14] Linear extensions are also called *topological sorting* (Knuth, 1973).

be impossible to extend the hierarchy to a common subclass of c_1 and c_2 without losing monotonicity, because $clin_{c_1}$ and $clin_{c_2}$ conflict on some pair x, y—i.e. $clin_{c_1}(x) < clin_{c_1}(y)$ and $clin_{c_2}(x) > clin_{c_2}(y)$. This *linearization conflict* involves a cycle in the union of $\leq_{clin(c_1)}$ and $\leq_{clin(c_2)}$.

Local and extended precedence order. The actual linearization principle is to totally order unrelated superclasses, especially direct superclasses. As such orderings are rather arbitrary, they are usually explicitly given by the programmer, as the order of superclass declaration—hence, in Definition 2.5, `supernames` is a *totally ordered* set. These orders are called *local precedence orders* in CLOS, and the linearization is required to respect them. This is, however, not always possible, for the same reasons as monotonicity. In the following, *lpo* denotes a function which associates with each class the ordered list of its direct superclasses.

An *extended precedence order* has also been considered (Huchard et al., 1991). It is defined as a kind of bottom-up propagation of local precedence orders. It is the third constraint underlying the C3 linearization (see below).

Poset-based, modular and transitivity-free. We consider that linearizations are based on posets, like inheritance. This has several consequences. The linearization should only depend upon the topology of the inheritance graph—for instance, it should not consider class names, or programmer ages. Moreover, two isomorphic inheritance graphs should have isomorphic linearizations—the linearizations which satisfy this isomorphism requirement are called *language-independent* (Forman and Danforth, 1999). It entails that the linearization must be *context-free*, hence *modular*. Indeed, the linearization of an inheritance graph cannot depend on the whole including hierarchy.

Introducing isomorphism of inheritance graph implies to formally define inheritance graphs:

Definition 4.4 (Inheritance graph) *Let $\mathcal{H}$ be a class hierarchy, $c \in X^{\mathcal{H}}$ be a class. Then the* inheritance graph *of c in $\mathcal{H}$, is the 4-tuple $\langle c, supc(c), \preceq, lpo\rangle$ uniquely determined by c, where the two latter components are restricted to the second one. Let $\mathcal{H}$ and $\mathcal{H}'$ be two class hierarchies, $\langle c, supc(c), \preceq, lpo\rangle$ and $\langle c', supc'(c'), \preceq', lpo'\rangle$ be two inheritance graphs from the respective hierarchies. Then $f : supc(c) \to supc'(c')$ is an* isomorphism of inheritance graphs *iff*

$$
\begin{aligned}
f(c) &= c' \\
\forall x, y \in supc(c), x \preceq y &\iff f(x) \preceq' f(y) \qquad \textit{(hence, this is a poset isomorphism)} \\
\forall b \in supc(c), lpo(b) = (c_1, c_2, .., c_k) &\iff lpo'(f(b)) = (f(c_1), f(c_2), .., f(c_k))
\end{aligned}
$$

Finally, the aforementioned limits of linearizations can be stated in the following concise way:

Proposition 4.1 (Ducournau et al., 1994; Forman and Danforth, 1999) *There are no language-independent linearization that are monotonic for all inheritance hierarchies.*

So, in practice, monotonicity must likely remain a desired but not required property. For instance, the strategy proposed by Forman and Danforth (1999) involves considering whether the disagreement provoked by such a conflict is 'serious'. For his part, Ernst (1999) considers linearizations as total preorders and proposes to unify all classes on a cycle by merging their definitions. Of course, such merging may provoke some conflicts between local properties, but the author does not analyze them. On the contrary, in a subsequent paper, he seems to consider that the idea is not feasible (Ernst, 2002).

A last point is much more debated. The metamodel is only based on sets and posets, so we think that linearizations should not be sensitive to transitivity edges—hence it would be meaningless to allow the programmer to declare that the direct superclasses of C are A and B when A is already a superclass of B (see Note 6). The effect of transitivity edges would only be to add extra ordering in the local precedence orders, that should be verified by the resulting total order. However, sensitiveness to transitivity edges is often counter-intuitive and may hinders monotonicity, since extra constraints are added. Nevertheless, most languages—e.g. CLOS, DYLAN, PYTHON—have adopted transitivity-dependent linearizations.

4.3. *Some linearizations*

Many linearizations have been studied and a few of them are used in production languages. We present them in a transitivity-free variant, i.e. we assume that there is no transitivity edges and undirect superclasses are not considered.

Notations. Let A be a class and B_i, $i = 1..n$ its direct superclasses, i.e. $lpo(A) = (B_1, .., B_n)$. $L(C)$ denotes $clin(C)$, the linearization of C, i.e. a list of classes. Moreover, $\centerdot$ is the LISP `cons` operator, i.e. $x \centerdot (y_1, .., y_k) \triangleq (x, y_1, .., y_k)$. Its priority is lower than that of all concatenation operators.

4.3.1. *In* COMMON LOOPS *and* C++

This the simplest linearization which gives a linear extension, but it does not satisfy the other requirements—see for instance (Ducournau and Habib, 1987; Huchard, 2000) for counter examples. It was introduced in COMMON LOOPS and is used in C++, for constructors and destructors, and in SCALA. $L(A)$ can be defined in the following way:

$$L(A) = A \centerdot L(B_1) \oplus \cdots \oplus L(B_n)$$

where $\oplus$ is a special concatenation (`append`) which removes duplicates, i.e. $() \oplus x = x \oplus () = x$ and:

$$(a_1, .., a_k) \oplus (b_1, .., b_m) = \begin{cases} (a_2, .., a_k) \oplus (b_1, .., b_m) \text{ if } a_1 \in \{b_1, .., b_m\} \\ a_1 \centerdot (a_2, .., a_k) \oplus (b_1, .., b_m) \text{ otherwise} \end{cases}$$

The definition is sound because $\oplus$ is associative.

4.3.2. *C3 linearization*

It has been designed for DYLAN but finally not applied to it for compatibility reasons (Barrett et al., 1996). In PYTHON, it is used for 'new-style' classes only. C3 is the only linearization which satisfies all requirements—actually it takes its name from three criterion, namely linear extension, monotonicity and extended precedence order. (Ernst, 1999) gives the following definition:

$$L(A) = A \centerdot L(B_1) \boxplus \cdots \boxplus L(B_n)$$

where $\boxplus$ is the usual `merge` operator extended to partial orders, i.e. $() \boxplus x = x \boxplus () = x$ and:

$$(a_1, .., a_k) \boxplus (b_1, .., b_m) = \begin{cases} a_1 \centerdot (a_2, .., a_k) \boxplus (b_2, .., b_m) \text{ if } a_1 = b_1 \\ a_1 \centerdot (a_2, .., a_k) \boxplus (b_1, .., b_m) \text{ if } a_1 \notin \{b_1, .., b_m\} \\ b_1 \centerdot (a_2, .., a_k) \boxplus (b_2, .., b_m) \text{ if } a_1 \in \{b_1, .., b_m\} \wedge b_1 \notin \{a_1, .., a_k\} \\ \text{impossible if } a_1 \in \{b_2, .., b_m\} \wedge b_1 \in \{a_2, .., a_k\} \end{cases}$$

However, this definition is not sound because $\boxplus$ is not associative. The correct definition, i.e. the original definition by (Barrett et al., 1996), is obtained with an n-ary operator $\bigboxplus_i$:

$$L(A) = A \centerdot \bigboxplus_i L(B_i)$$

Let $L_i = (a_1^i, .., a_{k_i}^i)$ be p lists to merge, for $i \in 1..p$. $\bigboxplus_i L_i$ is defined as follows: let j be the least integer such that $a_1^j \notin (a_2^i, .., a_{k_i}^i)$ for all $i \neq j$. Then,

$$\bigboxplus_i L_i = \begin{cases} a_1^j \centerdot \bigboxplus_i L_i' \text{ if } j \text{ exists, where } L_i' = \begin{cases} L_i \text{ if } a_1^j \neq a_1^i \\ (a_2^i, .., a_{k_i}^i) \text{ if } a_1^j = a_1^i \end{cases} \\ \text{impossible otherwise.} \end{cases} \tag{19}$$

If there is no such j, then there is one or more cycles, i.e. a chain $n_0, n_1, .., n_q = n_0$ with $1 < q \leq k$, such that, for all $0 < i \leq q$, $a_1^{n_{i-1}} \in \{a_2^{n_i}, .., a_{k_{n_i}}^{n_i}\}$. Conversely, if there is a cycle, the algorithm will reach a state where there is no such j.

4.3.3. *Other linearizations*

Some other linearizations cannot be expressed in terms of concatenation or merging. The CLOS linearization involves taking successive minimals according to the union of specialization ($\prec$) and local precedence orders (*lpo*). When there are several minimals, the algorithm selects the single minimal which is a direct superclass of the most recently taken class in the linearization. It is not always monotonic. (Ducournau et al., 1994) proposes a monotonic linearization by computing a linear extension of the union of linearizations of all direct superclasses ($\cup_i L(B_i)$). It applies the COMMON LOOPS linearization to the graph resulting from the union. The DYLAN linearization is a mix of both approaches. It takes successive minimals according to the union of the superclass linearizations. When there is a choice, it uses the same criterion as CLOS.

Besides the fact that these linearizations do not satisfy at least one requirement, their definitions and algorithms are more obscure and less intuitive for programmers. In contrast, C3 meets all requirements and its algorithm is much more intuitive, though its results are not easily predictable.

4.4. *Prospects*

Linearizations can be further improved in several ways.

Quasi-monotonicity. Monotonicity is desired but difficult to maintain. When defining a class C with two direct superclasses B_1 and B_2, a linearization conflict may occur, making it impossible to compute a monotonic linearization of C, whatever the *lpo* between B_1 and B_2 is. However, the programmer may prefer a non-monotonic linearization to a failure. Hence, one can modify the (19) formula as follows. In the otherwise case, when, for all j, there exists i such that $a_1^j \in (a_2^i, .., a_{k_i}^i)$, j is then defined as the least integer such that a_1^j is minimal according to $\preceq$ among $\cup_i L_i$. Of course, a_1^j must also be removed from all $L_i, i \neq j$.

Partial class linearizations. It follows from Definitions 4.1 and 4.2 that linearizations are only intended to order local properties, for selecting and combining them. Therefore, only $llin(g, c)$ should be required to be monotonic linear extensions. For instance, if there is no local property conflict, local precedence orders do not matter and any linear extension makes a good linearization. In the case of a linearization conflict, if there is no conflict between local properties in the cycle, this cycle in $clin(c)$ does not appear in any $llin(g, c)$ and the cycle ordering does not matter at all. So, instead of computing $clin(c)$ as a total order, one could restrict it to a partial order that would only totally order all local properties, for each generic property $g \in G_c$. More precisely, $\leq_{clin(c)}$ could be defined as $\bigcup_{g \in G_c} def(\leq_{llin(g,c)})$. Thus, monotonicity should be more often—but still not always—preserved.

