Modules and Class Refinement—A Meta-modeling Approach to Object-Oriented Languages

ROLAND DUCOURNAU and FLOREAL MORANDAT
LIRMM – CNRS and Université Montpellier II, France
and
JEAN PRIVAT
Purdue University, Indiana – USA

With object-oriented programming, *classes* and *inheritance* have sometimes been considered as a definitive answer to the need for *modularity* and *reusability* in programming languages. However, this view of object-oriented languages is often considered as too limited and many authors have claimed that classes do not make appropriate modules. On the one hand, separate concerns are generally distributed on a set of closely related classes, while on the other, a class implements several unrelated concerns. The so-called *expression problem* stresses this limitation of the traditional view on object-oriented programming, which makes it easy to add new classes but quite impossible to add concerns to previously defined classes. Accordingly, recent research has yielded a lot of proposals addressing these commonly agreed needs. These proposals are based on various notions, e.g. *virtual classes*, *higher-order hierarchies*, *nested inheritance*, *mixin layers* or *classboxes.* All of these proposals are very similar and significantly different, so comparing and evaluating them is not easy, and selecting or designing the 'best' one is a challenge. Moreover, an entire research field has emerged, i.e. *aspect-oriented programming*, dedicated to separation of crosscutting concerns.

In this paper, we propose two coupled notions of *modules* and *class refinement* which provide the required modularity and reusability and represent a pretty good response to the expression problem. However, the goal of this paper is not only to propose these new notions—they are actually close to previous propositions, though somewhat different—but also to base this proposal on very simple and intuitive notions that already underlie object-oriented languages. *Meta-modeling* is the key of our approach and we use it as a sound criterion to select a precise specification among a large state space. We first propose a meta-model of classes and properties (i.e. methods and attributes), which provides clear insight into inheritance, especially when multiple. The point is that modules and module dependence are specified by a second meta-model which is isomorphic to the first one. Finally, this proposal is not only based on conceptual and theoretical considerations. Modules and class refinement are distinctive features of a new object-oriented language, PRM, and they are intensively used in the organization of its libraries and tools.

Categories and Subject Descriptors: D.1.5 [**Programming techniques**]: Object-oriented programming; D.2.2 [**Software Engineering**]: Design tools and techniques—*modules and interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*information hiding*; D.2.13 [**Software Engineering**]: Reusable Software—*reusable models*; *reusable libraries*; D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; EIFFEL; SMALLTALK; CLOS; PRM; D.3.3 [**Programming languages**]: Language constructs and features—*classes and objects*; *inheritance*; *modules and packages*

General Terms: Languages

Additional Key Words and Phrases: aspects, classes, crosscutting concerns, expression problem, hierarchies, import, inheritance, linearization, meta-modeling, mixins, modules, multiple inheritance, overloading, redefinition, refinement, static typing

The work of Jean Privat was started at Montpellier University then continued at Purdue University. Authors' address: R. Ducournau and F. Morandat, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France; J. Privat, Département d'Informatique, UQAM, 210, avenue du Président-Kennedy, Montréal, QC H2X 3Y7, Canada.

February 12, 2008, RR LIRMM 07-021

1. INTRODUCTION

Modularity and reusability are key requirements in the field of software engineering. Object-oriented languages are generally understood as meeting these requirements by providing two original features, namely the notion of *class* and the *inheritance* mechanism. A class is both a data structure and a code unit. Classes are organized in a *specialization* relationship which supports a powerful sharing mechanism, i.e. *inheritance*—the newly defined subclass inherits all features defined in its superclasses. Hence, in the most common understanding of object-oriented languages, classes are modules—i.e. units of code which serve for both information hiding and reuse—and inheritance is the main support to reusability and extendability. Furthermore, from the very beginning of object-oriented programming, with the SIMULA language [Birtwistle et al. 1973], this role of modules played by classes is reinforced, in many languages, by class nesting.

However, this view of object-oriented languages is also criticized and some authors have claimed that classes do not make suitable modules because classes and modules have distinct roles—modules structure programs while classes describe and create objects [Szyperski 1992]. From a more technical standpoint, a class is appropriate for information hiding if it is a simple abstract data type such as a stack. If several classes collaborate closely to implement a function, these classes are no longer appropriate as units of information hiding [Ichisugi and Tanaka 2002]. Accordingly, classes are considered as modules of too small scale [Smaragdakis and Batory 2002]. However, on the contrary, a class, or a set of classes, often involves several functions, or concerns, which are reputed to *cross-cut*—this makes classes also inappropriate as reuse units. The so-called *expression problem* [Findler and Flatt 1999; Torgersen 1994; Zenger and Odersky 2004] stresses the limitation of this traditional view on object-oriented programming. A class or a set of classes express a set of concerns. Object-oriented languages make it easy to add new classes or to add new concerns to new subclasses. However, it is usually impossible to add new concerns to the original set of classes. Altogether, there is a need for notions that would provide larger-scale modularity, by grouping collaborative classes and separating cross-cutting concerns, while improving reusability and extendability.

Accordingly, recent research has yielded a lot of proposals addressing these commonly agreed needs. *Collaboration-based design* [VanHilst and Notkin 1996] and *family polymorphism* [Ernst 2001] have been proposed for grouping closely collaborating classes. Many other proposals are based on variations on modules or class nesting, e.g. *virtual classes* [Madsen and Møller-Pedersen 1989], *mixin layers* [Smaragdakis and Batory 1998; 2002], *open classes* [Clifton et al. 2000], *difference-based modules* [Ichisugi and Tanaka 2002], *classboxes* [Bergel et al. 2003], *higher-order hierarchies* [Ernst 2003], *nested inheritance* [Nystrom et al. 2004] and *multiple nested inheritance* [Nystrom et al. 2006]. Moreover, an entire research field has emerged, *aspect-oriented programming* (AOP), dedicated to separation of cross-cutting concerns [Kiczales et al. 1997]. All of these proposals are very similar but yet significantly different, so comparing them is not easy [Bergel et al. 2006] and deciding which is the best is a challenge, if at all meaningful.

In this paper, we focus on two coupled notions of *class refinement*—a class can be extended by adding/redefining methods and attributes and adding superclasses while retaining its name—and *module hierarchy*—a module can refine[1] classes imported from its super-modules. The point is that the class refinement is deduced from the module hierarchy and that classes and modules are strictly distinguished—so, there is no nesting of classes or modules. One can see the proposed approach either as an incremental class definition or as inheritance between class hierarchies. Altogether, modules and class refinement provide the required modularity and reusability and pretty well address the expression problem.

It follows from class refinement that the inheritance mechanism is now supported by both specialization and refinement. Hence, 'multiple inheritance' is ubiquitous in all of these approaches since it always appears when class specialization and module dependence are combined, even if both are single. Therefore, it is futile to try to avoid it and our proposal stresses both multiple specialization[2] in class hierarchies and multiple dependence in the module hierarchy. On the contrary, most of the aforementioned works are based on single inheritance class hierarchies or on a degraded form of multiple inheritance, e.g. JAVA interface multiple subtyping, *mixins* [Bracha and Cook 1990] or *traits* [Ducasse et al. 2005].

However, the goal of this paper is not only to propose these new notions—they are actually close to some previous propositions, though somewhat different—but to base this proposal on very simple and intuitive notions that already underlie object-oriented languages. *Meta-modeling* is the key of our approach and an open sesame to the selection of a precise specification among a large state space. Our fundamental argument is that *things* are not *names* and metamodeling allows us to get rid of names and their associated ambiguities, while usual considerations on scope and extent are unable to simply take the specificity of object-oriented programming into account. So we first propose a metamodel of *classes* and *properties*[3], which provides clear insight into *inheritance*, especially when *multiple*, and *redefinition*. This first metamodel provides an ontology of all related entities—what is a property? what is common to a property and its redefinition in a subclass? and conversely, what is a name? So, the metamodel precisely specifies the meaning of basic operations on properties, namely *definition*, *introduction*, *inheritance* and *redefinition*.

The keystone of our proposal is that modules and module dependence are specified by a second metamodel which is isomorphic to the first one. Module dependence hierarchies are analogues of class specialization hierarchies, classes in modules are analogous to properties in classes, and class refinement is analogous to property redefinition. This very nice property yields a rather simple and intuitive formal

[1] In this context, 'refinement' is preferred over 'extension' to avoid confusion with a common usage of the term 'class extension' for denoting class specialization, as with the JAVA `extends` keyword. See also [Smaragdakis and Batory 2002, note 1] and [Bergel et al. 2005, note 1].

[2] In this paper, we carefully distinguish the specialization relationship, which supports inheritance, from inheritance itself. Therefore, the terms 'single' or 'multiple inheritance', according to whether a class can have more than one superclass, are improper—one should speak of single or multiple specialization—but this metonymical usage is now so common that we cannot escape it.

[3] The object-oriented terminology is quite variable, according to authors and languages. We use the term 'property' with the same meaning as 'feature' in EIFFEL or 'member' in C++. Hence, it denotes both *methods* (aka 'routine', 'function member'), *attributes* (aka 'instance variable', 'field', 'slot', 'data member') and also types in the sense of *virtual types*.

definition of our central couple of notions. Furthermore, this proposal is not only based on conceptual and theoretical considerations. Modules and class refinement are distinctive features of a new object-oriented language, PRM[4] [Privat 2006b], and they are intensively used in the organization of its libraries and tools. The modular architecture of its bootstrapped compiler PRMc has been designed to use as a testbed for evaluating various object-oriented implementation techniques [Privat and Ducournau 2005], thus making it easy to replace a module by another one, offering the same functionality with a different implementation.

Finally, in order to precisely fix our framework, this paper focuses on static typing languages. Besides the fact that static typing fully meets software engineering requirements, the point with modules and refinement is to define static semantics, which must be separately established for each module. On the contrary, dynamic typing is generally less compatible with static analysis. Moreover, this is also a technical constraint yielded by the metamodel—dynamic typing eliminates its deciding effect on multiple inheritance. Dynamic loading is beyond the scope of this paper, the adaptation of modules and class refinement to such a dynamic framework remains to be investigated.

So, to conclude this introduction, the novelty of the present paper is threefold: (i) a metamodel of classes and properties which clarifies the basic notions of object-oriented programming and yields a sound analysis of multiple inheritance and its associated conflicts; (ii) the specification of class refinement and module dependence hierarchy in an isomorphic metamodel where multiple dependence between modules is analogous to multiple class specialization, and multiple class refinement and combination of refinement and specialization are handled like multiple class specialization; and (iii) the application of the previous two points to a new language, PRM, and to the architecture of its bootstrapped compiler.

Paper organization and plan

The article is organized as follows. Section 2 provides an intuitive presentation, on a simple example, of modules and class refinement. The next three sections are the core of the proposal. Section 3 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 4 uses the metamodel to shed light on the somewhat well-known difficulties of multiple inheritance, which involve the two distinct inheritance levels—and the associated conflicts—which were previously identified in [Ducournau et al. 1995]. The specifications of the most commonly used object-oriented languages are compared with the metamodel and some alternatives to multiple inheritance, e.g. mixins or traits, are examined. Section 5 formalizes modules and module dependence in a metamodel which is structurally isomorphic to that of classes and properties. In this framework, class refinement appears as an exact analogue of property redefinition. Section 6 presents a case study, by describing the PRM language and the modular architecture of its bootstrapped compiler, PRMc. The implementation of

[4] PRM stands for *Programming with Refinement and Modules*, which of course implies *object-oriented* Programming with *class* Refinement and Modules. More information is available on the PRM homepage: `http://www.lirmm.fr/prm`.

our proposal is outlined—a nice feature is that it does not entail any overhead at run time. Section 7 surveys various related works. Finally, the last section concludes the paper by presenting the current perspectives together with the known limitations of our proposal.