Specific property linearizations. An alternative involves allowing the programmer to specify a partial linearization for each generic property—there is no necessity for two generic properties to combine their local properties in the same order. So each class could provide a default linearization which could be overridden by the programmer for individual generic properties. This is, however, just a research issue—contrary to the previous proposals which can remain hidden from the programmer, specific linearizations would need some syntactic constructs.

4.5. *Conclusion on linearizations*

Linearizations have often been criticized:

– As a selection mechanism, they would be rather arbitrary; when they are linear extensions, they actually represent a default selection mechanism in case of conflict—the programmer can always switch it off by redefining the considered property in the class introducing the conflict; hence, the only assumption is the masking rule and, in the situation of Figure 4-b, the programmer must redefine `area` if he/she disagrees with masking.

– As a combination mechanism, linearizations would break class modularity because $l \ll cnm(c, l)$ is not always verified (Snyder, 1991). This seems, however, unavoidable in method combination when inheritance is multiple and the modularity breaking is not more important than with usual late binding—the essence of OO programming is that a function call can invoke an unforeseeable function. Moreover, linearizations can coexist with other combination techniques, e.g. the qualified call to `super` that we propose, but the supposed antimodular behavior of linearizations must be carefully balanced against double evaluations yielded by static calls.
– The choice of a precise linearization would be arbitrary; actually, all arguments lead us to think that C3 is the best known linearization; moreover, according to (Huchard et al., 1991), it should be the only one, because it satisfies also the extended precedence order.
– Linearization would also be hard to understand by programmers. This has been actually improved. Firstly, the declarative definition of C3 is far more comprehensive than the algorithmic definitions of CLOS and DYLAN linearizations. Forman and Danforth (1999) make a pedagogical presentation of linearizations. Moreover, development tools like ECLIPSE could easily provide the programmer with some help for solving conflicts, combining methods, proposing *lpo* and giving precise diagnoses of non-monotonic situations. Actually, like any high-level construct, the intuition about linearization is a matter of habits. It would seem that CLOS and PYTHON programmers have adopted linearizations without too much trouble.

5. Static Typing

Static typing and type safety are mostly beyond the scope of this paper. Hence, this section does not present a precise type system but only examines how static typing interact with multiple inheritance and how different type systems could be adapted to the metamodel. There are several acceptions of the term 'type safety'. Usually, a program fragment is said *type safe* if it does not cause type errors to occur at run-time. Recently, Cardelli (2004) proposed a slightly different definition, whereby only *untrapped* errors are rejected—hence, run-time type checks are generated by the compiler in the unsafe cases. Hereafter, we shall speak of 'strict' and 'permissive' type safety policies. Hence, an unsafe code fragment will be rejected in a strict policy but specific unsafe code fragments can be accepted in a permissive policy.

5.1. *Record types and generic properties*

Usual type theories apply to our framework, with just a slight difference. Object types are generally presented as *record types*, i.e. functions from *names* to *types* (Cardelli, 1988). However, names are now inadequate and they must be replaced by generic properties. This should not have any effect on the considered type system. Substituting generic properties to names would likely be sufficient to adapt the metamodel to any other formal semantics of object-oriented programs such as object calculi (Abadi and Cardelli, 1996).

5.2. *Specialization vs. subtyping*

It is commonly agreed that classes are not types and specialization is not subtyping (Cook et al., 1990). However, commonly used languages, like C++, C#, EIFFEL and JAVA, identify classes to types and class specialization ($\preceq$) to subtyping ($<:$), with the proviso that class specialization is constrained to coincide with subtyping [15]. This constraint concerns property redefinition, through the well known *contravariance rule* which states that return types must be covariant but parameter types contravariant (Cardelli, 1988). However, class specialization and static typing entail a much debated tradeoff between type safety and expressiveness. The contravariance rule is essential in a strict type safe policy, but some authors argue that strictly contravariant parameter types have turned out to be useless for modeling real-world applications [16]. On the contrary, modeling real world often requires covariant parameter types—Shang (1996), Meyer (1997),

[15] As most assertions about C++, this one presents some exceptions, e.g. *private inheritance*.
[16] For instance, Shang (1996) ends its often cited paper by "We can live without contravariance".

Ducournau (2002b), Büttner and Gogolla (2004) present arguments in favor of this covariant policy. In practice, most languages have type-invariant parameters and covariant return types, thus ensuring strict type safety. On the contrary, some other languages, e.g. EIFFEL, BETA, O_2 (Bancilhon et al., 1992) and JAVA (for arrays only), have adopted a covariant policy, ensuring only permissive type safety.

CWA vs. OWA. The unsafety yielded by covariance is highly correlated to whether type checking is done under the *closed world assumption* (CWA) or the *open world assumption* (OWA). The CWA requires the whole class hierarchy to be known at compile time, whereas, under the OWA, the hierarchy which is known at compile-time represents only a small part of actual program hierarchies. CWA corresponds to global compilation and OWA to separate compilation. CWA is a condition of the *system level safety* argued by Meyer (1997). This assumption is unfortunately contrary to the reusability and modularity principles.

Under the OWA, covariance entails that all method calls are potentially unsafe—unless one of the two specific situations: (i) all parameters are primitive or *final* types, without any potential proper subtype; (ii) the receiver is proved to always have an *exact type* (Bruce et al., 1998), as in EIFFEL's *catcall rule*, i.e. its static and dynamic types are equal. Obviously, unconstrained covariance presents too much unsafety under the OWA—i.e. with a strict policy, most method calls would be rejected and, with a permissive policy, numerous compiler-generated run-time type checks would make the code both unsafe and inefficient. Therefore, a sensible tradeoff between expressivity, efficiency and type safety would be to declare whether a generic property is covariant or not when introducing it—it must be done however for each parameter whose type is not *final*.

5.3. *Virtual types and intersection types*

Virtual types represent a safer alternative to covariant parameters which can be strictly safe, according to Torgersen (1998), or more permissive as in BETA. They must not be confused with *virtual classes* (Madsen and Møller-Pedersen, 1989), which are *virtual*—in the sense of Note 2, page 4—*inner classes*. We follow here the presentation of virtual types by Torgersen (1998), while also considering the more permissive BETA subtyping rule.

Virtual types. A virtual type is a kind of attribute declared in a class and denoting a type. The metamodel applies to virtual types in a straightforward way. In some class C, the declaration of a virtual type T may take two forms, either $T <: D$ or $T = D$, where T is the name of the virtual type and D some concrete type, e.g. a class name. The former forces T to be a subtype of D in all subclasses of C and the latter is a *final* declaration fixing T for all subclasses. If D is *final*, both forms are equivalent—hence, we assume that $T <: D$ implies that D is not *final*. The key idea of virtual types is that (i) as virtual types they must be covariantly redefined, unless they are final ($T = D$), (ii) as a parameter or return type they are invariant, (iii) they have no proper subtypes, unless they are final, (iv) they are not even their own subtype, unless they are final or 'anchored' to the same receiver. Besides these explicit virtual types, `SelfType` (aka `MyType` or `ThisType`) represents the implicit declaration `SelfType`$<: D$ in each class D—it corresponds to `like current` in EIFFEL. It is never final, unless the class itself is final.

When a variable, say x, is typed by a virtual type T (noted $x : T$), the dynamic type of the x value must be a subtype of the virtual type T in the *dynamic type* of `self`. In other words, $x : T$ stands for $x :$ `self`$.T$. Let us consider now the code fragment {`y:A; z:D; y.foo(z);`}, whereby the parameter of `foo` has the virtual type T, and $T <: D$ in A. The call is unsafe because the dynamic type of `y` could be $B \prec A$, such that B redefines $T = E$, where E is some proper subclass of D. However, the BETA permissive rule accepts it, at the expense of a compiler-inserted run-time type check.

Under the CWA, virtual types and covariant parameters are strictly equivalent—when a parameter type is redefined in a subclass, a virtual type is introduced in the superclass and substituted to the parameter type. Hence, under the OWA, virtual types are a convenient way to declare that a parameter type is covariant—accordingly, all parameter types become invariant. If such a type redefinition has not been anticipated when

introducing the generic property, the programmer must manage the redefinition in an exceptional way, by explicitly checking the parameter dynamic type. All this analysis is close to (Shang, 1996).

Return types. Regarding the return type, free covariant redefinition is type safe. However, parameter and return types are often correlated and there are strong arguments for keeping also return types invariant. Let $C_k \prec C_{k-1} \prec ... \prec C_0$ and $D_k \prec D_{k-1} \prec ... \prec D_0$ be two specialization chains, T be a virtual type introduced in C_0 and defined as $T <: D_i$ in each class C_i. Suppose also that `foo` and `bar` are methods introduced in C_0, the former with a parameter of type T and the latter with return type T. Then, if the static type of `x` is some C_i, `x.foo(x.bar)` is safe, because its involves the *same virtual type*, depending on the *same receiver*. In contrast, if the return type of `bar` were D_i in each C_i, `x.foo(x.bar)` would be unsafe. We shall see that method combination entails a similar conclusion—it might be preferred to also forbid explicit return type redefinition. Hence, the method and attributes types would remain invariant, all covariant redefinition going through virtual types.

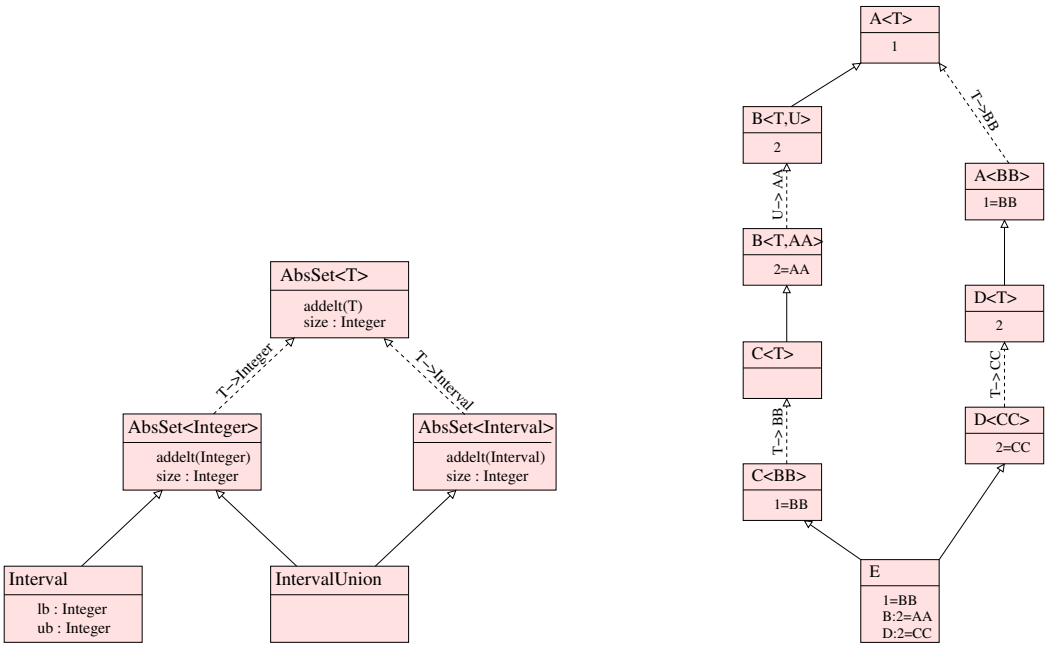
Intersection types. So far there is nothing new except substituting generic properties to names. Local property conflicts require combining types. It follows from the specification of virtual types that the conflict solution is quite constrained when the conflicting property represents a virtual type. Indeed, the possible conflicts can take three forms in the two conflicting classes C and C':
– $T = D$ and $T = D'$—this is an error unless $D' = D$;
– $T <: D$ and $T = D'$—this is an error unless $D' <: D$;
– $T <: D$ and $T <: D'$—this is an error unless D and D' are *compatible*, i.e. they can have a common subclass.
In all three cases, the error is unrecoverable, i.e. the two classes C and C' are not *compatible*. Moreover, this is the only exception to the ability of defining a common subclass of two $\prec$-unrelated classes—two classes are *compatible* unless they present an error in a virtual type conflict [17]. The third conflict case implies *intersection types* (Reynolds, 1996; Compagnoni and Pierce, 1996)—i.e. T must be a subtype of $D \cap D'$. The essential difference with *meet types* in (Cardelli, 1988) is that, here, two atomic types, i.e. two classes, A and B can have a non-trivial intersection type $A \cap B \neq \bot$ if they are compatible, whereas, for Cardelli's record types, two atomic types have only a trivial meet type. Another difference relies on generic properties. For Cardelli, two record types can be incompatible because the same field name is associated with two incompatible types in both records. Generic properties make it possible with virtual types only. With type-variant redefinition, the conclusion would be similar. Covariance implies intersection types and contravariance would add *union types*—the only difference is that a union type always exists.

Compatibility is however not exempt from the possibility of *actual type conflicts*, since $x \cap y = \bot$ when there is no common subclass to x and y, at the time when a value of type $x \cap y$ is needed. However, this error cannot be detected under the OWA, since all future subclasses have not to be known. A solution might be to consider that a class that yields a proper intersection type is abstract, forcing the programmer to define at least a subclass which would specify a concrete subtype in the intersection. This is similar to the aforementioned solution to local property conflicts proposed by Nystrom et al. (2006), but the necessity is likely greater here.

5.4. *Genericity*

Virtual types represent also an alternative to parametrized classes (Thorup and Torgersen, 1999; Igarashi and Pierce, 1999). This suggests that *formal type parameters* might be included in the meta-model in a way analogous to virtual types. Hence, the form $T <: D$ in the class A would correspond to the parametrized class $A\langle T <: D\rangle$ and $T = D'$ to the instantiation $A\langle D'\rangle$. This is not an equivalence, just a strong similarity. It follows that, according to both usual type theories and virtual types, two instances of the same parametrized class, say $A\langle C\rangle$ and $A\langle D\rangle$ are never related by subtyping.

[17] This entails possibly cyclic constraints such as C and C' are compatible iff D and D' are. This is however not a problem since this is an optimistic rule—compatibility is the default and only incompatibility must be justified.



-a Local property conflict -b Generic property conflict

Fig. 5. Multiple Inheritance and Generics. A class cannot specialize two distinct instances of the same parametrized class (a); a fully qualified syntax disambiguates parameter position conflicts (b). Solid lines represent specialization and dashed lines instantiation.

A formal integration to the metamodel would require to add a new association, namely the *instantiation* corresponding to substituting a concrete type to a formal type. We do not formalize it here and only sketch the way multiple inheritance conflicts occur and their solution. As the formal type has a lexical scope, it cannot serve as a property identifier and the parameter position must be used instead. This position must take into account the possible parameter instantiation in superclasses—i.e. a class inherits the formal types instantiated in its superclasses and their positions is not free. Moreover, as a property name, it may yield generic property conflicts, when a class inherits two formal type parameters introduced in different classes with the same position. Figure 5-b depicts such a situation, with three classes A, B and D each introducing a new formal parameter, and class E inheriting from chains of successive specialization and instantiation, whereby AA, BB and CC are concrete types.

Inheriting multiple instances of the same parametrized class amounts to local property conflict and is strongly constrained. In Figure 5-b, it occurs with parameter `1`, which is fortunately instantiated by the same class BB in the conflicting classes $A\langle BB\rangle$ and $C\langle BB\rangle$. In contrast, consider the example in Figure 5-a. Sets can be implemented in different ways and the abstract class $\texttt{AbsSet}\langle T\rangle$ represents the interface and implements some default methods for all set implementations. `Interval` describes an efficient implementation of integer sets when all their elements are consecutive. An efficient way to implement integer sets whose elements are almost consecutive is a normalized union of intervals. The `IntervalUnion` class obviously specializes $\texttt{AbsSet}\langle\texttt{Integer}\rangle$ but the programmer may want to consider also interval unions as sets of intervals, thus making `IntervalUnion` specialize both $\texttt{AbsSet}\langle\texttt{Integer}\rangle$ and $\texttt{AbsSet}\langle\texttt{Interval}\rangle$. This would be poor design, since a set of intervals is not a set of integers from the mathematical standpoint. Anyway, this might be a pragmatic implementation. This is however not possible in the meta-model, since this situation is similar to the first case of virtual type conflicts, where $T = \texttt{Integer}$ and $T = \texttt{Interval}$. The functions `addelt` could be disambiguated by static overloading but the function `size` would remain ambiguous—would it be the interval or integer number? An explicit type parameter, e.g. $\texttt{size}\langle\texttt{Integer}\rangle$ or $\texttt{size}\langle\texttt{Interval}\rangle$, could clear up the situation, but implementing it would be a problem. Obviously, *homogeneous implementations* would make it impossible, since both methods cannot stand at the same position in

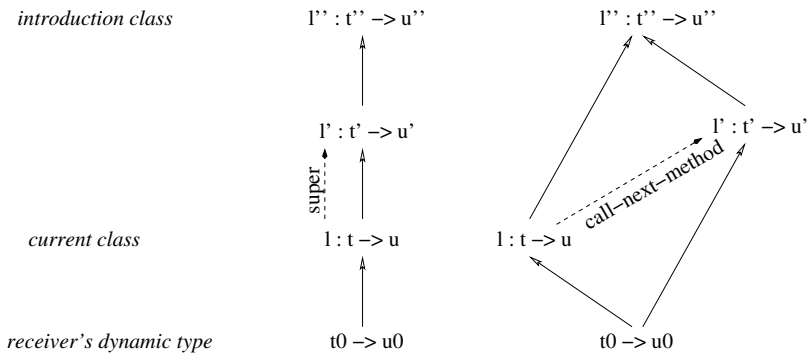


Fig. 6. Typing `super` and `call-next-method`—solid lines represent redefinition and some implied subtyping rule, dashed lines calls to `super` and `call-next-method`.

the `AbsSet⟨T⟩` method table [18] . On the contrary, C++ *heterogeneous implementation* would likely make it possible, at the expense of some implied *repeated inheritance* (see Section 6.1). Note that this example and the rule that forbids subtyping between two instances of the same parametrized class can be generalized—an object cannot be instance of two such instances.

A complete modeling of parametrized classes is beyond the scope of this paper. One should for instance examine the case where a class with two parameters, e.g. `map⟨T,U⟩`, is specialized in order to force both parameters to be equal, e.g. `selfmap⟨T⟩<: map⟨T,T⟩`. The same situation could occur with virtual types. Anyway, this is not a matter of multiple inheritance.

5.5. *Method combination*

Method combination presents also some interesting typing questions. In practice, calls to `super` or `call-next-method` often represent one of two specific pipeline patterns: (i) the method parameters are passed to the callee as they were received by the caller; conversely, (ii) the value returned by the callee is returned at once by the caller (*tail call* in the following). In the former case, the parameter can be implicit and both cases can be combined in a `return super` instruction, without parameters. In CLOS, the `call-next-method` syntax allows for the former case, since its usage without parameters implies passing all parameters as they were received. Both cases are of course type safe in a type invariant framework, including virtual types. It means that, with the BETA permissive rule, the primary method call may need run-time type checking but all subsequent `call-next-method` with implied parameters or tail call are type safe.

However, in a type-variant setting, method combination is more paradoxical. Contrary to `super`, `call-next-method` is not a static call and the type of the actually called method may be as general as its type in the introduction class. In Figure 6 (right), `call-next-method` in method l might call l', whose type is statically unknown and is only statically known to be bounded by the type of l''. Therefore, (i) if the return types are strictly covariant, tail calls are unsafe—it is another argument for using virtual types for return types; (ii) if the parameter types are strictly contravariant, `call-next-method` without parameters is unsafe; and paradoxically, (iii) if the parameter types are strictly covariant and `call-next-method` is always without parameters, it is safe since $t0 <: t \cap t'$.

6. Comparison with other approaches to multiple inheritance

The main contribution of this metamodel is to make the specifications of languages unambiguous when naming problems occur, i.e. when there are several properties with the same identifier in the context of a

[18] This is the reason why, in JAVA, a parametrized method must be static. The implementations of generics lie between two extremes (Odersky and Wadler, 1997). In heterogeneous implementation, e.g. C++ templates, each instance of the parametrized class is separately compiled. In homogeneous implementation, e.g. JAVA 1.5, a single instance is compiled, after replacing each formal type by its bound. Intermediate policies still present a research issue.

single class. Such naming problems mostly occur with multiple inheritance or with *static overloading* (Meyer, 2001). In the absence of these naming problems, there is no need, except conceptual, to distinguish generic properties from property identifiers and all languages agree with the metamodel, at least at the generic property level. On the contrary, when naming problems arise, different languages have different behaviors, and our claim is that the present metamodel is a good basis for better specifications of the relationship between object-oriented entities and their names.

6.1. *Inheritance and metamodels in some object-oriented languages*

We review hereafter the most commonly used languages.

6.1.1. *Dynamic typing*

In the SMALLTALK terminology, 'method' and 'method selector' respectively denote local and generic properties. Nevertheless, selectors are simply reified as symbols and there is no equivalent for attributes (*instance variables* in SMALLTALK terminology). In CLOS (Steele, 1990), 'method' and 'generic functions' stand for local and generic properties—they are reified, but multiple dispatch changes the model as they do not belong to classes, hence are not inherited in the usual meaning. As for attributes (*slots* in CLOS terminology), they are reified into two kinds of *slot descriptions*, which can be *direct* or *effective*: but both must be understood as local properties (Kiczales et al., 1991). Moreover, as aforementioned, in both SMALLTALK and CLOS, dynamic typing cannot distinguish two properties with the same name, whether it is a selector, a generic function or a slot. This does not hinder a fully qualified syntax, but makes it impossible to statically instantiate the metamodel when parsing method calls in method definitions (Req. 2.1). In the example of Figure 3, a call site `x.department` could alternatively reference, at run-time, `Researcher:department` or `Teacher:department`, according to the dynamic type of `x`. So either Requirement 2.1 is dropped, but the call site becomes ambiguous, or the requirement is kept but both `department` are unified. As aforementioned, in dynamic typing, a naming convention is the only way to anticipate conflicts. For instance, in the CLOS *meta-object protocol*, all generic functions related to classes are prefixed by '`class-`', e.g. `class-name`, `class-slots`, `class-precedence-list`, etc. Despite this limitation, these languages are the only ones whose terminology is at least partly suitable for distinguishing the two key notions that we have called local and generic properties. In the following languages, one word (method, feature, etc.) stands for both notions.