2. MODULES AN CLASS REFINEMENT: PRINCIPLE AND EXAMPLE

2.1 Principle

One can present the refinement of a class c_1 by a class c_2 as an incremental class definition in which the properties (attributes and methods) defined in c_2 are added or take the place of those of c_1. Of course, this sentence also applies to specialization but, in contrast with specialization, once c_1 has been refined by c_2, the latter takes the place of the former in all of its occurrences throughout the program, i.e. all instances of c_1 will be instances of c_2. In practice, this means that c_1 and c_2 have the same name and, once c_2 has been defined, throughout the program, this name denotes c_2. Intuitively, this is the only difference between refinement and specialization. Such a mechanism is relatively common in languages with dynamic typing. In object-oriented extensions of Lisp, like Clos [DeMichiel and Gabriel 1987; Steele 1990], this is mainly reserved to methods as they are defined across and outside classes. In another Lisp-based object-oriented language, Yafool [Ducournau 1991], incremental class definition was commonly used and object-oriented modules were coupled with functional modules, à la Modula, of the underlying Lisp dialect, namely LeLisp [Chailloux 1993]. All languages supporting a *meta-object protocol* [Kiczales et al. 1991] should also be able to support refinement, even though experiments show that these protocols are not always adapted to dynamic class modifications.

More recently, different proposals have considered similar notions in a static typing framework. Now, *class refinement* is coupled with a notion of *module*, in its simplest and quite traditional form, which makes static analysis relevant. These proposals are based on various extensions of the object model called mixin layers [Smaragdakis and Batory 1998; 2002], difference-based modules [Ichisugi and Tanaka 2002], higher-order hierarchies [Ernst 2003], nested inheritance [Nystrom et al. 2004; Nystrom et al. 2006] or class boxes [Bergel et al. 2005]. Our proposal is within the scope of these approaches and our definition of these notions—similar to these different proposals but not strictly equivalent—is the following.

A *module* is a class hierarchy, i.e. a set of classes ordered by specialization. It is a reuse unit [Szyperski 1992] which can be separately compiled and then linked to other modules to produce a final executable. On the contrary, classes are no longer reuse units and there is no class nesting. A module *depends on* a set of other modules (called *supermodules*) and can *refine* classes *imported* from them. The dependence relation is acyclic, like class specialization. Our usage of 'submodule' and 'supermodule' is a pure matter of convenience, by analogy with 'subclass' and 'superclass'. In any case, a submodule is definitely *not* included in its supermodule—there is no module nesting either. The class refinement order is then deduced from the dependence order between the modules. Finally, we do not associate a notion of visibility (or export) with modules, because this is not critical to the issue dealt with here. Of course, this would be of practical use and a natural extension, in the

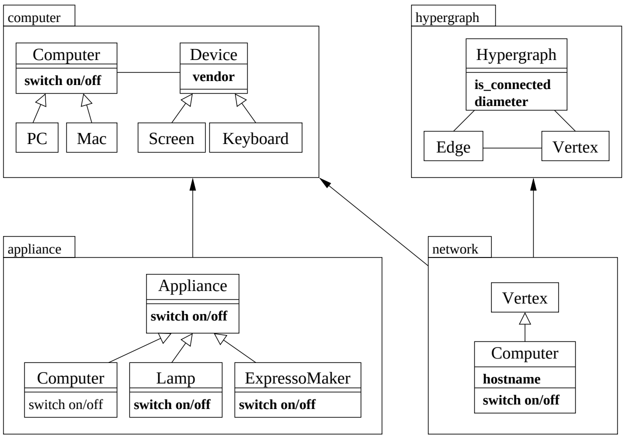


Fig. 1. Concrete Example with Modules and Class Refinement. Each module is depicted as a boxed UML-like model. Both class specialization and module dependence are explicit. In contrast, class refinement is implied by module dependence, e.g. `Computer` in module `appliance` refines `Computer` in module `computer`.

tradition of Modula-2 and -3 [Harbinson 1992]. However, the interaction between refinement and visibility may not be straightforward and we prefer to postpone this point for further work. As a first approximation, the reader may consider that refinement only concerns visible, i.e. exported, entities.

Regarding class refinement, we identify four atomic mechanisms:

(1) *adding a property*, i.e. the definition of a *newly introduced* method or attribute;

(2) *redefining* (aka *overriding*) *a property*—this is more common for methods than for attributes;

(3) *adding a superclass*, since we consider languages with multiple inheritance;

(4) *generalizing a property*, i.e. defining a property in superclasses of the class which introduced it in the supermodules;

The latter presents another difference with specialization which just involves the first three points.

A fifth mechanism might also be considered—namely *unifying classes*, i.e. asserting that two classes defined in different modules are the same by merging their definitions. However, in the present paper, we shall only examine the first four mechanisms, leaving class unification for further research.

Like specialization and inheritance, refinement is quite intuitive when the different refinements are totally ordered—in an analogue of single inheritance. However, it is not obvious how to extend this intuition to multiple refinement of one class or to a combination of refinement and specialization, since their ordering does matter. While dynamic languages can use the chronological ordering of class definitions, at the risk of making the semantics depend on the control flow of each execution, this is not possible when class refinement is computed at compile-time.

2.2 Example

This section presents an example of four modules (Fig. 1) to illustrate a concrete application of the class refinement mechanism. The example is written in Prm [Privat 2006b]. The syntax has a simple straightforward style: class definitions start with the `class` keyword and property definitions (attributes and methods) start

-a Computer Module

```
class Computer
  def switch_on_off ...
end
class PC
  inherit Computer
  ...
end
```

-b Appliance Module

```
import computer
class Appliance
  def switch_on_off ...
end
class Computer
  inherit Appliance
end
class Lamp
  inherit Appliance
  def switch_on_off ...
end
class ExpressoMaker
  inherit Appliance
  def switch_on_off ...
end
```

-c Hypergraph Module

```
class Hypergraph
  def @nodes: Array[Vertex]
  def @edges: Array[Edge]
  def is_connected ...
  def diameter ...
end
class Vertex
  def @edges: Array[Edge]
  ...
end
class Edge
  def @nodes: Array[Vertex]
  ...
end
```

-d Network Module

```
import hypergraph
import computer
class Computer
  inherit Vertex
  def @hostname: String
  def switch_on_off ...
end
```

Fig. 2. Module Implementations of Figure 1

with the `def` keyword. Attribute names start with a `@` character. As modules are reuse units, each Prm source file is a module.

Computer Module. The first module models the workings of a computer (Fig. 2-a). It defines a `Computer` class, subclasses for different computer types and other classes for devices.

Appliance Module. A programmer wants to generalize computer behavior with other appliances like lamps or expresso makers without changing the `computer` module. A new abstract class, `Appliance`, is created as an ancestor of `Computer` (*adding a superclass*), and is specialized by two classes: `Lamp` and `ExpressoMaker`. Appliances use electricity, and can be powered on and off: the `switch on/off` method defined in the `Computer` class must be generalized in the `Appliance` class (*property generalization*), and then specialized in both `Lamp` and `ExpressoMaker` subclasses. The `import` keyword[5] at the first line of the implementation (Fig. 2-b) declares that the `computer` module is a supermodule. Hence, classes from the `computer` module are imported and the `Computer` class is refined.

Hypergraph Module. This module models finite hypergraphs, i.e. a generalization of graphs in which edges contain any number of vertices (Fig. 2-c). Methods of the `Hypergraph` class implement two algorithms: `is_connected` returning `true` iff there is a path between any couple of vertices and `diameter` computing the maximal distance between two vertices of the graph.

[5] The `import` keyword may seem improper for expressing the module dependence relationship, but it is coherent with the metonymical use of the `inherit` keyword and common object-oriented terminology (see also Note 2, page 3).

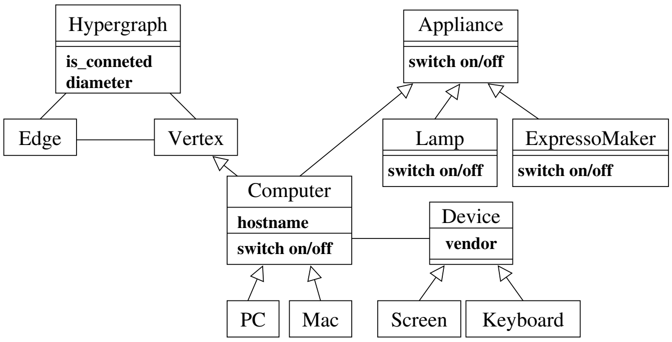


Fig. 3. Final Class Model of the Program

Network Module. Both computer and hypergraph modules are independent and might be developed for different programs by different programmers. Class refinement allows programmers to easily create a derived module and to model networks—Fig. 2-d. On a network, every computer has an hostname. Hence, a new attribute, `hostname`, is added to the `Computer` class (*property addition*). Moreover, complex networks, e.g. Internet, can be represented by hypergraphs with computers as vertices. For this reason, a specialization relation is added between `Vertex` class and `Computer` class (*adding a superclass*). When a computer is powered on, it becomes reachable on the network and unreachable when it is powered off. Hence, in order to characterize the computer state on the network, the `switch on/off` method is redefined (*method redefinition*). All of these modifications made on the `Computer` class are naturally inherited by PCs and Macs.

Final Program. For programs which merge these four modules, each class is imported (Fig. 3) and the final `Computer` class consists of the combination of the `Computer` class declarations from the above modules as follows: `Computer` is a superclass of `PC` and `Mac` (from `computer`), it is also a subclass of `Vertex` (from `network`) and of `Appliance` (from `appliance`). It has a `switch on/off` method (the one declared in `network` overrides that from `computer`) and a `hostname` attribute (from `network`).

3. CLASS AND PROPERTY METAMODEL

In this section, we present a metamodel for classes and properties in object-oriented languages. Here, we only consider properties which are described in a class but dedicated to its instances, and which depend on their dynamic types. Class properties, i.e. properties which only concern the class itself, not its instances, are excluded from the scope of the metamodel. Hence, we do not consider either *static* methods and variables, the fact that a class may be *abstract* (aka *deferred*), or properties which would be called *non-virtual* in C++ jargon—we only consider properties which are concerned by *late binding*, hence tagged by the `virtual` keyword in C++ when considering methods[6]. In CLOS, however, we would also consider slots

[6] 'Virtual' comes from SIMULA and has several usages—virtual functions (SIMULA, C++), virtual types (SIMULA, BETA [Madsen and Møller-Pedersen 1989; Madsen et al. 1993]) and virtual mul-

declared with `:allocation :class`.

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. It can be considered as an implicit metamodel of JAVA and EIFFEL—in the latter case, with a limited use of renaming—but it has never been explicitly described in any programming language nor even in UML [OMG 2004].

In the following, we successively present 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, and the comparison with existing languages will be tackled in Section 4.

3.1 The UML model

3.1.1 *Object-oriented meta-modeling of object-oriented languages.* In an object-oriented setting, metamodeling 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[7] [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[8]. 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 3.1 (CONTEXT-DEPENDENT UNAMBIGUITY). In the modeled program, any occurrence of an identifier denoting a modeled entity must unambiguously map to a single instance of the metamodel. Of course, this mapping is context-dependent.

Accordingly, metamodeling 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.

tiple 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 [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 'virtual'—they are different enough to require different terms and we shall only consider the former usage.

[7] 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].

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

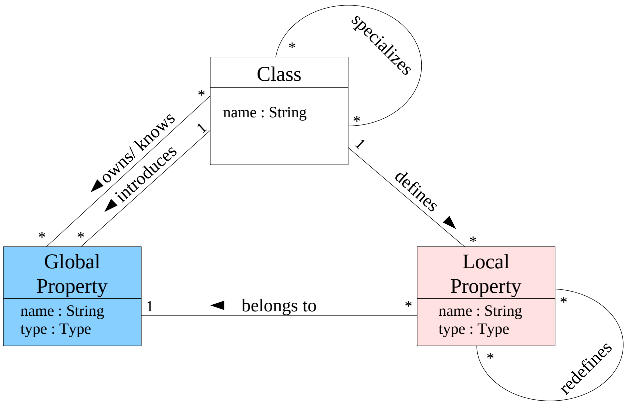


Fig. 4. Metamodel of Classes and Properties

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.

3.1.2 *Classes and properties.* The metamodel consists of three main kinds of entities: *classes*, *local properties* and *global properties*—Fig. 4. The former is of course natural, but the two latter are key features of the model. When one wants to metamodel properties, it follows from Requirement 3.1 that late binding (aka message sending) implies the definition of exactly two categories of entities. *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. *Global properties* are intended to model this 'same property' idea 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 global property and is *defined* in a single class.

Global and local properties should 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]. Anyway, in the following, 'property' stands for all three kinds and a partial but more intuitive translation of our terminology is possible—global (resp. local) properties stand for methods (resp. method implementations). Furthermore, in this section, the specific kind of property does not matter.