6.1.2. *Static typing*

JAVA requires two simple adaptations of the metamodel. First, static overloading implies that the 'name' of a property also involves its parameter types. This ensures that overloaded methods represent different generic properties. JAVA class hierarchies are also made of two kinds of entities, classes and interfaces, which are disjoint at the notable exception of the hierarchy root—i.e. $X = X_c \cup X_i$ and $X_c \cap X_i = \{\texttt{java.lang.Object}\}$. Classes are in single inheritance—$(X_c, \preceq)$ is a tree—and interfaces imply multiple subtyping but cannot specialize classes, hence $\prec$ is a subset of $(X_c \times X) \cup (X_i \times X_i)$. Interfaces define only *abstract* methods. This type system can be understood as the result of statically typing SMALLTALK, by adding interfaces for all methods introduced by more than one class. So far, JAVA is fully compatible with the metamodel [19] . However, multiple inheritance is possible with interfaces and, when a generic property conflict occurs, JAVA cannot distinguish between two methods with the same name and signature. Accordingly, there is no reification of generic properties, either in the introspection facilities (package `java.lang.reflect`), or in reflective extensions of the language, as OPENJAVA or JAVASSIST, though (Chiba, 1998) acknowledges the need for a metamodel. It seems that other languages with multiple subtyping, like C# or ADA 2005, suffer from the same inability to distinguish conflicting generic properties.

EIFFEL is also almost fully compatible, but only in common usage. In EIFFEL terminology, 'feature' stands for property, without distinguishing the two kinds, even though a notion of *feature seed* could be understood as the introduction of generic properties (Meyer, 1992). Feature renaming allows the programmer to deal

[19] Until JAVA 1.4, a slight restriction concerns static overloading, but it has been raised in JAVA 1.5 (Ancona et al., 2000, 2001).

with generic property conflicts in the desired way. However, a conservative use of renaming is advisable, since full usage of the `rename` clause is not compatible with the metamodel: (i) in subclasses, a feature with the new name can coexist with the old-name feature, as two distinct features; (ii) in a class, if two features from different seeds are inherited under the same name, then they can be locally merged; (iii) repeated inheritance, i.e. multiple inheritance of the same class, or even transitivity edges are accepted, with a lot of renamings. Moreover, EIFFEL does not follow the masking rule (Req. 3.1) and finds extra local property conflicts, as in Figure 4-b, but the programmer can restore the desired behavior with the `undefine` keyword—this is however clumsy.

Among the most commonly used languages, C++ is the least compatible with the metamodel. Besides *static overloading* which is managed in the metamodel as for JAVA [20], multiple inheritance raises several difficulties. Firstly, the keyword `virtual` [21] is mandatory in inheritance, to avoid duplication of attributes introduced in a superclass inherited through multiple paths. In C++ jargon, our proposal resorts to *shared multiple inheritance*, in contrast with *repeated multiple inheritance* [22]. Whereas virtual inheritance is fully compatible with the OWA and dynamic loading, non-virtual inheritance is only sound under the CWA, when the super-classes of the considered class have no shared superclass. Furthermore, inheritance can be made private, thus changing the relationship between classes and types. Anyway, when it is used in the way the most appropriate to multiple inheritance, i.e. with public virtual inheritance, C++ is like JAVA, i.e. it does not distinguish between two methods with the same signature, introduced in different unrelated superclasses. However, this does not apply to attributes which are well defined, at least when `virtual` is used. This dissymmetry between methods and attributes represents one of the most striking flaws of the language specifications—e.g. two attributes named `foo` would be distinguished but their accessors `get-foo` would be confused with each other. Finally, contrary to EIFFEL, C++ signals local property conflicts according to the masking rule.

Overall, only EIFFEL allows the programmer to implement any multiple-inheritance model which follows our metamodel. Conversely, not all EIFFEL programs satisfy it. In contrast, C++ and JAVA do not allow the programmer to define a class as a subclass of any two unrelated preexisting classes (or interfaces). When each of both superclasses owns a generic property of the same name, it is impossible to express that their common subclass own both properties. This is a rare but inescapable obstacle to reusability.

At first sight, all three languages could be slightly modified to conform to our proposal. Both JAVA and C++ require some fully qualified syntax, together with checking for ambiguous unqualified names. In EIFFEL, once a feature has been renamed, its old name should no longer be accessible in the considered class and its subclasses. This should be sufficient for making all three languages sound—however, in C++, it only applies to virtual inheritance. Sound non-virtual inheritance also requires to forbid the diamond situation when the repeated class is not empty, i.e. it introduces some attributes—actually, in the original paper describing it, Krogdahl (1985) forbids all diamond situations.

6.1.3. *Modeling languages*

Eventually, one cannot speak of meta-modeling without considering UML, where all entities are meta-defined. As a matter of fact, concerning properties, the UML metamodel is left unfinished—i.e. the *Features diagram* of (OMG, 2004) shows only one kind of entity called 'feature'. Page 38, the specification says that "one classifier may specialize another by adding or redefining features". There is no way to map this single term to our metamodel. Büttner and Gogolla (2004) explain how specialization and redefinition are specified in UML 2.0—they make it clear that UML 2 introduces *covariant overriding* but show that combining possibly covariant redefinition and overloading would lead to unspecified semantics. However, they do not consider

[20] C++ name overriding yields a difference—when an overloaded method is redefined in a subclass, all the homonymic methods defined in superclasses become invisible from the subclass, unless they are in turn redefined. However, this is a matter of visibility, not of the metamodel.

[21] Though more obscure, the use of `virtual` in inheritance is similar to its use for methods—a 'virtual base class' (in C++ jargon) has a 'redefinable' position in the object layout. See also Note 2, page 4.

[22] It is common opinion that "repeated inheritance is an abomination" (Zibin and Gil, 2003).

virtual types—actually, when covariant parameter types are restricted to virtual types, covariance remains compatible with static overloading (Torgersen, 1998).

6.2. *Alternatives to full multiple inheritance*

Several alternatives have been proposed, which rely on a degraded form of multiple inheritance, e.g. JAVA or C# interface multiple subtyping, *mixins* (Bracha and Cook, 1990) or *traits* (Ducasse et al., 2005). *Interfaces* have been discussed about JAVA—they only imply slightly constraining the metamodel.

Mixins (aka *mixin classes*) are commonly presented as an alternative to full multiple inheritance. They first appear as a programming style in LISP-based languages [23], before becoming explicit patterns or even first class entities in theories or actual programming languages. Mixin proposals are numerous and variable—so this section must not be considered as a complete survey. Generally speaking, mixins are *abstract classes* with some restrictions in the way they are defined and related to other classes or mixins. Above all, a mixin is not self-sufficient—it must be used to qualify a class. To take up a distinction from linguistics, a class is *categorematic*, like a noun, whereas a *mixin* is *syncategorematic*, like an adjective (Lalande, 1926).

For instance, in the SCALA language (Odersky et al., 2008), one can define a class C such that C `extends` B `with` M, where B is the direct superclass of C and M is a *mixin*. An additional constraint is that the single direct superclass of M must be a superclass of B. Actually, the constraint is a little bit more general—the single mixin M can be replaced by a set of $\prec$-related mixins such that all their superclasses [24] must be superclasses of B. Intuitively, the effect of this definition is to copy the definition of M into B (Fig. 7-b). An alternative and more formal view involves *parametrized heir classes*, whereby C `extends` $M\langle B\rangle$ (Fig. 7-c). This is the common way of using mixins with C++ *templates* (VanHilst and Notkin, 1996; Smaragdakis and Batory, 2002). The *heterogeneous* implementation of templates (see Note 18) makes it possible, but mixins cannot be separately compiled. On the contrary, the *homogeneous* implementation of generics makes it impossible in JAVA. Finally, in a last view compatible with a *homogeneous* implementation, M is transformed into a *proxy*, which involves both an interface and a class: C implements the interface and is associated with the class by an aggregation, in such a way that each instance of C contains exactly one instance of M (Fig. 7-d). The latter approach has also been called *automated delegation* (Viega et al., 1998).

Actually, mixins are not exempt from multiple inheritance conflicts—for instance, B and M may each introduce a property with the same name `bar`, or redefine the same property `foo` introduced in their common superclass A (Fig. 7). Hence, mixins are not incompatible with the present metamodel, which could be extended to include them, in the same manner as for JAVA interfaces. Generic property conflicts are exactly the same and require the same solutions. Actually, most mixin-based languages, e.g. SCALA, do not recognize these conflicts and 'solve' them by *unification* (Section 3.1). MIXJAVA presents a notable exception, with a `view` keyword which resembles full qualification (Flatt et al., 1998). Regarding local property conflicts, there is no uniform policy among various mixin proposals, but all involve some explicit or implicit linearization. The previous definitions—by copy or parametrization—yield the same result and the mixin M overrides the direct superclass B. SCALA uses the same linearization as COMMON LOOPS where M—or all the mixin set—precedes the totally ordered superclasses.

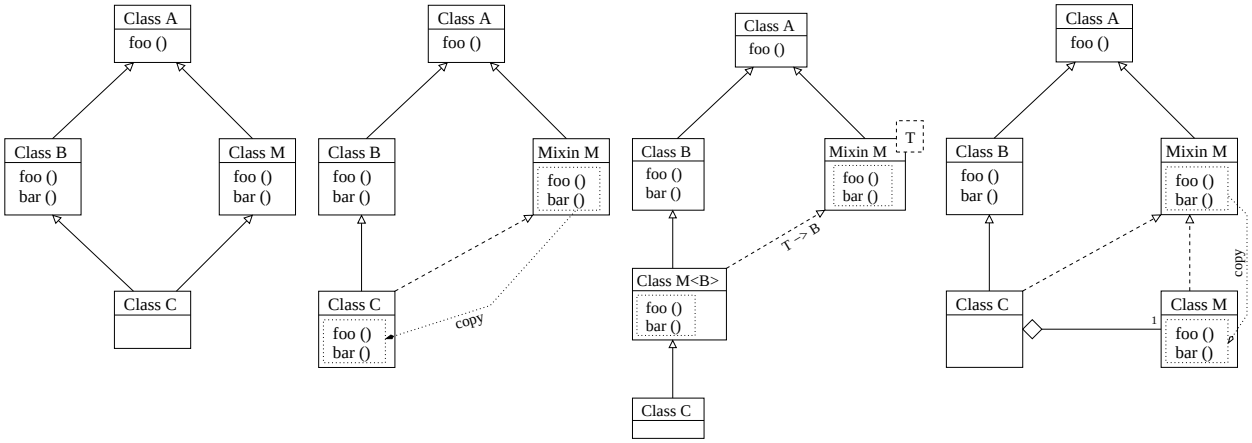
Traits [25] (Ducasse et al., 2005) are a variant of mixins which provides a more formal way of combining them, with a finer grain. Traits are intermediate between SCALA mixins and JAVA interfaces—they can only define methods, contrary to mixins which can also define attributes [26], but these methods can have

[23] The term has been originally coined according to the FLAVORS metaphor—classes are compared to ice-cream flavors, the hierarchy root is vanilla flavor and inheritance is viewed as mixing-in flavors (Weinreb and Moon, 1980). This confusion between aggregation and specialization has unfortunately perverted a lot of authors—it can be found for instance in the pastry example of multiple inheritance in the CLOS specifications (Steele, 1990) and it is likely at the basis of C++ repeated inheritance.