A class definition is a triplet consisting of the name of the class, the name of its superclasses, presumably already defined, and a set of local property definitions. The specialization relation supports an inheritance mechanism—i.e. classes inherit the properties of their superclasses. When translated in terms of the metamodel, this yields two-level inheritance. First of all, the new class *inherits* from its superclasses all their global properties—this is *global property inheritance.* The subclass *knows* all the global properties *known* by the superclasses. Then each local property definition is processed. If the name[9] of the local property is the same as that of an inherited global property, the new local property is attached to the global property. If there is no such inherited global property, a new global property with the same name is *introduced* in the class. We do not consider here the questionable possibility of introducing a new global property with the same name as an inherited one, as with the `reintroduce` keyword in TURBO PASCAL.

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 *global 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 global properties with the same name while satisfying Requirement 3.1. Static overloading can also slightly complicate the point. If the name '`foo`' is overloaded in A, i.e. if A knows several global 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 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 making actual parameter types more precise, i.e. paradoxically more general, at the call site. Anyway, at run-time, this call site is interpreted as the invocation of the **single** *local property* corresponding to both the **single** statically selected *global 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 global property must be inherited from superclasses. Static typing ensures that such a local property exists.

3.1.3 *Example.* The JAVA example in Figure 5 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 global properties, respectively introduced as `foo` in `A` and as `bar` in `B`. The corresponding numbered instance di-

[9] The 'name' of properties is a convenient shorthand for more complex identifiers. Static overloading of C++, JAVA, C#, etc. forces, in these languages, to include the parameter types in the identifier in order to unambiguously denote properties. This implies distinguishing *short names* and *long names.* In the present framework, static overloading is formally defined by the fact that a class knows two global properties with the same short name but different long names.

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

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

   ...

   {...
    A x
    B y
4   x.foo()
5   y.bar()
    ...}
```

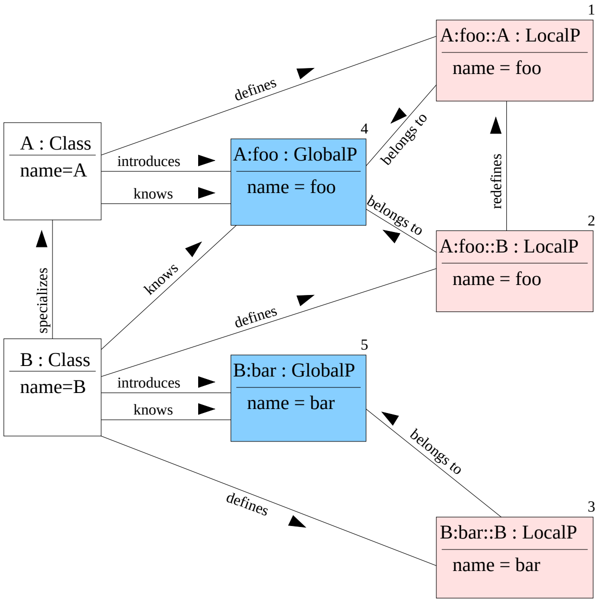


Fig. 5. A simple JAVA example and the corresponding instance diagram

agram specifies the unambiguous mapping between the code and the instances of the metamodel (Req. 3.1).

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

—A class is first created; it does not inherit any explicit global property—in practice, it would inherit all global 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 global property with this name, a global property (4) `foo` is introduced;

—B class is then created as a specialization of `A`; it then inherits all explicit global properties from `A`, especially the global property `foo`;

—a method named `foo` is defined in `B`: the corresponding local property (2) is created and attached to the global 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 global property named `bar`, then a global property (5) `bar` is introduced;

—in the following code sequence, `foo` (resp. `bar`) is understood as the single global 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 global 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.

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 get rid of names and their associated ambiguities and instead just consider reified entities. Actually, these reified entities represent the 'reality'—the *ontology*—of object-oriented programs and names should remain at the human-computer interface.

3.2 Notations and 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, i.e. the incremental process of class definition, and (ii) for late binding.

Notations. Let E, F and G be sets. 2^E denotes the power set of E, $|E|$ is the size 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.

Definition 3.2 (Class hierarchy). A *model of a hierarchy*, i.e. an instance of the metamodel, is a tuple $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, name_{\mathcal{H}}, glob_{\mathcal{H}}, intro_{\mathcal{H}}, def_{\mathcal{H}}\rangle$, where:

—$X^{\mathcal{H}}$ is the set of *classes*;

—$\prec^{\mathcal{H}}$ is the *class specialization* relationship, which is transitive, antisymmetric and antireflexive; $\preceq^{\mathcal{H}}$ (resp. $\prec_d^{\mathcal{H}}$) 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 *global* 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;

—$glob_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ associates with each local property a global property;

—$intro_{\mathcal{H}} : G^{\mathcal{H}} \to X^{\mathcal{H}}$ associates with a global 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 (1–9).

The sets $X^{\mathcal{H}}$, $G^{\mathcal{H}}$ and $L^{\mathcal{H}}$ correspond to the three classes in the metamodel, whereas the total functions $glob_{\mathcal{H}}$, $intro_{\mathcal{H}}$ and $def_{\mathcal{H}}$ correspond to the three functional associations—all six elements form the metamodel 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 4. The definition of a *legal model* is achieved by a set of constraints which ensure that: (i) the triangular diagram commutes 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 Sections 3 and 4, parameter $\mathcal{H}$ will remain implicit for the sake of readability. The reader must keep in mind that all components of the model $\mathcal{H}$ are relative to it and that the parameter must be explicit when one deals with more than one model, as in Section 5.2. All proofs are trivial and left to the reader.

3.2.1 *Global Properties.* Given a class $c \in X$, G_c denotes the set of global properties of c. Global 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

$$G_{+c} \overset{\text{def}}{=} \mathit{intro}^{-1}(c) \ , \tag{1}$$

$$G_{\uparrow c} \overset{\text{def}}{=} \bigcup_{c \prec_{\mathrm{d}} c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'} \ , \tag{2}$$

$$G_c \overset{\text{def}}{=} G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \ , \tag{3}$$

$$G = \bigcup_{c \in C} G_c = \biguplus_{c \in C} G_{+c} \ . \tag{4}$$

The definitions and equations (1-3) formally define *global property inheritance.*

Names of newly introduced global properties are constrained:

Constraint 3.3. When a global property is introduced, its name must be unambiguous, hence (i) the restriction of *name* on G_{+c} is injective and (ii) inherited and introduced properties cannot have the same name:

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

Note that this constraint is actually implied by the constraints concerning local properties (Section 3.2.2). It follows that the function $\mathit{gid} : G \to X \times N$ that maps a global property $g \in G$ to the pair

$$\mathit{gid}(g) \overset{\text{def}}{=} (\mathit{intro}(g), \mathit{name}(g)) \tag{5}$$

is injective. Besides this constraint, all $\langle X, \prec, G, \mathit{name}, \mathit{intro}\rangle$ tuples that follow Definition 3.2 are legal.

There is a global property conflict when a class knows two distinct global properties with the same name. Constraint 3.3 implies that such a conflict is always caused by multiple inheritance:

Definition 3.4 (Global property conflict). Given a class $c \in X$ and two distinct global properties $g_1, g_2 \in G_{\uparrow c}$, a *global property conflict* occurs between g_1 and g_2 when

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

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 global 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 3.3 implies that the restriction of *name* to G_c is also injective. Therefore, in the context of a class c, the identifier of a global 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. Multiple inheritance conflicts will be examined in Section 4.

3.2.2 *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:

$$L = \biguplus_{c \in C} L_c \qquad \text{with} \qquad L_c \overset{\text{def}}{=} def^{-1}(c) \ . \tag{6}$$

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

Constraint 3.5. The function $glob : L \to G$ associates with each local property a global property, such that both have the same name:

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

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

Constraint 3.6. For all $c \in X$, the restriction of *glob* to L_c is *injective.* Equivalently, for all $g \in G$, the restriction of *def* to $glob^{-1}(g)$ is injective.

Therefore, if there is no global property conflict, the restriction of *name* to L_c is also injective. Thus determining the global property associated with a local one is unambiguous in the context of some class c, at least when the name of the global property is unambiguous, i.e. when there is no global property conflict:

$$\forall l \in L_c, \forall g \in G_c, name(l) = name(g) \Longrightarrow glob(l) = g \ . \tag{7}$$

Of course, *glob* 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 triangle and achieves the definition of a *legal model*:

Constraint 3.7. The associated global properties must be known by the considered class and all global properties must have been introduced by a local property definition:

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

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 3.8 (Legal model). A *legal model* of a *class hierarchy* is a model which satisfies all Constraints 3.3 to 3.7.

It follows from Constraint 3.6 that, in a legal model, the function $\mathit{lid} : L \to G \times X$ that maps a local property $l \in L$ to the pair

$$\mathit{lid}(l) \overset{\text{def}}{=} (\mathit{glob}(l), \mathit{def}(l)) \tag{8}$$

is injective.

Finally, one can formally define *property redefinition.* A local property corresponds to an inherited global property, or to the introduction of a new global property. Let $L_{\uparrow c}$ and L_{+c} be the corresponding sets.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{with} \begin{cases} L_{\uparrow c} & \overset{\text{def}}{=} L_c \cap \mathit{glob}^{-1}(G_{\uparrow c}) \ , \\ L_{+c} & \overset{\text{def}}{=} L_c \cap \mathit{glob}^{-1}(G_{+c}) \ . \end{cases} \tag{9}$$

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

Definition 3.9 (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:

$$l \ll l' \overset{\text{def}}{\iff} \mathit{glob}(l) = \mathit{glob}(l') \wedge \mathit{def}(l) \prec \mathit{def}(l') \ .$$

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

3.2.3 *Class Definition and Model Construction.* The model is built by successive class definitions.

Definition 3.10 (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. including parameter types. 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 $(c, \mathit{name}^{-1}(n))$ 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 is 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 $\prec_{\mathrm{d}}$[10].

(ii) *Global property inheritance.* The protocol then proceeds to global property inheritance. $G_{\uparrow c}$ is computed (2) and global property conflicts are checked (Def. 3.4).

[10] In both cases, this is contrary to C++ and Eiffel behavior.

(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 (9). Then, G_{+c} is constituted as the set of *new* global 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 global property conflicts are discussed in Section 4.1.

(iv) *Local property inheritance.* Finally, the protocol proceeds to local property inheritance and checks conflicts for all inherited and not redefined properties, i.e. $G_{\uparrow c} \backslash glob(L_{\uparrow c})$. Conflicts are discussed in Section 4.2.

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 3.2, together with Constraints 3.3 to 3.7 and equations (1–9), are sufficient to characterize a legal model as long as there is no global property conflict. All such legal models could be generated by Definition 3.10—given a legal class hierarchy $\mathcal{H}$, for all c in X, $\langle name(c), \{name(c') \mid c \prec_d c'\}, name(L_c)\rangle$ form a legal class definition.

3.3 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. 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[11].

Local property inheritance is a matter of selection of a single local property in a given global property, according to the dynamic type of an object called the receiver. Though it applies only to methods in most languages, it can also be used for attributes—and it is indeed used in some languages like Clos, Yafool or Eiffel. We only detail the case concerning methods. Method invocation usually involves two distinct mechanisms, namely late binding (aka message sending) and

[11] 'Constructor' is another 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 global 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.

call to `super`, and 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 $g \in G_c$.

3.3.1 *Late binding.* Given a class $c \in C$—the dynamic type of the receiver—and a global 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

$$loc(g,c) \stackrel{\text{def}}{=} \{l \in glob^{-1}(g) \mid c \preceq def(l)\} \ . \tag{10}$$

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

$$spec(g,c) \stackrel{\text{def}}{=} \min_{\ll}(loc(g,c)) \ , \tag{11}$$

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

Definition 3.11 (Local property conflict). Given a class $c \in X$ and a global property $g \in G_{\uparrow c}$, a *local property conflict* occurs when $|spec(g,c)| > 1$. The *conflict set* is defined as the set

$$cs(g,c) \stackrel{\text{def}}{=} def^{-1}(spec(g,c)) = \min_{\preceq}(def^{-1}(loc(g,c)))$$

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

We shall examine, in Section 4.2, the case of multiple inheritance, local property conflicts and the reasons for this definition.

3.3.2 *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

$$supl(l) \stackrel{\text{def}}{=} \{l'' \mid l \ll l''\} \ . \tag{12}$$

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 global property, i.e. $glob(l) = glob(l')$, while Smalltalk and Java accept `super.foo` in method `bar`. 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 alternatively determined by the uniqueness of c' such that $def(l) \prec_d c'$—then $l' = sel(glob(l), c')$.

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 the same problems in case of multiple inheritance. We shall examine them in Section 4.2.

4. MULTIPLE INHERITANCE

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.

4.1 Global Property Conflict

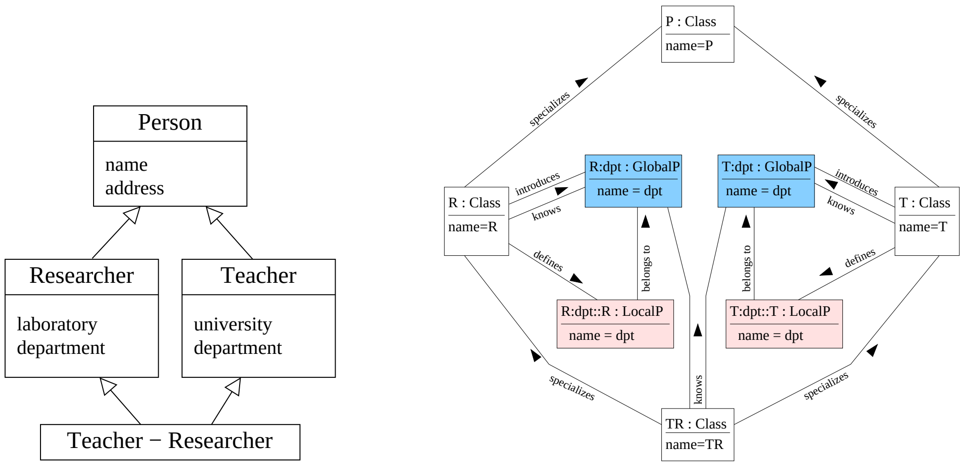
A *global property conflict* (in [Ducournau et al. 1995], it was called 'name conflict') occurs when a class specializes two classes having distinct but homonymic global properties (Def. 3.4). Figure 6 shows two global properties named `department`. The first one specifies a department in a research 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 global 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:

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 global property. In the example, `Teacher:department` would denote the global property known as `department` in the class `Teacher`. Such a naming would be unambiguous since, in a legal model, *gid* is injective (5). A similar solution is used for attributes in C++ with the operator :: [Stroustrup 2000][12]. Note that, with fully qualified names, a global 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 global 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 `dept-teach` and `department` inherited from `Researcher` as `res-dept`. Thus, `department` denotes,

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



-a Class diagram

-b Instance diagram

Fig. 6. Global 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.

in `Researcher`, the same global property as `res-dept` in `Teacher-Researcher`; conversely, in the class `Teacher-Researcher`, `res-dept` and `teach-dept` denote two distinct global 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, 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[13]), consider that if two global properties have the same name then they are not distinct. Java has the same behaviour when a class implements two interfaces which introduce a global property with the same name and signature[14]. Hence, the global 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 global properties, unless there is global renaming. In Figure 6, 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 global property for this concept. However, silence adds an extra flaw—i.e. the programmer may remain unaware of the problem or might misunderstand it.

As a provisional conclusion, global property conflicts represent a very superficial problem. They could easily be solved in any programming language at the expense