[24] Of course, here 'superclass' denotes a class, not a mixin.

[25] The term 'trait' is overloaded. Actually, SCALA uses 'trait' instead of 'mixin' but it merges different approaches, so we prefer to avoid confusions and call them mixins. 'Trait' was previously used in SELF (Ungar et al., 1991) with yet another, though similar, meaning. In C++, 'trait' denotes a programming pattern which allows the programmer to somewhat 'refine' primitive types, especially characters.

[26] This limitation of traits may be caused by the target language, SMALLTALK.



-a Multiple inheritance -b Copy view of mixins -c Parametrized heir -d Proxy

Fig. 7. Multiple Inheritance and Mixins—(a) in multiple inheritance, the example reproduces the two conflicting situations in Figures 3 and 4—`foo` stands for `area` and `bar` for `department`; (b) the same example involving a copy view of mixins; (c) mixins as parametrized heir classes; (d) mixins as proxies. Solid arrows denote class specialization (JAVA `extends`) and dashed arrows represent interface implementation (JAVA `implements`).

implementations, contrary to JAVA interfaces. Moreover, traits cannot have superclasses or even 'super-traits', but they can be explicitly combined to make composite traits. Different combination operators allow the programmer to precisely manage name conflicts, but at the local property level only—like their underlying languages, SMALLTALK and JAVA, traits do not recognize generic property conflicts. In a static typing framework, traits are types, like usual mixins (Nierstrasz et al., 2006).

6.3. *Related works*

Multiple Nested Inheritance. Nystrom et al. (2006) follow an analysis of multiple inheritance very similar to ours, though the lack of metamodel makes the authors uniformly speak of *name conflicts*. Generic property conflicts are solved by fully qualified syntax, whereas local property conflicts make the considered class *abstract*—hence, they require further redefinitions. However, method combination is not addressed.

On the distinction between homonymic properties. Nystrom et al. (2006) assign to Borning and Ingalls (1982) the paternity of the distinction between properties according to the class which introduces them: "The distinction between name conflicts among methods introduced in a common base class and among methods introduced independently with possibly different semantics was made as early as 1982 by Borning and Ingalls." Actually, we carefully re-read this 4-page paper and did not find any trace of that. We only found a mention of *compound selectors* for disambiguating calls to `super`. Anyway, this 1982 paper is quite informal and cannot yield precise semantics. For instance, it is impossible to determine whether the paper implies the *masking rule*, or not. Besides this reference to (Borning and Ingalls, 1982), Nystrom etal. consider that their view of multiple inheritance derives from *intersection types* (Compagnoni and Pierce, 1996; Reynolds, 1996)—see Section 5. We did not find, in the referenced papers on *intersection types*, any information about the way name conflicts are managed—it might be as well unions of names. In these affiliation proceedings, the historical paper by Cardelli (1988) must also be ruled out—it proposes a theory of record types without classes, hence, without any notion of introduction—actually, it should have been more appropriately entitled "a semantics of multiple subtyping". Knudsen (1988) attempts to analyze *name collisions* in multiple specialization hierarchies. Though he considers qualifying property names by some class, or even a class path, he does not consider the introduction class. Moreover, he concludes that the programmer is in charge of resolving all collisions, including redefinitions, whereas our proposal restricts the programmer's role to local property conflicts.

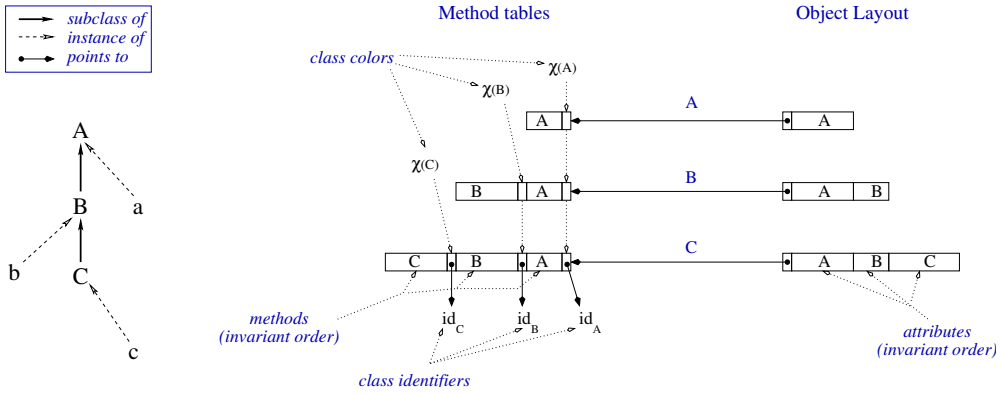


Fig. 8. A small class hierarchy with 3 classes A, B and C with their respective instances, a, b and c. Method tables—including class identifiers for Cohen's display—(left) and object layout (right) in single subtyping. This diagram respects the following conventions. In method tables (left), A (resp. B, C) represents the set of addresses of methods introduced by A (resp. B, C). Method tables are drawn from right to left to reduce edge crossing. In object layouts (right), the same convention applies to attributes but the tables are drawn from left to right. Solid lines are pointers.

Altogether, to our knowledge (Ducournau et al., 1995), this idea of distinguishing properties according to the class which introduces them goes back to some studies on the notion of *point of view* [27], in a framework of dynamic typing and knowledge representation (Carré and Geib, 1990; Dugerdil, 1991). We are not aware of any prior or independent mention of it.

Higher-order meta-modeling. Since BETA *virtual classes* (Madsen and Møller-Pedersen, 1989), there have been many proposals of nested classes (Ernst, 2003; Nystrom et al., 2006), where each class can play, in turn, the role of an ordinary class and/or the role of a *module* (or *package*) containing a set of classes. Ducournau et al. (2007) propose a 2-level metamodel, where the class-module metamodel is isomorphic to the property-class metamodel presented here. The *import* relationship between modules is then analogous to inheritance between classes and property redefinition is translated, at the class-module level, by the notion of *class refinement* which allows the programmer to incrementally define classes. Of course, import conflicts are managed as inheritance conflicts.

7. Implementation of multiple inheritance

Object-oriented programming languages represent an original implementation issue due to *late binding*. The underlying principle is that the address of the actually called procedure is not statically determined at compile-time, but depends on the dynamic type of the *receiver*. An issue similar to message sending arises with attributes (aka *instance variables*, *slots*, *data members* according to the languages), since their position in the object layout may depend on the object's dynamic type. Furthermore, subtyping introduces another original feature, i.e. run-time subtype checks, which can be generated by the compiler in unsafe situations and are the basis for so-called *downcast* operators.

All three mechanisms need specific implementations, data structures and algorithms. It turns out that an efficient implementation of multiple inheritance remains an open issue, especially in the framework of dynamic loading, i.e. under the OWA. Actually, in this framework, we do not know any implementation which does not entail a significant overhead, *even when multiple inheritance is not used*.

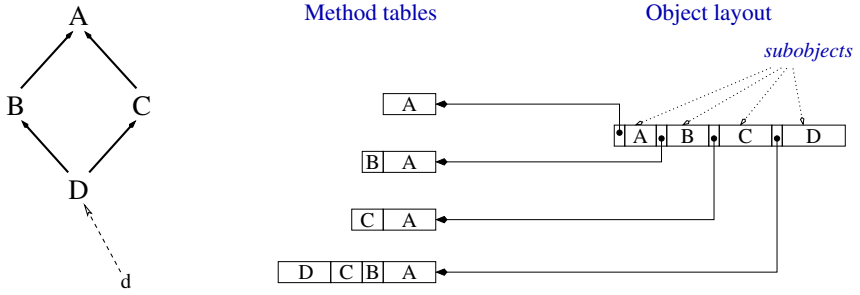


Fig. 9. Object layout and method tables with multiple inheritance: an instance d of the class D is depicted—the diagram follows the same convention as in Figure 8

7.1. *Under the OWA*

Single subtyping. The point with OWA is to efficiently precompile late binding during class definition (Def. 2.5). In separate compilation of statically typed languages, this is generally done with method tables (aka *virtual function tables* in C++ jargon). Method calls are then reduced to function calls through a small fixed number (usually 2) of extra indirections. An object is laid out as an attribute table, with a header pointing at the method table and possibly some added information, e.g. for garbage collection. With single inheritance, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single direct superclass (Figure 8). The resulting implementation respects two essential *invariants*: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position* of attributes and methods in the table does not depend on the dynamic type of the object. Moreover, the subtype test can be implemented in a similar way, with the technique known as Cohen (1991) display. Therefore, all accesses to objects are straightforward. This simplicity is due to both static typing and single inheritance—dynamic typing adds the same kind of complication as multiple inheritance, since the same property name may be at different places in unrelated classes.

Subobject-based implementation. With multiple inheritance, the two invariants of reference and position, which hold with single inheritance and static typing, cannot hold together, at least when class implementations are separately computed, e.g. at load-time. Therefore, the C++ implementation of multiple inheritance in static typing is based on *subobjects*. The object layout is composed of several subobjects, one for each superclass of the object's class. Each subobject contains attributes *introduced* by the corresponding class, together with a pointer to a method table which contains the methods *known* by the class (Figure 9). A reference of a given static type points at the subobject corresponding to this type. This is the C++ implementation, when the keyword `virtual` annotates each superclass (Ellis and Stroustrup, 1990; Lippman, 1996; Ducournau, 2002a). It is time-constant and compatible with dynamic loading, but the number of method tables is equal to the cardinality of $\preceq$, instead of being equal to the number of classes. Accordingly, the total size is no longer linear in the cardinality of the relation $\preceq$, it is linear in the product of relation cardinality and class number and, in the worst case, cubic in the number or classes. Object layouts also contain extra pointers to method tables, which can be as many as the object attributes. Furthermore, all polymorphic object manipulations, which are quite numerous, need pointer adjustments between the static and target types, as they correspond to different subobjects. Finally, the main drawback of this implementation is that its overhead remains the same with single inheritance hierarchies. In contrast, C++ non-virtual inheritance can be considered as an optimization, similar to single inheritance implementation, which is sound only when the class hierarchy does not include any diamond (or undirected cycle). However, this property cannot be checked in the OWA.

[27] The `view` keyword of MixJava (Flatt et al., 1998) seems to belong to the same family though we cannot confirm that it actually behaves like full qualification, apart from the fact that there are mixins.

Method combination. As aforementioned, `super` involves a static call in single inheritance. In contrast, `call-next-method` is purely dynamic under the OWA. Therefore, a usual implementation, e.g. that of SCALA, consists of implementing `call-next-method` as a special message sending. Let us assume that the method `foo` defined in class A uses `call-next-method`: this is translated as a call to a special method, say `cnm-foo-A`, which is introduced by A and implemented by each of its subclasses as the actual local property `foo` which should be called for its direct instances. Hence, method combination is not any more costly than usual method invocation, and it does not entail any overhead when `call-next-method` is not used.

Alternatives. With multiple subtyping, e.g. in JAVA and C#, all mechanisms concerning a class-typed receiver are implemented as in single inheritance. In contrast, interface-typed receivers—this is the so-called `invokeinterface` primitive, which is not renowned for its efficiency (Alpern et al., 2001; Ducournau, 2008)—imply the same difficulties as multiple inheritance. Furthermore, usual virtual machines require type-invariant references, hence forbid subobjects. Hence, they mostly use ad hoc techniques such as caching and searching.

Currently, alternatives to subobjects for efficiently implementing multiple inheritance in a dynamic loading framework remain an open issue. *Perfect hashing* (Ducournau, 2008) has been proposed for subtype testing and method invocation, the latter in the specific case of JAVA interfaces. In principle, it could also apply to full multiple inheritance but its time-efficiency presents, in theory, a significant overhead w.r.t. single inheritance and remains to assess in practice. Most JAVA implementations use rather ad hoc techniques that may not be scalable with a language like SCALA where each class and trait is transtlated into one interface.

Mixins. Mixins are also—sometimes above all—a specification of how things are implemented. The code of the mixin M (Fig. 7) is at least indirectly copied into the class C. Hence, mixins are compatible with a single-inheritance implementation, which is presumed to be more efficient than full multiple-inheritance implementations, or which is imposed by the target runtime system, e.g. in SCALA. However, in SCALA, the code is compiled into the JVM bytecode (or the .NET CRL) at the expense of translating each SCALA type into a JAVA interface. Therefore, the resulting implementation makes extensive use of method invocation on an interface-typed receiver. Moreover, the SCALA approach is a midterm between copy and proxy. The mixin M is compiled as an abstract class and each mixin method is actually compiled as a `static` method with an extra explicit parameter, the current receiver, and the copy is done by defining in C the corresponding method which simply calls the static one.

7.2. *Under the CWA*

The complete knowledge of the whole class hierarchy offers several ways to efficiently implement multiple inheritance. We present briefly three representative approaches, namely *devirtualization*, *binary tree dispatch* and *coloring*.

Devirtualization. Global compilation allows optimizing the C++ subobject-based implementation, by removing all unnecessary `virtual` keywords. A static global analysis can detect all diamond situations and determine which superclass declarations require the `virtual` keyword (Gil and Sweeney, 1999; Eckel and Gil, 2000). The same thing can also be done for methods, e.g. when a method has a single implementation.

Binary tree dispatch (BTD). In SMART EIFFEL, the GNU EIFFEL compiler, method tables are not used. Instead, objects are tagged by their class identifier and all polymorphic operations are implemented using small dispatch trees. The technique is called *binary tree dispatch* (BTD) (Zendra et al., 1997; Collin et al., 1997). However, the approach is practical only because compilation is global, hence all classes are statically known, including the program entry point. Therefore, a type analysis restricts the set of *concrete types* (Bacon and Sweeney, 1996; Grove and Chambers, 2001) and makes dispatch efficient.

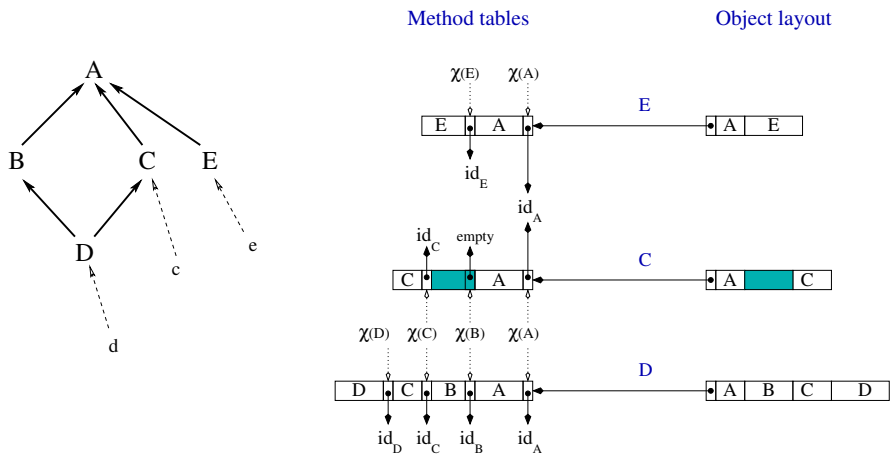


Fig. 10. Unidirectional coloring applied to classes, methods and attributes—the holes (in grey) in the C tables are reserved space for B in the tables of D. A and B classes are presumed to have the same implementation as in Figure 8 and the diagram follows the same convention.

Coloring. The coloring approach is quite versatile and naturally extends the single inheritance implementation to multiple inheritance, with the same efficiency. The technique takes its name from *graph coloring*, as its computation amounts to coloring some particular graph [28]. *Method coloring* was first proposed by Dixon et al. (1989) under the name of *selector coloring*. Pugh and Weddell (1990) and Ducournau (1991) applied *coloring* to attributes and Vitek et al. (1997) to classes (under the name of *pack encoding*).

The general idea of coloring is to keep the two *invariants* of single inheritance, i.e. *reference* and *position*. An injective numbering of attributes, methods and classes verifies the position invariant, so this is clearly a matter of optimization for minimizing the size of all tables—or, equivalently, the number of *holes*, i.e. empty entries. However, this optimization cannot be done separately for each class, it requires a global computation for the whole hierarchy. Minimizing the total table size is akin to the *minimum graph coloring* problem (Jensen and Toft, 1995), which is a well-known problem in the compiler construction area, e.g. for register allocation (Chaitin, 1982). Like minimum graph coloring, the coloring problem considered here has been proven to be NP-hard in the general case. Therefore heuristics have been proposed and many experiments have shown their efficiency and that the technique is tractable. Finally, an important improvement is *bidirectionality*, introduced by Pugh and Weddell (1990), which involves using positive and negative offsets and reduces the hole number. Figure 10 depicts the implementation yielded by unidirectional coloring in the diamond example (Fig. 9), assuming that the implementation of A and B is identical to that of Figure 8. Hence, computing the tables for C must reserve some space for B in the tables of D, their common subclass. Thus, some holes appear in the C tables and these holes are filled, in D, by all data specific to B. In contrast, B and E have no common subclass and share the same indexes in both method table and object layout. In bidirectional coloring, all holes would have been saved by placing C at negative offsets.

A detailed presentation of coloring is beyond the scope of this paper and the reader is referred to (Ducournau, 2006) which reviews the approach. The point to get is 2-fold: (i) in practice, object layout, method tables and code sequences are exactly those of single inheritance, except for the presence of holes; (ii) this is obtained by rather sophisticated algorithms which require complete knowledge of the class hierarchy.

[28] This graph is a *conflict graph* with a vertex for each class and an edge between any two vertices that have a common subclass and thus must have their attributes (resp. methods or class IDs) stored at distinct offsets, since attributes (resp. methods or class IDs) of both classes coexist in objects (resp. method tables) of the common subclass. So, this conflict graph is the graph which supports all possible property conflicts, generic and local alike.

7.3. *Global compilation vs. global linking*

Without loss of generality, the CWA assumption can be obtained by *global compilation* or *global linking*. In the former, the whole program is compiled in a single stage—this is for instance the case for most implementations of the EIFFEL language, e.g. SMART EIFFEL. Global compilation obviously affords the best optimizations but it presents also significant drawbacks: (i) it does not allow the programmer to check the safety of a single piece of code—it only provides *system level safety*; (ii) usual optimizations include *dead code elimination*—in practice only the live code is actually compiled and the dead code is not even checked; (iii) the source code of the whole libraries must be available to all programmers. So, leaving the modularity brought by separate compilation may be considered too highly payed for program optimization. An alternative was already noted by Pugh and Weddell (1990). Coloring does not require knowledge of the code itself, but only of the "model" (aka "schema") of the classes, all of which is already needed by separate compilation of object-oriented programs, namely the model of the superclasses of the currently compiled class, together with other classes which are used for typing the code of the current class. This class model is included in specific header files (in C++) or automatically extracted from source or compiled files (in JAVA). In our framework, this is merely an instance of the metamodel. Therefore, the compiler can separately generate the compiled code without knowing the value of the colors of the considered entities, representing them with specific symbols. At link time, the linker collects the models of all the classes and color all the entities, before substituting values to the different symbols, as a linker commonly does.

This global linking approach applies to coloring, not to BTD or devirtualization. However, other powerful optimizations can be added to this compilation framework, e.g. type analysis and dead code elimination, which extend it to other techniques like BTD, at the expense of generating stub functions at link time (Privat and Ducournau, 2005).

8. Conclusion and perspectives

In this paper, we have proposed a metamodel of classes and properties which gives a sound and simple semantics to object-oriented programming and multiple inheritance. This metamodel can have several usages. It can serve as a conceptual support for the programmer who wants to map an object-oriented model to his/her Aristotelian intuition. It can—and of course should—serve as a basis for implementing all the tools which are required for making object-oriented languages operational—i.e. compilers, run-time systems, development tools, etc. Here, we have only explored its consequences on the interpretation of multiple inheritance. The first and fundamental consequence is to distinguish two different conceptual levels for properties, inheritance and the associated conflicts. The benefits of this analysis cover also virtual types and genericity.

Generic properties are intended to model the idea of the 'same property' shared by different related classes. They represent the 'essence' of properties and closely correspond to the properties modeled by the Aristotelian syllogistic. Generic property conflicts are pure matter of names—they can be easily and safely solved in any language, provided that this language offers some fully qualified syntax. However, in dynamic typing, this does not apply but trivially, and a naming convention is required.

Local properties model the implementation of a given generic property in different classes. They may present conflicts, when a class cannot choose between different implementations for the considered generic property. This kind of conflicts is quite different, since the choice is a programmer decision. Programming languages can only provide some constructs for helping the programmer in his/her choice.

This also sheds some light on the distinction between *static overloading* and *overriding* (called here *redefinition*). 'Overloading' is often used for both—this is confusing (not to say overloading). *Static overloading* concerns different generic properties whereas *overriding* is a relationship between the local properties of the same generic property.

The specifications of most languages could adopt the present metamodel while only marginally changing language syntax, programming habits or program behavior. This would provide a terminological and conceptual basis for object-oriented programming, either for program documentation or teaching, and a sound

basis for all tools which are dedicated to program manipulation—compilers, programming environment, etc. Our claim is that languages like JAVA, C++ and EIFFEL would be markedly improved if they complied to this metamodel. In practice, at least for C++, this is however a dream, as such a compliance requires numerous changes which are incompatible with existing programs, even though the incompatible cases are likely quite marginal. Conversely, the metamodel can be easily adapted, while preserving its principle, to include interfaces or even mixins.

Generic property conflicts are not an obstacle to multiple inheritance—their solution is syntactically simple and intuitive for all programmers. This is, however, a major flaw of both C++ and JAVA, which hinders reusability. Local property conflicts present a remaining issue—method combination. There are several solutions—static calls, linearizations, mixins—but none of them is perfect. This is likely inherent to multiple inheritance.