[13] This dissymmetry between methods and attributes represents one of the most striking flaws of the language specifications.
[14] Java provides a fully qualified syntax for classes, not for properties.

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 namespaces are explicit, e.g. packages in JAVA or COMMON LISP. 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 3.10.

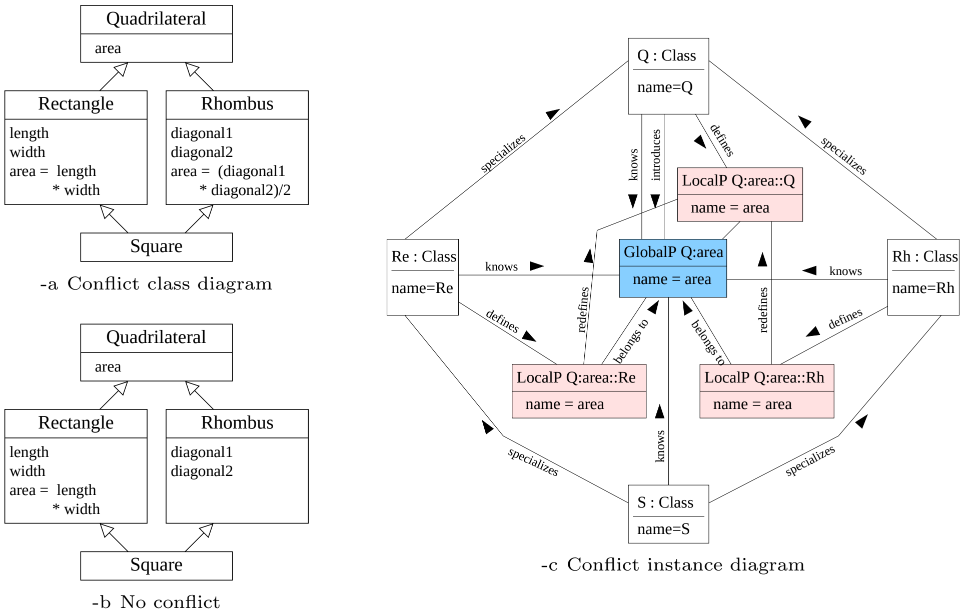
On the contrary, when languages deal with global property conflicts by signaling an error or unifying conflicting properties, the situation may have no solution. A global renaming would be the only way out, but it might be impossible if the source code of the conflicting classes is not available, or if the conflicting classes are contractually used by other people.

4.2 Local Property Conflict

4.2.1 *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 global property, with none of them more specific than the other according to $\ll$ (Def. 3.11). Figure 7-a illustrates this situation with two classes, `Rectangle` and `Rhombus`, redefining the method `area` whose global property was introduced into a common superclass, `Quadrilateral`. In the common subclass `Square`, none is most specific—which one must be selected? Figure 7-c depicts the model of the example, restricted to the concerned property. Let g be the `area` global property introduced in `Quadrilateral`—`Q:area` for short—and c be the `Square` class, abbreviated in `S` in the diagram. Then,

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

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. 7-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 for instance [Rayside and Campbell 2000b; 2000a; Rayside and Kontogiannis 2001; Ducournau 2002b]. On the contrary, property redefinition entails *non-monotonicity*, in the sense of *nonmonotonic* (aka *defeasible*) inheritance theories, e.g. [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*:



-a Conflict class diagram

-b No conflict

-c Conflict instance diagram

Fig. 7. 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).

Requirement 4.1 (Masking rule). Let g be a global 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 $l' \ll l$ redefines l in A', l' will *mask* l for all instances of A', direct and indirect alike.

So in the considered example of Figure 7-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].

4.2.2 *Conflict solutions.* Unlike the global 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[15] (`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 4.3); 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*[16]—the conflict resolution must be done 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 4.4. 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 to the selected one. This static call is done with the :: operator. It is always possible in a legal model since the *lid* function (8) is injective. To be compatible with the syntax for global properties, we shall use a slightly different syntax here—e.g. `area`::`Rhombus`[17] denotes the local property `area` defined in the class `Rhombus` (Fig. 7). If the global property name is ambiguous, full qualification can lead to `Quadrilateral:area`::`Rhombus`.

4.2.3 *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 global property and $c \in X$ be the receiver's dynamic type. This means that $\mathit{spec}(g, c) = \{l\}$ and $\mathit{loc}(g, c)\backslash\{l\} = \mathit{supl}(l)$—see definitions in equations (10-12). In other words, all other local properties that might be combined are in $\mathit{supl}(l)$. Therefore, the situation is exactly the same as late binding, except that the selection or combination process must now consider $\mathit{supl}(l)$ instead of $\mathit{spec}(g, c)$. Then the set $\min_{\ll}(\mathit{supl}(l)) = \{l' \mid l \ll_d l'\}$ 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. The solution is to consider that `super` is legal only when $\min_{\ll}(\mathit{supl}(l))$ is a singleton. This is the approach of EIFFEL, with

[15] When considering *abstract local properties*, conflict definition must be slightly adapted. An actual conflict occurs when $\mathit{spec}(g, c)$ contains several non-abstract properties. If all members of $\mathit{spec}(g, c)$ are abstract, the inherited property in c is also abstract.

[16] Local properties are objects, hence 'composed' of some meta-properties. So solving the conflict always amounts to combining the conflicting objects and, for each meta-property, selecting or combining. This is for instance quite explicit in the CLOS slot-description specifications.

[17] Instead of `Rhombus`::`area` in C++.

`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 7—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 $\mathit{spec}(g, c)$, the point is that $\mathit{supl}(l)$ is far from including all other local properties in $\mathit{loc}(g, c)$—actually $\mathit{spec}(g, c) \backslash \{l\}$ and $\mathit{supl}(l)$ 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 $\mathit{loc}(g, c)$. So, `call-next-method` avoids multiple evaluation risks, but linearizations also have their own drawbacks (Section 4.3). Of course, 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:

(1) `call-next-method` and linearizations, discussed hereafter;

(2) a constrained keyword `super`, which behaves like `Precursor` in EIFFEL, i.e. only sound when there is no conflict;

(3) static call, with a fully qualified syntax such as `foo::C`;

(4) 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.

4.3 Linearizations

Linearizations have been widely used in dynamically typed object-oriented languages such as FLAVORS [Moon 1986], 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 and SCALA (see Section 4.5.2).

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

$$supc(c) \overset{\text{def}}{=} \{c' \mid c \preceq c'\}. \tag{13}$$

This ordering is called *class precedence list* in CLOS.

Definition 4.2 (CLASS LINEARIZATION). Given a class hierarchy $\mathcal{H}$, a *class linearization* is defined as a function $clin^{\mathcal{H}} : X \rightarrow (X \rightarrow \mathbf{N})$ (*clin* for short) such that $clin(c)$ (noted hereafter $clin_c$ for the sake of readability) is a bijective function $clin_c : supc(c) \rightarrow 0..|supc(c)| - 1$ (aka a *permutation*). It yields a total order $(supc(c), \leq_{clin(c)})$, whereby $x \leq_{clin(c)} y \overset{\text{def}}{\iff} clin_c(x) \leq clin_c(y)$. 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 then be mapped from classes to local properties, i.e. from the poset $(supc(c), \preceq)$ to the poset $(loc(g, c), \ll)$, for each global property $g \in G_c$:

Definition 4.3 (LOCAL PROPERTY LINEARIZATION). Given a class hierarchy $\mathcal{H}$, equipped with a class linearization *clin*, a *local property linearization* is defined by the function $llin : G \times X \rightarrow (L \rightarrow \mathbf{N})$ 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 an analogous total order $(loc(g, c), \leq_{llin(g,c)})$.

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 4.4 (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 C++ static calls or with qualified `super`. 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[18], hence $c \preceq def(l_i)$. Contrary to single inheritance, $l \ll cnm(c, l)$ is not always verified. Here, some authors, e.g. [Snyder 1991], diagnose a drawback—this would break class modularity. This seems, however, unavoidable in method combination when inheritance is multiple.

4.3.2 *Requirements.* Some theoretical studies have determined what should be a 'good' linearization. We review here their main conclusions.

[18] 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.

Linear extensions. In order to ensure that the selection respects the *masking rule* (Req. 4.1), i.e. that no other property would be more specific—the total order must be a *linear extension*[19] [Ducournau and Habib 1987]. This means that

$$c \preceq c' \preceq c'' \implies clin_c(c') \leq clin_c(c'') \tag{14}$$

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 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:

$$c \preceq c' \preceq c'', c''' \implies \big(clin_c(c'') \leq clin_c(c''') \iff clin_{c'}(c'') \leq clin_{c'}(c''')\big) \tag{15}$$

Since $clin_c(c) = 0$, a monotonic linearization is always a linear extension.

Obviously, 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, this second requirement is not as easy to meet as the first one. Actually, given a class hierarchy $(X, \preceq)$ equipped with a monotonic linearization *clin*, and two classes $c_1, c_2 \in X$, it may 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 $\leq_{clin(c_2)}$.

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. This resembles *class unification*—the fifth mechanism outlined in Section 2.1 and further discussed in Section 5.3.2—but in Ernst's proposition, the unification only concerns the considered class which causes a linearization cycle. Of course, such merging may provoke some conflicts between local properties, but the author does not analyze them. On the

[19] Linear extensions are also called *topological sorting* [Knuth 1973].

contrary, in a subsequent paper [Ernst 2002], he seems to consider that the idea is not feasible.

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, for instance as the order of superclass declaration—hence, in Definition 3.10, `supernames` is an *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. An *extended precedence order* has also been considered [Huchard et al. 1991]—it is the third constraint underlying C3 linearization [Barrett et al. 1996; Ernst 1999] which is also used by PYTHON (for 'new-style' classes only).

4.3.3 *Prospects.* Linearizations can be further improved in several ways.

Partial linearizations. It follows from Definitions 4.2 and 4.3 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 global 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 linearizations. However, one might also allow the programmer to specify a partial linearization for each global property—there is no necessity for two global 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 global properties. This is, however, just a research issue.

Finally, linearizations can coexist with other combination techniques, e.g. the qualified call to `super` that we propose.

Linearizations are often criticized because they would be hard to understand by programmers. This could also be improved—for instance, Forman and Danforth [1999] make a pedagogical presentation of linearizations and Ernst [1999] gives a declarative definition of the C3 linearization proposed by [Barrett et al. 1996], which is far more comprehensive than previous algorithmic definitions. Moreover, development tools like ECLIPSE could easily provide precise diagnoses of non-monotonic situations and help the programmer to fix them.

4.4 Static Typing

Type safety is mostly beyond the scope of this paper. Hence, this section does not present a precise type system but only examine how different type systems could be adapted to the metamodel.

4.4.1 *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[20]. This constraint concerns property redefinition, through the well known *contravariance rule* [Cardelli 1984]. However, class specialization and static typing entail a much debated tradeoff between type safety and expressiveness. The contravariance rule is essential in a type safe policy, but strictly contravariant parameter types have turned out to be useless for modelling real-world applications[21]. On the contrary, modeling real world often requires covariant parameter types—Shang [1996], Meyer [1997], Ducournau [2002b], Büttner and Gogolla [2004] present arguments in favor of this covariant policy and EIFFEL, BETA and O_2 [Bancilhon et al. 1992] have adopted it.

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, only the hierarchy which is known represents only a small part of actual program hierarchies.

CWA corresponds to global compilation and OWA to separate compilation. CWA is similar to 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 type safe policy, most method calls would be rejected. Hence, a sensible solution is to declare whether a global property is or is not covariant when introducing it—it must be done however for each parameter whose type is not *final*. This is a sensible tradeoff between expressivity and type safety.

Subtyping vs. type safety. We do not pretend to conclude here this old debate, considering both covariance and contravariance, type safe and unsafe policies, together with CWA and OWA. So, we identify subtyping with specialization, as it is formalized in the metamodel, but distinguish 'subtyping' which refers to the chosen conformance rule from 'type safety' which is related to the possible need for runtime checks[22]. These checks are generated by the compiler but the programmer remains in charge of handling failures. We only detail hereafter the differences with usual approaches that are yielded by the metamodel.

[20] As most assertions about C++, this one presents some exceptions, e.g. *private inheritance.*

[21] For instance, Shang [1996] ends its often cited paper by "We can live without contravariance".

[22] There are also 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. However, Cardelli [2004] proposes a slightly different definition, whereby only *untrapped* errors are rejected. We use 'type safe' is the former sense but our approach (see Note 24) is type safe in the latter sense.

In the method case, the type of a local property l, noted $\mathit{ltype}(l)$, is as usual a functional type $t \rightarrow u$, where t and u are the parameter and return types which can be identified to classes. Attributes can be reduced to functions by typing their reader and writer functions. Let $l \ll l'$ be two local properties such that $\mathit{ltype}(l) = t \rightarrow u$ and $\mathit{ltype}(l') = t' \rightarrow u'$, then $\mathit{ltype}(l') <: \mathit{ltype}(l)$, $u' <: u$ and $t' <: t$ (in a covariant setting) or $t <: t'$ (in a contravariant one).

Furthermore, in the covariant case, $\mathit{ltype}(l)$ must also be *tagged* according to whether l can be covariantly redefined or not[23].

Virtual types. Virtual types represent a safer alternative to covariant specialization which can be completely type safe, according to Torgersen [1998], or more permissive as in Beta[24].