In contrast, mixins do not answer the whole question and they need to be supplemented by our metamodel at the generic property level. At the local property level, mixins and linearization are roughly equivalent [29], since mixins are linearized—they are either introduced one by one or separately linearized. However, mixins are only equivalent to a subset of linearization-based multiple inheritance hierarchies. When generalizing the example of Figure 7, only the $CB..A$ path can contain classes between B and A, while all other 'superclasses' of C must be mixins. Hence, linearization-based multiple inheritance permits a full mixin-like programming style, but mixins only permit a very restrictive use of multiple inheritance. Our view of multiple inheritance is both set-theoretical, i.e. based on union of property sets, and order-theoretical, i.e. based on partial and total orders. Multiple inheritance mostly amounts to combining, i.e. 'mixing in', superclasses, by union of property sets, and ordering conflicting local properties, by linearization. So, 'mixin' is often used in a loose way and multiple inheritance and 'mixins' are hard to distinguish. For instance, in (Ernst, 1999), it is unclear whether 'mixin' could be uniformly replaced by 'class' without any change. Furthermore, all criticism against linearizations in a full multiple inheritance framework is also applicable to mixins. Finally, mixins have the disadvantage of adding a new kind of entities—mixins or traits—which are akin to classes, but different, rather ad hoc and which do not appear to be stable enough. On the contrary, full multiple inheritance relies only on the simple notion of class which is conceptually well understood, after centuries of Aristotelian tradition (Rayside and Campbell, 2000b,a; Rayside and Kontogiannis, 2001).

Here we did not examine the question of *visibility* (aka 'protection' or 'static access control'), i.e. the fact that a given property can be accessed from a given program point. We have two reasons to escape this point. Firstly, we consider that visibility is only a mask on a program and its associated model—it concerns the legality of an access, not the reality of the underlying accessed entities. Hence, visibility should not interact with multiple inheritance and could be independently specified. This is consistant with Requirement 2.1 but not verified by actual languages. In JAVA and C++, the combination of static overloading and visibility is such that changing the visibility of a methode may change the program behavior. The second reason is that we think that visibility should not be expressed at the class level but, instead, at the module or package level. There are strong arguments in favor of modules in object-oriented programming languages, see for instance (Szyperski, 1992), and we prefer to postpone this point for further work and address it in the context of a module system such as that of (Ducournau et al., 2007).

To sum up, our conclusions on multiple inheritance are that

(i) there are state-of-the-art solutions for syntactic conflicts; the flaws of current production languages should be fixed—though backward compatibility likely makes it impossible; new languages should adopt sound specifications—namely fully qualified names, possibly with lexical-scope renaming;

(ii) semantics conflicts are precisely defined; forcing the redefinition of the conflicting property or making the class abstract is a sensible answer; linearization-based selection is an alternative;

(iii) method combination remains the most controversial point: the supposedly antimodular behavior of linearizations must be balanced against the risk of multiple evaluation entailed by static calls;

[29] Mixins are sometimes defined by the fact that they are linearized—e.g. "They were defined as classes that allow their superclass to be determined by linearization of multiple inheritance." (Smaragdakis and Batory, 2002). This condition is necessary but not sufficient. Bracha and Cook (1990) made it clear that mixins are not ordinary classes, whereas linearizations apply to unrestricted class hierarchies.

(iv) anyway, C3 is the right linearization, possibly in its quasi-monotonic variant; partial or specific linearizations would improve their use—this is a matter of further research;

(v) virtual types and invariant signatures seem to present the best trade-off between expressiveness, efficiency and type safety, especially in the context of multiple inheritance and method combination; moreover, they yield a very simple characterization of *incompatible* classes, which cannot have a common subclass; static overloading would remain possible, though we do not defend it;

(vi) development tools should implement the metamodel in order to efficiently support the programmer for multiple inheritance—apart from backward compatibility, this is of course straightforward;

(vii) there are state-of-the-art implementations of multiple inheritance under the CWA, either at compile or link time; these implementations are as efficient as single inheritance implementations;

(viii) under the OWA, efficient implementations compatible with dynamic loading remain an open issue: the apparent scalability of C++ subobject-based implementation is likely due to an intensive use of non-virtual inheritance; alternatives such as perfect hashing still require precise assesment and comparison with both subobject-based and CWA implementations; anyway, we do not know any OWA implementation which does not entail a significant overhead, *even when multiple inheritance is not used*.

All of these conclusions hold in a static typing setting. Dynamic typing changes both the solution of syntactic conflicts and the basic implementation of object-oriented mechanisms. It remains a matter of future work.

References

Abadi, M., Cardelli, L., 1996. A theory of objects. Monographs in Computer Science. Springer Verlag.

Alpern, B., Cocchi, A., Fink, S., Grove, D., 2001. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In: Proc. OOPSLA'01. SIGPLAN Notices, 36(10). ACM Press, pp. 108–124.

Ancona, D., Drossopoulou, S., Zucca, E., 2001. Overloading and inheritance. In: 8th Intl. Workshop on Foundations of Object-Oriented Languages (FOOL8).

Ancona, D., Zucca, E., Drossopoulou, S., 2000. Overloading and inheritance in Java. In 2th Workshop on Formal Techniques for Java Programs.

Bacon, D., Sweeney, P., 1996. Fast static analysis of C++ virtual function calls. In: (OOPSLA, 1996), pp. 324–341.

Baker, H., 1991. CLOStrophobia: its etiology and treatment. ACM OOPS Messenger 2 (4), 4–15.

Bancilhon, F., Delobel, C., Kanellakis, P. (Eds.), 1992. Building an object-oriented database system: the story of 02. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Barnes, J., 1995. Programming In Ada 95, first edition. Addison-Wesley.

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., Shalit, A. L. M., Withington, P. T., 1996. A monotonic superclass linearization for Dylan. In: (OOPSLA, 1996), pp. 69–82.

Bezivin, J., Cointe, P., Hullot, J.-M., Liebermann, H. (Eds.), 1987. Proceedings of the 1st European Conference on Object-Oriented Programming, ECOOP'87. LNCS 276. Springer.

Birtwistle, G., Dahl, O., Myraug, B., Nygaard, K., 1973. Simula begin. Petrocelli Charter.

Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F., 1986. CommonLoops: Merging Lisp and object-oriented programming. In: (OOPSLA, 1986), pp. 17–29.

Borning, A., Ingalls, D., 1982. Multiple inheritance in Smalltalk-80. In: Proc. AAAI'82. pp. 234–237.

Bracha, G., Cook, W., 1990. Mixin-based inheritance. In: Proc. OOPSLA/ECOOP'90. SIGPLAN Notices, 25(10). ACM Press, pp. 303–311.

Bruce, K. B., Odersky, M., Wadler, P., 1998. A statically safe alternative to virtual types. In: (Jul, 1998), pp. 523–549.

Büttner, F., Gogolla, M., 2004. On generalization and overriding in UML 2.0. In: Patrascoiu, O. (Ed.), OCL and Model Driven Engineering, UML 2004 Conf. Workshop. Univ. Kent, pp. 1–15.

Cardelli, L., 1988. A semantics of multiple inheritance. Information and Computation 76, 138–164.

Cardelli, L., 2004. Type systems. In: Tucker, A. B. (Ed.), The Computer Science and Engineering Handbook, 2nd Edition. CRC Press, Ch. 97.

Cargill, T. A., 1991. Controversy: The case against multiple inheritance in C++. Computing Systems 4 (1), 69–82.
Carré, B., Geib, J.-M., 1990. The point of view notion for multiple inheritance. In: Proc. OOPSLA/E-COOP'90. ACM Press, pp. 312–321.
Chaitin, G. J., 1982. Register allocation & spilling via graph coloring. In: Proc. 1982 ACM SIGPLAN Symposium on Compiler Construction. pp. 98–105.
Chiba, S., 1998. Javassist—a reflection-based programming wizard for Java. In: Proc. ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java.
Cohen, N., 1991. Type-extension type tests can be performed in constant time. ACM Trans. Program. Lang. Syst. 13 (4), 626–629.
Cointe, P., 1987. Metaclasses are first class: the ObjVlisp model. In: Proc. OOPSLA'87. SIGPLAN Notices, 22(12). ACM Press, pp. 156–167.
Collin, S., Colnet, D., Zendra, O., 1997. Type inference for late binding. the SmallEiffel compiler. In: Proc. Joint Modular Languages Conference. LNCS 1204. Springer, pp. 67–81.
Compagnoni, A. B., Pierce, B. C., 1996. Higher order intersection types and multiple inheritance. Mathematical Structures in Computer Science 6 (5), 469–501.
Cook, W., Hill, W., Canning, P., 1990. Inheritance is not subtyping. In: Proc. POPL'90. ACM Press, pp. 125–135.
DeMichiel, L., Gabriel, R., 1987. The Common Lisp Object System: An overview. In: (Bezivin et al., 1987), LNCS 276, pp. 201–220.
Dixon, R., McKee, T., Schweitzer, P., Vaughan, M., 1989. A fast method dispatcher for compiled languages with multiple inheritance. In: Proc. OOPSLA'89. ACM Press, pp. 211–214.
Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A., 2005. Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28 (2), 331–388.
Ducournau, R., 1991. Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual. Sema Group, Montrouge, France.
Ducournau, R., 2002a. Implementing statically typed object-oriented programming languages. Tech. Rep. 02-174, LIRMM, Université Montpellier 2.
Ducournau, R., 2002b. "Real World" as an argument for covariant specialization in programming and modeling. In: Bruel, J.-M., Bellahsène, Z. (Eds.), Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc. LNCS 2426. Springer, pp. 3–12.
Ducournau, R., 2006. Coloring, a versatile technique for implementing object-oriented languages. Tech. Rep. 06-001, LIRMM, Université Montpellier 2, (subm. to ACM Trans. Program. Lang. Syst., rev. Aug. 2008).
Ducournau, R., 2008. Perfect hashing as an almost perfect subtype test. ACM Trans. Program. Lang. Syst., 1–51 (to appear).
Ducournau, R., Habib, M., 1987. On some algorithms for multiple inheritance. In: (Bezivin et al., 1987), pp. 243–252.
Ducournau, R., Habib, M., 1991. Masking and conflicts, or to inherit is not to own. In: (Lenzerini et al., 1991), Ch. 14, pp. 223–244.
Ducournau, R., Habib, M., Huchard, M., Mugnier, M.-L., 1992. Monotonic conflict resolution mechanisms for inheritance. In: Proc. OOPSLA'92. SIGPLAN Notices, 27(10). ACM Press, Vancouver, pp. 16–24.
Ducournau, R., Habib, M., Huchard, M., Mugnier, M.-L., 1994. Proposal for a monotonic multiple inheritance linearization. In: Proc. OOPSLA'94. SIGPLAN Notices, 29(10). ACM Press, pp. 164–175.
Ducournau, R., Habib, M., Huchard, M., Mugnier, M.-L., Napoli, A., 1995. Le point sur l'héritage multiple. Technique et Science Informatiques 14 (3), 309–345.
Ducournau, R., Morandat, F., Privat, J., 2007. Modules and class refinement: a metamodeling approach to object-oriented languages. Tech. Rep. 07-021, LIRMM, Université Montpellier 2.
Dugerdil, P., 1991. Inheritance mechanism in the Objog language: multiple selective and multiple vertical with points of view. In: (Lenzerini et al., 1991), Ch. 15, pp. 245–256.
Eckel, N., Gil, J., 2000. Empirical study of object-layout and optimization techniques. In: Bertino, E. (Ed.), Proc. ECOOP'2000. LNCS 1850. Springer, pp. 394–421.
Ellis, M., Stroustrup, B., 1990. The annotated C++ reference manual. Addison-Wesley, Reading, MA, US.