We follow here the presentation of virtual types by Torgersen [1998]. 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. The key idea of virtual types is that (i) as virtual types they must be covariantly redefined, (ii) as a parameter or return type they are invariant, and (iii) they have no proper subtypes, unless they are final ($T = D$). Besides these explicit virtual types, `MyType` (aka `SelfType` or `ThisType`) represents the implicit declaration `MyType`$<: D$ in each class D—it corresponds to `like current` in Eiffel. In all cases, for such a local property l, whatever its form $T <: D$ or $= D$, $\mathit{ltype}(l) = D$. In the former form, it must also be tagged as covariant. Furthermore, if another local property, say l', is typed by a virtual type T, this virtual type T is replaced in $\mathit{ltype}(l')$ by the corresponding concrete type D. Accordingly, $\mathit{ltype}(l')$ must be tagged as covariant if the virtual type is used for typing a parameter and has the form $T <: D$.

Under the CWA, virtual types and covariant parameters are strictly equivalent—when a parameter type is redefined in a subclass, a virtual type can be introduced in the superclass and substituted to the parameter type. Hence, virtual types are a convenient way to declare that a parameter type is covariant. However, whereas virtual types of the form $T <: D$ have no proper subtypes, they actually have supertypes, namely all supertypes of D. So, a concrete parameter type, say D, could be covariantly redefined by a virtual type $T <: D$. This would bring back general unsafety.

Altogether, the aforementioned tradeoff between type safety and expressivity amounts to forbidding explicit parameter type redefinitions. When such a redefinition has not been anticipated when introducing the global 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].

[23] For the sake of simplification, we consider a single parameter—the generalization is straightforward and, of course, each parameter must be separately tagged.

[24] Torgersen [1998] claims that virtual types are type safe *without any loss of expressivity*. This claim actually implies the CWA. Under the OWA, virtual types can be safe but this reduces the expressivity for future classes. So we prefer the permissive approach of Beta. The point has anyway no effect on the present discussion, since it is only a matter of type checking specific expressions.

4.4.2 *Record types and global 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 1984]. However, names are now inadequate and they must be replaced by global properties. This should not have any effect on the considered type system. Substituting global properties to names would likely allow to adapt the metamodel to any other formal semantics of object-oriented programs such as object calculi [Abadi and Cardelli 1996].

Hence, the type of a global property g in a class c, noted $\mathit{gtype}(g, c)$, is the type of the local property l of g defined in, or inherited by, c. If $\mathit{ltype}(l)$ is tagged as covariant, gtype must be tagged accordingly. Finally, the type of a class c, noted $\mathit{ctype}(c)$, is a function which maps each global property in G_c to its type in c:

$$\mathit{gtype}(g, c) = \mathit{ltype}(\mathit{sel}(g, c)) \ , \tag{16}$$

$$\mathit{ctype}(c) = \{g \mapsto \mathit{gtype}(g, c)\}_{g \in G_c} \ . \tag{17}$$

Finally, type checking a method call amounts to checking the actual parameter and target types against $\mathit{gtype}(g, c)$, where c is the receiver's static type and g the considered global property. If $\mathit{gtype}(g, c)$ has been tagged as covariant, i.e. if it involves a covariant type, e.g. a virtual type of the form $T <: D$, the call is rejected or a dynamic type check is inserted, according to the type safety policy[25].

4.4.3 *Local property conflicts and intersection types.* So far there is nothing new except substituting global properties to names. Local property conflicts require combining types. Let g be a global property, namely a method, and c a class which presents a local property conflict, such that $\mathit{spec}(g, c) = \{l, l'\}$, $\mathit{ltype}(l) = t \to u$ and $\mathit{ltype}(l') = t' \to u'$. If the programmer solves the conflict by redefining g in c, this redefinition must conform to both l and l', as usual. However, if the language does not force the redefinition, i.e. if a default selection mechanism is used, such as linearization, the type of g in c implies *intersection types* [Compagnoni and Pierce 1996; Reynolds 1996], together with *union types* in a contravariant setting only:

$$\mathit{gtype}(g, c) = \begin{cases} t \cap t' \to u \cap u' & (\mathit{covariant}) \\ t \cup t' \to u \cap u' & (\mathit{contravariant}) \end{cases} \tag{18}$$

Intersection types have their analogue at the class level—i.e. when a class C is defined as a subclass of A and B, it can be understood as a subclass of an implicit abstract class $A\&B$ such that $G_{A\&B} = G_A \cup G_B$, and for all $g \in G_A \cap G_B$, $\mathit{gtype}(g, A\&B) = \mathit{gtype}(g, A) \cap \mathit{gtype}(g, B)$.

Virtual type conflicts. It follows from the specification of virtual types that the 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 entails that $T <: D \cap D'$ in the conflicting class;

[25] Of course, some specific situations can make the call type safe. Besides the aforementioned case when the receiver is proven to be exact, a safe case occurs when the actual and formal parameters have the *same virtual type*, dependent on the *same receiver*, e.g. in `x.foo(x.bar)`, when the formal parameter of `foo` and the value returned by `bar` have the same virtual type.

—$T <: D$ and $T = D'$—this is an error unless $D' <: D$;

—$T = D$ and $T = D'$—this is an error unless $D = D'$.

These errors are unrecoverable, i.e. the two conflicting classes C and C' cannot have any common subclass, unless the virtual type T is modified in C or C'.

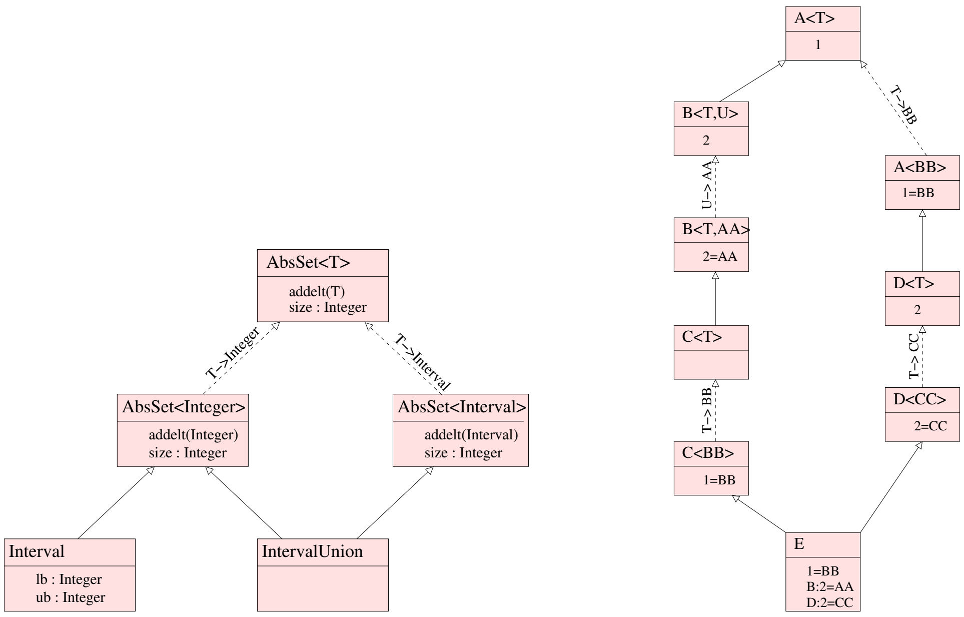
Type conflicts. Covariant specialization, either for parameter or return types, with explicit redefinition or virtual types, leads to intersection types. This makes the type system a lattice but it may yield *type conflicts*, since $x \cap y = \bot$ when there is no common subclass to x and y. This is an error, since $\bot$ cannot have any instance. However, this error cannot be detected at compile-time under the OWA, since all future subclasses have not to be known. In contrast, union types do not yield such conflicts since two classes have always a common superclass, the hierarchy root.

4.4.4 *Genericity.* Virtual types represent also an alternative to parametrized classes. This suggests that *formal type parameters* might be included in the metamodel 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, accordingly to usual type theories, two instances of the same parametrized class, say $A\langle C\rangle$ and $A\langle D\rangle$ are never related by subtyping.

A formal integration would require to add a new association to the metamodel, 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 parameter 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. Moreover, as a property name, it may yield global property conflicts, when a class inherits two formal type parameters introduced in different classes with the same position. Figure 8-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.

However, inheriting multiple instances of the same parametrized class amounts to local property conflict and is strongly constrained. Consider the example in Figure 8-a. Sets can be implemented in different ways and the abstract class `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 `AbsSet`$\langle$`Integer`$\rangle$ but the programmer may want to consider also interval unions as sets of intervals, thus making `IntervalUnion` specialize both `AbsSet`$\langle$`Integer`$\rangle$ and `AbsSet`$\langle$`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 third case of virtual type conflicts, where $T =$ `Integer` and $T =$ `Interval`.



-a Local property conflict

-b Global property conflict

Fig. 8. Multiple Inheritance and Generics—a class cannot specialize two distinct instances of the same parametrized class (a) and a fully qualified syntax disambiguates parameter position conflicts (b). Solid lines represent specialization and dashed lines instantiation.

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. `size`⟨`Integer`⟩ or `size`⟨`Interval`⟩, could clear up the situation, but implementing it would be a problem. Obviously, *homogeneous implementations*[26] would make it impossible, since both methods cannot stand at the same position in the `AbsSet`⟨`T`⟩ method table. On the contrary, C++ *heterogeneous implementation* would likely make it possible, at the expense of some implied *repeated inheritance* (see Section 4.5.1).

4.4.5 *Method combination.* Method combination also presents some interesting, but slightly paradoxical, typing issues.

Call to `super` . First, `super` has implicitly the type of the redefined method in the superclass[27]. Let l be a local property of the global property $g = glob(l)$, whereby $ltype(l) = t \rightarrow u$. Suppose that g is introduced in $intro(g)$ with the local property l'' whose type $ltype(l'') = t'' \rightarrow u''$. Suppose that a call to `super` in l calls

[26] 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.

[27] Remember that our usage of `super` differs from that of Java. In Java, `super` is typed by the superclass but, here, `super` implies the global properties, hence corresponds to `super.foo` in Java.

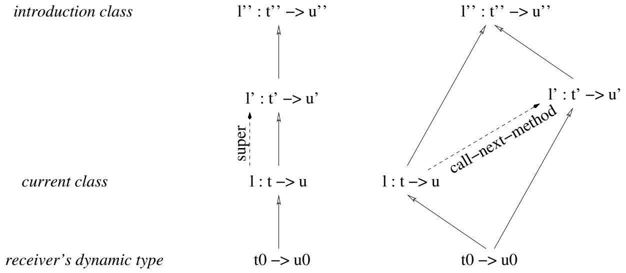


Fig. 9. Typing `super` and `call-next-method`—solid lines represent subtyping and redefinition, dashed lines calls to `super` and `call-next-method`.

another local property l': then

$$ltype(l) <: ltype(l') <: ltype(l'') \tag{19}$$

and both the existence of l' and the type of `super`, i.e. $ltype(l') = t' \rightarrow u'$, are statically known (Figure 9).

In practice, calls to `super` often represent one of two specific patterns: (i) passing the method parameter values as they were received or, conversely, (ii) returning by the calling method the value returned by `super`. In the former case, the parameter can be implicit and both cases can be combined in a `return super` instruction, without parameters.

—Both cases are of course type safe in a type invariant framework, including virtual types, or when the considered types have no proper subtypes.

—The latter case is unsafe when strictly covariant return types are used—i.e. when $u <: u'$ and $u \neq u'$.

—Interestingly, the former case is type safe when the parameter types are covariant, i.e. in a potentially type unsafe framework—i.e. when $t <: t'$. This is an interesting proof of monotonicity but the exact dual of contravariance rule!

—In contrast, the former case is unsafe when the parameters are strictly contravariant—i.e. when $t' <: t$ and $t' \neq t$.

—In the general case, actual parameter and target types must be respectively checked against t' and u'.

Bottom-up `call-next-method` . In Clos, the `call-next-method` syntax allows for the former case, since its usage without parameter implies passing all method parameters as they were received. However, `call-next-method` cannot be statically typed as precisely as `super`. Actually, if $cnm(l) = l'$, l' can be defined in some directly unrelated class and is never statically known. First, as aforementioned, the existence of l' can be statically ensured only if the method has not been declared abstract in the superclasses. Hence, an explicit run-time check may be required, or alternatively . Moreover, the type bounds of (19) must be replaced by the following ones:

$$gtype(g, c) = t_0 \rightarrow u_0 <: ltype(l') <: ltype(l''). \tag{20}$$

where c is the receiver's dynamic type (Figure 9).

Firstly, in all cases, the static return type of `call-next-method` is u'', the return type of the method in $\mathit{intro}(g)$—hence, case (ii) is type safe only if the current method has not redefined the return type, i.e. if $u = u''$. Moreover, in a covariant setting,

—the unknown parameter types of l' are supertypes of the static parameter types—possibly intersection types—of the considered method in the receiver dynamic type, i.e. $t_0 <: t'$. Thus, if all `call-next-method` pass the method input parameters—i.e. if `call-next-method` is always called without explicit parameters—this will be type safe.

—When some `call-next-method` passes explicit parameters, the current actual parameter is no longer forced to be a subtype of t_0 and `call-next-method` will be unsafe throughout the global property[28].

In contrast, in a type safe strictly contravariant setting,

—(20) only provides lower bounds on parameter types, i.e. $t'' <: t'$, and `call-next-method` will be unsafe unless an explicit parameter with static type t'' is passed—hence, case (i) is type safe only if the current method has not redefined the parameter type, i.e. if $t = t''$.

—In the case (ii), regarding the return type, the only difference between `super` and `call-next-method` is that `return call-next-method` is always unsafe[28], whereas `return super` is type safe, when the return type is invariant for the two considered methods.

The overall conclusion is that the safest and most expressive specification of `call-next-method` is in a potentially type unsafe covariant setting and it does not allow for explicit parameters. Accordingly, it does not increase unsafety. In a contravariant framework, `call-next-method` seems to be antinomic with type redefinition.

Top-down `call-next-method` . In Beta, bottom-up `super` is replaced by a top-down mechanism, `inner`. As Beta is in single inheritance, if l calls l' by `inner`, then $\mathit{ltype}(l') <: \mathit{ltype}(l)$ and the situation is similar to `call-next-method`, except that the bound is based on l instead of l''.

In multiple inheritance, reversing the direction of `call-next-method` does not change very much the conclusions.

4.5 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 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 global properties from property identifiers and all languages agree with the metamodel, at least in the global properties. On the contrary, when naming

[28] Unless the considered type has no proper subtype, including virtual types—hence, its is recommended for return types also, not only for parameter types, to use virtual types instead of explicitly covariant types. Moreover, it improves type checking (see Note 25).

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.

4.5.1 *Inheritance and metamodels in some object-oriented languages.* We review hereafter the most commonly used languages.

In the SMALLTALK terminology, 'method' and 'method selector' respectively denote local and global 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 global 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. 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. 3.1). In the example of Figure 6, 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 3.1 is dropped, but the call site becomes ambiguous, or the requirement is kept but both `department` are unified. Despite this last point, 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 global properties. In the following languages, one word (method, feature, etc.) stands for the two notions.

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 global 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[29]. However, multiple inheritance is possible with interfaces and, when a global property conflict occurs, JAVA cannot distinguish between two methods with the same name and signature. Accordingly, there is no reification of global 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.

EIFFEL is also almost fully compatible, but only in common usage. In EIFFEL terminology, 'feature' stands for property, without distinguishing the two kinds,

[29] With a slight restriction until JAVA 1.4, which has been raised in JAVA 1.5 [Ancona et al. 2000; Ancona et al. 2001].

even though a notion of *feature seed* could be understood as the introduction of global properties [Meyer 1992]. Feature renaming allows the programmer to deal with global property conflicts in the desired way. However, 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-named feature, as two distinct features; (ii) in a class, if two features from different seeds are inherited under the same name, then they are locally merged; (iii) multiple inheritance of the same class and transitivity edges are accepted, with a lot of renamings. Moreover, Eiffel does not follow the masking rule (Req. 4.1) and finds extra local property conflicts, as in Figure 7-b.

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[30], multiple inheritance raises several difficulties. Firstly, the keyword `virtual`[31] is mandatory in inheritance, to avoid duplication of attributes introduced in a superclass inherited through multiple paths. In C++ jargon, our proposition resorts to *shared multiple inheritance*, in contrast with *repeated multiple inheritance*[32]. Nevertheless, C++ does not distinguish between two methods with the same signature, introduced in different unrelated superclasses but, in contrast, it signals local property conflicts according to the masking rule.

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, page 27] 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 virtual types—actually, when covariant parameter types are restricted to virtual types, covariance remains compatible with static overloading [Torgersen 1998].

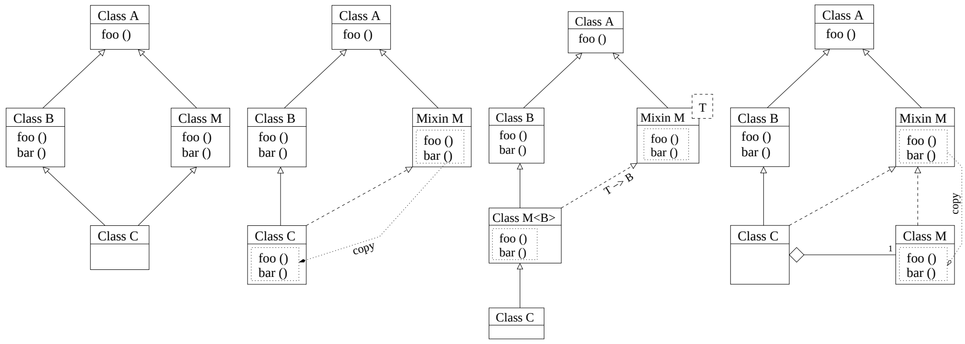
4.5.2 *Alternatives to full multiple inheritance.* Several alternatives have been proposed, which rely on a degraded form of multiple inheritance, e.g. Java or C# interface multiple subtyping, *mixins* [Bracha and Cook 1990] or *traits* [Ducasse et al. 2005]. *Interfaces* have been discussed about Java—they 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, before becoming explicit patterns or even first class entities in theories or actual programming languages. Mixin proposals are numerous and variable—so this sec-

[30] 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.

[31] 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 6, page 9.

[32] It is common opinion that "repeated inheritance is an abomination" [Zibin and Gil 2003, note 2].



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

Fig. 10. Multiple Inheritance and Mixins—(a) in multiple inheritance, the example reproduces the two conflicting situations in Figures 6 and 7—`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`).

tion 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. 2004], 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 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[33] must be superclasses of B. Intuitively, the effect of this definition is to copy the definition of M into B (Fig. 10-b). An alternative and more formal view involves *parametrized heir classes*, whereby C `extends` $M\langle B\rangle$ (Fig. 10-c). This is the common way of using mixins with C++ *templates* [VanHilst and Notkin 1996; Smaragdakis and Batory 1998]. The *heterogeneous* implementation of templates makes it possible, but mixins cannot be separately compiled[34]. 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. 10-d). The latter approach has also been called *automated delegation* [Viega et al. 1998]. Actually, the Scala approach is a midterm between copy and proxy. M is compiled as an abstract class whose all methods are *static* and the corresponding redefined method in C calls this static method.

Actually, mixins are not exempt from multiple inheritance conflicts—for instance,

[33] Of course, here 'superclass' denotes a class, not a mixin.
[34] See Note 26.

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. 10). Hence, mixins are not incompatible with the present metamodel, which could be extended to include them, in the same manner as for JAVA interfaces. Global 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 4.1). MIXJAVA presents a notable exception, with a `view` keyword which behaves like full qualification [Flatt et al. 1998]. Regarding local property conflicts, there is no uniform policy among various mixin propositions, 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 a linearization where M—or all the mixin set—precedes the totally ordered superclasses[35].

However, mixins are also—sometimes above all—a specification of how things are implemented. The code of M is at least indirectly copied into 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—this is the so-called `invokeinterface` primitive, which is not renowned for its efficiency [Alpern et al. 2001; Ducournau 2005]. See also [Ducournau 2002a] for a survey of object-oriented implementations.

Traits[36] [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[37], but these methods can have 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 global property conflicts. In a static typing framework, traits are types, like usual mixins [Nierstrasz et al. 2006].

4.5.3 *Conclusion on multiple inheritance.* The specifications of most languages could adopt the present metamodel while only marginally changing language syn-

[35] In a previous version of this paper, we wrote that "incidentally, the linearization used for the mixin part has all the flaws of multiple inheritance in early versions of FLAVORS and LOOPS—it is a depth-first ordering, hence not a linear extension and, of course, it is not monotonic. Such a linearization was already obsolete in the mid-80s." This was a misunderstanding and the linearization is actually a linear extension.

[36] 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.

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

tax, 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 C++ and EIFFEL would be markedly improved if they complied to this metamodel. In practice, 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.

Global property conflicts are not an obstacle to multiple inheritance—their solution is syntactically simple and intuitive for all programmers. 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 global property level. At the local property level, mixins and linearization are roughly equivalent[38], 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 10, 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; 2000a; Rayside and Kontogiannis 2001].

5. MODULES AND CLASSES

Let us now more formally present modules and class refinement, with a strong analogy with the class and property metamodel. In the following, we successively

[38] 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.

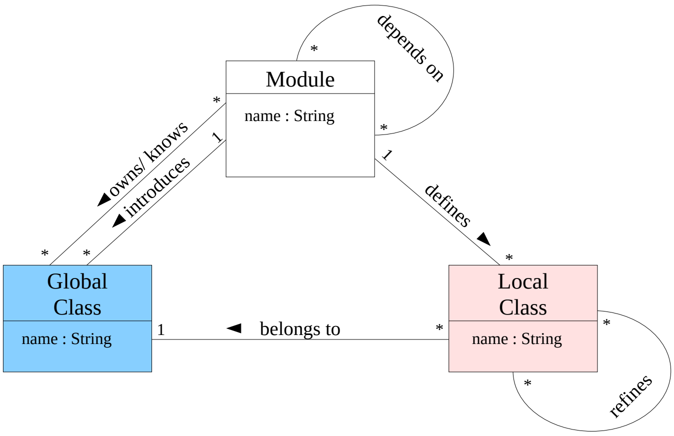


Fig. 11. Metamodel of Modules and Classes

present the metamodel, its formalization, an analysis of multiple import and multiple inheritance conflicts, and a discussion about static typing.

5.1 Module and Class Metamodel

A more rigorous approach follows from the observation that classes are to modules what properties are to classes, the dependence relation between modules matches specialization, and import matches inheritance. Formalizing this observation just involves defining an isomorphic metamodel with two entities associated with the class concept, and one entity associated with the module concept—Fig. 11.

5.1.1 *Modules.* *Modules* are hierarchies of classes. A module *depends* on zero, one or more other modules—if module m depends on module n, n is a supermodule of m, and m is a *submodule*[39] of n. Like the class specialization relation, the module dependence relation is a strict partial order.

Local classes are defined in modules. A local class is described by an ordinary *class definition* (Def. 3.10). *Global classes* gather local classes and are orthogonal to modules. Each module has global classes which correspond to the classes that it knows, i.e. those that can be used in the module, for specialization, refinement, typing or instantiation.

A module definition is a triplet constituted with the name of the module, the name of its supermodules, presumably already defined, and a set of local class definitions. The dependence relation supports an import mechanism—i.e. modules import the classes of their supermodules. The two kinds of classes yield two-level import. First of all, a module *imports* all global classes of its supermodules—this is *global class import*. Then, each local class definition is processed. If the name of the local class is the same as the name of an imported global class, the new local class is attached to the global class. If there is no such imported global class, a new global class with the same name is *introduced* in the module. Note that we do not distinguish between supermodules, according to whether their imported classes

[39] Remember that a submodule is not a module inside a module (*nested module*) but a module which depends on another module.

are only used in the current module for typing and instantiation, or whether they are also specialized or refined. Like visibility, this would be of practical interest but this does not change the model. Anyway, only the relation which supports specialization and refinement is required to be acyclic.

For any global class known by a module, the existence of the corresponding local class is supposed: it is either an explicit definition, or an implicit refinement. In the latter case, local classes are said to be *implicit classes* since they do not have explicit definitions. The *class refinement relation* is the analogue of property redefinition. It is deduced from the module dependence relation. On the other hand, since modules are class hierarchies, their local classes are also related by a *specialization relation.* This specialization relation is deduced from both the explicit declarations of superclasses in local class definitions and the corresponding relationship in supermodules. Hence, a module imports specialization relationships from its supermodules. Altogether, property inheritance is now supported by both specialization and refinement.

5.1.2 *Program and Module Semantics.* A *program* is a set of modules closed by the dependence relationship. It corresponds to a *bottom module* (possibly empty and implicit) that depends on every other module in the program. So far, the simplest way to establish the semantics of a program is probably to reduce it to a single class hierarchy by a kind of *flattening*[40] mechanism. There are roughly three ways for doing this, which are not exactly equivalent. The set of classes of the resulting class hierarchy may be:

(1) the set of global classes, with a specialization relationship resulting from the projection of the specialization of each module, and local properties defined as the most specific w.r.t. refinement;
(2) the set of local classes of the *bottom module*, submodule of all other modules, with specialization imported from other modules, and local properties defined as the most specific w.r.t. refinement;
(3) the union of all local class sets over all modules—including the implied bottom module—with specialization defined as the union of all specialization plus refinement, and local properties being exactly what they are in each module and each class. All local classes are abstract, except those of the bottom module.

Though conceptually simple, the last solution has an obvious drawback—the resulting class hierarchy would be quite large and multiple inheritance situations quite numerous. Moreover, it is not a good idea to put class specialization and refinement on the same level. Though they are intuitively akin, intuition tells us that refinement is, in some sense, stronger than specialization, in the same way as defining a property is stronger than inheriting it (*Masking rule*, page 21). On the other hand, the first two solutions are quite similar. The first one avoids the artifact of introducing an implied module and makes global classes concrete—they are the actual classes of the final program. However, conflict resolution may require to

[40] The term '*flattening*' is commonly used in extensions of object-oriented models—it involves translating a class hierarchy extended by some new features into an usual one without these new features. For instance, the term is used for mixins and traits [Nierstrasz et al. 2006]. It can be understood as a source-to-source transformation.

explicitly define the bottom module by inserting some *glue code*—see Section 5.3.4. For technical convenience, our definition will use both.

5.1.3 *Limit of the analogy.* Both models are isomorphic but it does not mean that modules are classes. Indeed, a fundamental difference must be stressed, namely modules do not exist at run-time. They have no instance, hence no constructor methods and there is no late binding at the module level since there is no receiver and classes are statically bound. Hence, Section 3.3 does not apply at the module level. Actually, we have exchanged the run-time semantics of late binding for the link-time semantics of flattening.

5.2 Notations and Formal Definitions

A *module* is a hierarchy of classes and a *program* is a hierarchy of modules.