Ernst, E., 1999. Propagating class and method combination. In: (Guerraoui, 1999), pp. 67–91.
Ernst, E., 2002. Safe dynamic multiple inheritance. Nord. J. Comput 9 (1), 191–208.
Ernst, E., 2003. Higher-order hierarchies. In: Cardelli, L. (Ed.), Proc. ECOOP'2003. LNCS 2743. Springer, pp. 303–329.
Flanagan, D., Matsumoto, Y., 2008. The Ruby Programming Language. O'Reilly.
Flatt, M., Krishnamurthi, S., Felleisen, M., 1998. Classes and mixins. In: Proc. POPL'98. ACM Press, pp. 171–183.
Forman, I. R., Danforth, S. H., 1999. Putting Metaclasses to Work. Addison-Wesley.
Gil, J., Sweeney, P., 1999. Space and time-efficient memory layout for multiple inheritance. In: Proc. OOPSLA'99. SIGPLAN Notices, 34(10). ACM Press, pp. 256–275.
Goldberg, A., Robson, D., 1983. SMALLTALK: the language and its implementation. Addison-Wesley.
Goldberg, D. S., Findler, R. B., Flatt, M., 2004. Super and inner: together at last! In: Vlissides, J. M., Schmidt, D. C. (Eds.), Proc. OOPSLA'04. SIGPLAN Notices, 39(10). ACM Press, pp. 116–129.
Grand, M., 1997. JAVA Language Reference. O'Reilly.
Grove, D., Chambers, C., 2001. A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. 23 (6), 685–746.
Guerraoui, R. (Ed.), 1999. Proceedings of the 13th European Conference on Object-Oriented Programming, ECOOP'99. LNCS 1628. Springer.
Harbinson, S. P., 1992. Modula-3. Prentice Hall.
Hickey, J., 2006. Introduction to the Objective Caml programming language. `http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf`.
Horty, J., 1994. Some direct theories of nonmonotonic inheritance. In: Gabbay, D., Hogger, C. (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2: Nonmonotonic Reasoning. Oxford University Press.
Huchard, M., 2000. Another problematic multiple inheritance mechanism: Construction and destruction in C++ in the virtual inheritance case. J. Obj. Orient. Program. 13 (4), 6–12.
Huchard, M., Mugnier, M.-L., Habib, M., Ducournau, R., sep. 1991. Towards a unique multiple inheritance linearization. In: Mrazik, A. (Ed.), Proc. EurOOP'91. Bratislava.
Igarashi, A., Pierce, B. C., 1999. Foundations for virtual types. In: (Guerraoui, 1999), pp. 161–185.
ILOG, 1996. Power Classes reference manual, Version 1.4. ILOG, Gentilly.
Jensen, T. R., Toft, B., 1995. Graph Coloring Problems. John Wiley.
Jul, E. (Ed.), 1998. Proceedings of the 12th European Conference on Object-Oriented Programming, ECOOP'98. LNCS 1445. Springer.
Kiczales, G., des Rivières, J., Bobrow, D., 1991. The Art of the Meta-Object Protocol. MIT Press.
Knudsen, J. L., 1988. Name collision in multiple inheritance hierarchies. In: Gjessing, S., Nygaard, K. (Eds.), Proceedings of ECOOP'88, Oslo, Norway. LNCS 322. Springer-Verlag, Berlin, pp. 93–109.
Knuth, D. E., 1973. The art of computer programming, Sorting and Searching. Vol. 3. Addison-Wesley.
Krogdahl, S., 1985. Multiple inheritance in Simula-like languages. BIT 25 (2), 318–326.
Lalande, A., 1926. Vocabulaire technique et critique de la philosophie. Presses Universitaires de France.
Lenzerini, M., Nardi, D., Simi, M. (Eds.), 1991. Inheritance Hierarchies in Knowledge Representation and Programming Langages. John Wiley & Sons.
Lippman, S., 1996. Inside the C++ Object Model. New York.
Liskov, B. H., Wing, J. M., 1994. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16 (6), 1811–1841.
Madsen, O. L., Møller-Pedersen, B., 1989. Virtual classes. A powerful mechanism in object-oriented programming. In: Proc. OOPSLA'89. ACM Press, pp. 397–406.
Madsen, O. L., Møller-Pedersen, B., Nygaard, K., 1993. Object-Oriented Programming in the Beta Programming Language. Addison-Wesley.
Masini, G., Napoli, A., Colnet, D., Léonard, D., Tombre, K., 1991. Object-Oriented Languages. Academic Press, London.
Meyer, B., 1992. Eiffel: The Language. Prentice-Hall.
Meyer, B., 1997. Object-Oriented Software Construction, 2nd Edition. Prentice-Hall.

Meyer, B., 2001. Overloading vs. object technology. J. Obj. Orient. Program. 14 (5), 3–7.
Microsoft, 2001. C# Language specifications, v0.28. Technical report, Microsoft Corporation.
Moon, D., 1986. Object-oriented programming with Flavors. In: (OOPSLA, 1986), pp. 1–8.
Mössenböck, H., 1993. Object-Oriented Programming in Oberon-2. Springer.
Nierstrasz, O., Ducasse, S., Schärli, N., 2006. Flattening traits. J. Object Technology 5 (4), 129–146.
Nystrom, N., Qi, X., Myers, A. C., 2006. $\mathcal{JE}$: Nested intersection for scalable software composition. In: Tarr, P. L., Cook, W. R. (Eds.), Proc. OOPSLA'06. SIGPLAN Notices, 41(10). ACM Press, pp. 21–35.
Odersky, M., Spoon, L., Venners, B., 2008. Programming in Scala, A comprehensive step-by-step guide. Artima.
Odersky, M., Wadler, P., 1997. Pizza into Java: Translating theory into practice. In: Proc. POPL'97. ACM Press, pp. 146–159.
OMG, 2004. Unified Modeling Language 2.0 superstructure specification. Technical report, Object Management Group.
OOPSLA, 1986. Proceedings of the 1st ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'86. SIGPLAN Notices, 21(11). ACM Press.
OOPSLA, 1996. Proceedings of the 11th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'96. SIGPLAN Notices, 31(10). ACM Press.
OOPSLA, 1997. Proceedings of the 12th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97. SIGPLAN Notices, 32(10). ACM Press.
Privat, J., Ducournau, R., 2005. Link-time static analysis for efficient separate compilation of object-oriented languages. In: ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05). pp. 20–27.
Pugh, W., Weddell, G., 1990. Two-directional record layout for multiple inheritance. In: Proc. PLDI'90. ACM SIGPLAN Notices, 25(6). pp. 85–91.
Rayside, D., Campbell, G., 2000a. An Aristotelian introduction to classification. In: Huchard, M., Godin, R., Napoli, A. (Eds.), ECOOP'2000 Workshop on Objects and Classification, a natural convergence. RR LIRMM 2000-095.
Rayside, D., Campbell, G., 2000b. An Aristotelian understanding of object-oriented programming. In: Proc. OOPSLA'00. SIGPLAN Notices, 35(10). ACM Press, pp. 337–353.
Rayside, D., Kontogiannis, K., 2001. On the syllogistic structure of object-oriented programming. In: Proc. of ICSE'01. pp. 113–122.
Reynolds, J., 1996. Design of the programming language Forsythe. In: O'Hearn, P., Tennent, R. (Eds.), Algol-like languages. Birkhauser.
Sakkinen, M., 1992. A critique of the inheritance principles of C++. Computing Systems 5 (1), 69–110.
Shalit, A., 1997. The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language. Addison-Wesley.
Shang, D. L., 1996. Are cows animals?
URL `http://www.visviva.com/transframe/papers/covar.htm`
Smaragdakis, Y., Batory, D., 2002. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Soft. Eng. Meth. 11 (2), 215–255.
Snyder, J., 1991. Inheritance in object-oriented programming. In: (Lenzerini et al., 1991), Ch. 10, pp. 153–171.
Steele, G., 1990. Common Lisp, the Language, 2nd Edition. Digital Press.
Stefik, M., Bobrow, D., 1986. Object-oriented programming: Themes and variations. AI Magazine 6 (4), 40–62.
Stroustrup, B., 2000. The C++ programming Language, Special ed. Addison-Wesley.
Szyperski, C., 1992. Import is not inheritance. Why we need both: Modules and classes. In: Madsen, O. L. (Ed.), Proc. ECOOP'92. LNCS 615. Springer, pp. 19–32.
Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E., Leroy, P. (Eds.), 2006. Ada 2005 Reference Manual: Language and Standard Libraries. LNCS 4348. Springer.
Taivalsaari, A., 1996. On the notion of inheritance. ACM Computing Surveys 28 (3), 438–479.
Thorup, K., Torgersen, M., 1999. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In: (Guerraoui, 1999), pp. 186–204.

Torgersen, M., 1998. Virtual types are statically safe. In: Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5). San Diego, CA.

Touretzky, D., 1986. The Mathematics of Inheritance. Morgan Kaufmann Publishers, Los Altos (CA), USA.

Ungar, D., Chambers, C., Chang, B., Hølzle, U., 1991. Organizing programs without classes. Lisp and Symbolic Computation 4 (3), 223–242.

van Rossum, G., Drake, Jr., F. L., Sep. 2003. The Python Language Reference Manual. Network Theory Ltd.

VanHilst, M., Notkin, D., 1996. Using role components to implement collaboration-based designs. In: (OOPSLA, 1996), pp. 359–369.

Viega, J., Tutt, B., Behrends, R., 1998. Automated delegation is a viable alternative to multiple inheritance in class based languages. Tech. Rep. CS-98-03, University of Virginia, Charlottesville, VA, USA.

Vitek, J., Horspool, R., Krall, A., 1997. Efficient type inclusion tests. In: (OOPSLA, 1997), pp. 142–157.

Waldo, J., 1991. Controversy: The case for multiple inheritance in C++. Computing Systems 4 (2), 157–171.

Weinreb, D., Moon, D., 1980. Flavors: message passing in the Lisp Machine. Tech. Rep. 602, MIT AI Lab.

Wirth, N., 1988. The programming language Oberon. Software Practice & Experience 18 (7), 671–690.

Zendra, O., Colnet, D., Collin, S., 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In: (OOPSLA, 1997), pp. 125–141.

Zibin, Y., Gil, J., 2003. Incremental algorithms for dispatching in dynamically typed languages. In: Proc. POPL'03. ACM, pp. 126–138.