Definition 5.1 (Module hierarchy). A *model of a program*, i.e. of a hierarchy of modules, is a tuple $\mathcal{P} = \langle X^{\mathcal{P}}, \prec^{\mathcal{P}}, G^{\mathcal{P}}, L^{\mathcal{P}}, N^{\mathcal{P}}, name_{\mathcal{P}}, glob_{\mathcal{P}}, intro_{\mathcal{P}}, def_{\mathcal{P}}\rangle$, where:

—$X^{\mathcal{P}}$ is the set of *modules* and $\prec^{\mathcal{P}}$ is the *module dependence* relationship, which satisfies the same notations and properties as class specialization;

—$G^{\mathcal{P}}$ and $L^{\mathcal{P}}$ are the set of *global* and *local classes*;

—$N^{\mathcal{P}}$ is the set of *module* and *class identifiers* (names);

—$name_{\mathcal{P}} : X^{\mathcal{P}} \uplus G^{\mathcal{P}} \uplus L^{\mathcal{P}} \to N^{\mathcal{P}}$ is the naming function of modules and classes.

—$glob_{\mathcal{P}} : L^{\mathcal{P}} \to G^{\mathcal{P}}$ associates with each local class a global class;

—$intro_{\mathcal{P}} : G^{\mathcal{P}} \to X^{\mathcal{P}}$ associates with a global class the module introducing it;

—$def_{\mathcal{P}} : L^{\mathcal{P}} \to X^{\mathcal{P}}$ associates with a local class the module in which it is defined.

Each module $\mathcal{M} \in X^{\mathcal{P}}$ is a class hierarchy $\mathcal{M} = \langle X^{\mathcal{M}}, \prec^{\mathcal{M}}, G^{\mathcal{M}}, L^{\mathcal{M}}, N^{\mathcal{M}}, name_{\mathcal{M}}, glob_{\mathcal{M}}, intro_{\mathcal{M}}, def_{\mathcal{M}}\rangle$, where

—$X^{\mathcal{M}} = L^{\mathcal{P}}_{\mathcal{M}}$ is the set of local classes of $\mathcal{M}$ in $\mathcal{P}$;

—all $N^{\mathcal{M}}$ are the same name-set;

—all $G^{\mathcal{M}}$ are part of some set G of global properties;

—all $L^{\mathcal{M}}$ are disjoint, so *def* (resp. *glob*) is an unambiguous shorthand for some $def_{\mathcal{M}}$ (resp. $glob_{\mathcal{M}}$).

Unless otherwise stated, all notations, equations and constraints available for $\mathcal{H}$ (Section 3.2) are also available for $\mathcal{P}$—apart from substituting $\mathcal{P}$ (resp. 'module', 'class') to $\mathcal{H}$ (resp. 'class', 'property')—and for $\mathcal{M}$—except (1) and (3) which are slightly modified into (21-23). Of course, exponents $\mathcal{P}$ or $\mathcal{M}$ are now mandatory in all notations.

5.2.1 *Class refinement.* Module dependence entails *class refinement*, which is simply the analogue of property redefinition (Def. 3.9):

Definition 5.2 (Class refinement). *Class refinement* is defined as the relationship $\ll^{\mathcal{P}}$ between a local class in $L^{\mathcal{P}}_{\uparrow m}$, defined in some module m, and the corresponding local classes in the supermodules of m:

$$c \ll^{\mathcal{P}} c' \overset{\text{def}}{\iff} glob_{\mathcal{P}}(c) = glob_{\mathcal{P}}(c') \wedge def_{\mathcal{P}}(c) \prec^{\mathcal{P}} def_{\mathcal{P}}(c') \ .$$

This is a strict partial order and $\ll^{\mathcal{P}}_d$ denotes its transitive reduction.

As a small difficulty arises regarding $X^{\mathcal{M}}$, for simplification's sake, one assumes that $X^{\mathcal{M}}$ includes a local class definition, possibly empty—i.e. without any local property definition—for each[41] class in the supermodules of $\mathcal{M}$:

Constraint 5.3 (Implicit class refinement).

$$\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}' \Longrightarrow glob_{\mathcal{P}}(X^{\mathcal{M}'}) \subseteq glob_{\mathcal{P}}(X^{\mathcal{M}}) \ .$$

Given two dependent modules $\mathcal{M}' \prec^{\mathcal{P}} \mathcal{M}$, let $id^{\mathcal{M}}_{\mathcal{M}'}$ be the function $id^{\mathcal{M}}_{\mathcal{M}'} : X^{\mathcal{M}} \rightarrow X^{\mathcal{M}'}$ which maps a class $c \in X^{\mathcal{M}}$ to the single $c' \in X^{\mathcal{M}'}$ such that $c' \ll^{\mathcal{P}} c$. It follows from Constraint 5.3 that $id^{\mathcal{M}'}_{\mathcal{M}}$ is a well-defined injective total function.

Moreover, one must also ensure that the specialization relationship $\prec^{\mathcal{M}}$ includes its analogue in supermodules.

Constraint 5.4 (Specialization import). Let $\mathcal{M} \in X^{\mathcal{P}}$ a module. For all supermodules $\mathcal{M}'$ such that $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$, for all c, c' in $X^{\mathcal{M}'}$, then

$$c \prec^{\mathcal{M}'} c' \Longrightarrow id^{\mathcal{M}'}_{\mathcal{M}}(c) \prec^{\mathcal{M}} id^{\mathcal{M}'}_{\mathcal{M}}(c') \ .$$

Moreover, let $\prec^{\mathcal{M}}_{e}$ denote the specialization relationship explicitly defined for module $\mathcal{M}$ by successive class definitions (Def. 3.10). Then $\prec^{\mathcal{M}}$ is defined as the transitive closure of the union of $\prec^{\mathcal{M}}_{e}$ and $id^{\mathcal{M}'}_{\mathcal{M}}(\prec^{\mathcal{M}'})$, for all $\mathcal{M}'$ such that $\mathcal{M} \prec^{\mathcal{P}}_{d} \mathcal{M}'$.

The last constraint makes modules slightly abnormal class hierarchies since, for normal ones, $\prec_e$ and $\prec_d$ coincide. However, to be legal, $\prec^{\mathcal{M}}$ must be antisymmetric—possible violations are discussed in Section 5.3.2.

Moreover, when considering global properties, class refinement makes modules special class hierarchies, that do not verify Constraint 3.7, because class refinement in turn induces inheritance of global properties.

Constraint 5.5 (Refinement-based inheritance). Let $\mathcal{M}, \mathcal{M}' \in X^{\mathcal{P}}$, such that $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$. Then $G^{\mathcal{M}'} \subseteq G^{\mathcal{M}}$.

Let $c \in X^{\mathcal{M}}$ and $c' \in X^{\mathcal{M}'}$ such that $c \ll^{\mathcal{P}} c'$. Then $G^{\mathcal{M}'}_{c'} \subseteq G^{\mathcal{M}}_{c}$.

Moreover, for all $g \in G^{\mathcal{M}'}$, $intro_{\mathcal{M}}(g) = id^{\mathcal{M}'}_{\mathcal{M}}(intro_{\mathcal{M}'}(g))$.

The last subconstraint does not take a possible *property generalization* into account—item (4) in Section 2.1. We shall examine it in Section 5.3.3.

Altogether, the set $G^{\mathcal{M}}_{c}$ of global properties known by c splits up into the sets of properties introduced in c ($G^{\mathcal{M}}_{+c}$), inherited by specialization ($G^{\mathcal{M}}_{\uparrow c}$) and by refinement ($G^{\mathcal{M}}_{\Uparrow c}$). The following equations replace (1) and (3):

$$G^{\mathcal{M}}_{c} \stackrel{\text{def}}{=} (G^{\mathcal{M}}_{\uparrow c} \cup G^{\mathcal{M}}_{\Uparrow c}) \uplus G^{\mathcal{M}}_{+c} = G^{\mathcal{M}}_{\uparrow c} \uplus G^{\mathcal{M}}_{\Uparrow + c} \uplus G^{\mathcal{M}}_{+c} \ , \tag{21}$$

$$G^{\mathcal{M}}_{\Uparrow c} \stackrel{\text{def}}{=} \bigcup_{c \ll^{\mathcal{P}} c'} G^{def_{\mathcal{P}}(c')}_{c'} \ , \tag{22}$$

$$G^{\mathcal{M}}_{\Uparrow + c} \stackrel{\text{def}}{=} \bigcup_{c \ll^{\mathcal{P}} c'} G^{def_{\mathcal{P}}(c')}_{+c'} \ , \tag{23}$$

$$G^{\mathcal{M}}_{+c} \cup G^{\mathcal{M}}_{\Uparrow c} = intro^{-1}_{\mathcal{M}}(c) \ . \tag{24}$$

[41] Actually, this is only required for all superclasses and subclasses of refined classes.

One just has to add the set $G^{\mathcal{M}}_{\Uparrow+c}$ of global properties introduced by classes which are refined by c. Finally, a small complement must be added to the tuples $\mathcal{M}$, since global properties are now introduced not only by a class but also by a given module. Therefore, for each module $\mathcal{M} \in X^{\mathcal{P}}$, one extends the domain of function $intro_{\mathcal{P}} : G^{\mathcal{P}} \uplus G \rightarrow X^{\mathcal{P}}$ in such a way that, for all $g \in G$:

$$g \in G^{\mathcal{M}}_{+c} \overset{\text{def}}{\Longleftrightarrow} intro_{\mathcal{P}}(g) = \mathcal{M} \ . \tag{25}$$

Class refinement also entails a redefinition relationship on local properties:

Definition 5.6 (Refinement-based property redefinition). Let $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$ be two dependent modules. *Refinement-based redefinition* is the relationship, noted $\lhd$, defined on the sets $L^{\mathcal{M}}$ and $L^{\mathcal{M}'}$ of local properties. Given $lp \in L^{\mathcal{M}}$ and $lp' \in L^{\mathcal{M}'}$, then

$$lp \lhd lp' \overset{\text{def}}{\Longleftrightarrow} glob_{\mathcal{M}}(lp) = glob_{\mathcal{M}'}(lp') \wedge def_{\mathcal{M}}(lp) \ll^{\mathcal{P}} def_{\mathcal{M}'}(lp') \ .$$

5.2.2 *Module Definition.* The program is built by successive module definitions. A module definition is analogous to a class definition (Def. 3.10), except that 'module' (resp. 'class') replaces 'class' (resp. 'property'). Moreover, local class definitions differ slightly from Definition 3.10.

Definition 5.7 (Module definition). A *module definition* is a triplet ⟨`modname`, `supermodnames`, `localclassdef`⟩, where `modname` is the name of the newly defined module, `supermodnames` is the set of names of its direct supermodules—they are presumed to be already defined—and `localclassdef` is a set of local class definitions.

The effect of this definition is given by the following protocol.

(i) *Module hierarchy update.* A *new* module $m = \mathcal{M}$ with name `modname` is added to $X^{\mathcal{P}}$. For each supermodule name in `supermodnames`, a pair is added to $\prec^{\mathcal{P}}_d$ (ambiguities and transitivity edges are checked, as in Definition 3.10).

(ii) *Global class import.* $G^{\mathcal{P}}_{\uparrow m}$ is computed according to (2) and global class conflicts are checked (Def. 3.4).

(iii) *Local class hierarchy.* Then each class definition in `localclassdef` is treated as a class definition, irrespective of the property definitions (Def. 3.10, step i). The *new* local classes are created, with their corresponding names—this yields $L^{\mathcal{P}}_m$. We assume here that a potentially empty class definition is explicitly provided for each global class in $G^{\mathcal{P}}_{\uparrow m}$ but a default behavior would be straightforward. $L^{\mathcal{P}}_{\uparrow m}$ is determined by (9). Specialization relationships are imported from direct supermodules (Const. 5.4) and possible cycles in the resulting relationship are checked—Section 5.3.2. Then, $G^{\mathcal{P}}_{+m}$ is constituted as the set of new global classes corresponding to each local class in $L^{\mathcal{P}}_{+m} = L^{\mathcal{P}}_m \backslash L^{\mathcal{P}}_{\uparrow m}$. $L^{\mathcal{P}}_m$ and $G^{\mathcal{P}}_{+m}$ are then respectively added to $L^{\mathcal{P}}$ and $G^{\mathcal{P}}$. Local class conflicts are not considered since they are treated during the local class definitions.

(iv) *Local class properties.* In the definition of the module $m = \mathcal{M}$, the definition of the local class c is made with the usual triplet ⟨`classname`, `supernames`, `localdef`⟩ (Def. 3.10). Once the complete class hierarchy $(X^{\mathcal{M}}, \prec^{\mathcal{M}})$ has been processed, the protocol proceeds to property inheritance and definition. First, global property inheritance is managed. The effect is the same as in Definition 3.10 (step ii), except that class refinement must be considered (Def. 5.5). Thus the set of properties inherited through specialization ($G^{\mathcal{M}}_{\uparrow c}$) and refinement ($G^{\mathcal{M}}_{\Uparrow c}$) are first computed (2,23). Global property conflicts are checked in the usual way, but in the set $G^{\mathcal{M}}_{\uparrow c} \cup G^{\mathcal{M}}_{\Uparrow c}$ instead of $G^{\mathcal{M}}_{\uparrow c}$. $G^{\mathcal{M}}_{+c}$ is then constituted as the set of newly introduced global properties—$G^{\mathcal{M}}_{+c} = G^{\mathcal{M}}_{c} \backslash (G^{\mathcal{M}}_{\uparrow c} \cup G^{\mathcal{M}}_{\Uparrow c})$. Finally, all local property definitions in `localdef` are treated in the usual way (Def. 3.10, step iii-iv). Note that a slight difference may occur with *global property generalization* discussed in Section 5.3.3.

5.2.3 *Semantics of a Program.* As aforementioned, the semantics of a program is given by a *flattening* mechanism which maps a module hierarchy to a single normal class hierarchy.

Definition 5.8 (Program semantics). The *semantics of program* $\mathcal{P}$ is the class hierarchy $\mathcal{Q} = \langle X^{\mathcal{Q}}, \prec^{\mathcal{Q}}, G^{\mathcal{Q}}, L^{\mathcal{Q}}, N^{\mathcal{Q}}, name_{\mathcal{Q}}, glob_{\mathcal{Q}}, intro_{\mathcal{Q}}, def_{\mathcal{Q}}\rangle$, where:

—$X^{\mathcal{Q}} = G^{\mathcal{P}}$ is the set of global classes of $\mathcal{P}$;

—$N^{\mathcal{Q}}$ is a name-set build from the common name-set for all $\mathcal{M} \in X^{\mathcal{P}}$;

—$name_{\mathcal{Q}}$ is defined by equations (29-30);

—$G^{\mathcal{Q}} = G$ is the set of all global properties in $\mathcal{P}$.

Other components of the hierarchy $\mathcal{Q}$ are defined by equations (26–31).

An alternative definition involves the *bottom module* $\perp$, which is empty—i.e. `localclassdef` $= \emptyset$—and depends on each module in $\mathcal{P}$. Then $glob_{\mathcal{P}}$ is a one-to-one correspondence between $X^{\perp} = L^{\mathcal{P}}_{\perp}$ and $G^{\mathcal{P}} = X^{\mathcal{Q}}$.

The resulting class specialization is then defined as the class specialization in the bottom module (Const. 5.5), or its image by $glob_{\mathcal{P}}$:

$$\prec^{\mathcal{Q}} \stackrel{\text{def}}{=} glob_{\mathcal{P}}(\prec^{\perp}) \ . \tag{26}$$

Violations of antisymmetry are discussed in Section 5.3.2.

The set of global properties is merely the union of all sets of global properties:

$$G^{\mathcal{Q}} = G = \bigcup_{\mathcal{M}\in X^{\mathcal{P}}} G^{\mathcal{M}} = G^{\perp} = \bigcup_{gc\in X^{\mathcal{Q}}} G^{\mathcal{Q}}_{gc} \ , \tag{27}$$

and, for all global class gc, $G^{\mathcal{Q}}_{gc}$ is defined as the union of the sets of global properties of all local classes of gc:

$$G^{\mathcal{Q}}_{gc} \stackrel{\text{def}}{=} \bigcup_{lc\in glob^{-1}_{\mathcal{P}}(gc)} G^{def_{\mathcal{P}}(lc)}_{lc} = G^{\perp}_{gc} \ . \tag{28}$$

The function $intro_{\mathcal{Q}}$ is defined by $glob_{\mathcal{P}} \circ intro_{\perp}$. In the absence of *global property generalization* (Section 5.3.3), this follows from the fact that all local classes which introduce a given global property belong to the same global class (Const. 5.5). However, property generalization does not change this definition.

In the hierarchy $\mathcal{Q}$, all names are fully qualified names in $\mathcal{P}$. Hence, $N^{\mathcal{Q}} \stackrel{\text{def}}{=} (N^{\mathcal{P}} \times N^{\mathcal{P}}) \uplus (N^{\mathcal{P}} \times N^{\mathcal{P}} \times N^{\mathcal{P}} \times N^{\mathcal{P}})$ and $name_{\mathcal{Q}} : X^{\mathcal{Q}} \uplus G^{\mathcal{Q}} \uplus L^{\mathcal{Q}} \to N^{\mathcal{Q}}$ is defined by:

$$\forall c \in X^{\mathcal{Q}}, name_{\mathcal{Q}}(c) \stackrel{\text{def}}{=} name_{\mathcal{P}}(intro_{\mathcal{P}}(c)){:}name_{\mathcal{P}}(c) \tag{29}$$

$$\forall g \in G^{\mathcal{Q}}, name_{\mathcal{Q}}(g) \stackrel{\text{def}}{=} name_{\mathcal{Q}}(intro_{\mathcal{Q}}(g)){::}name_{\mathcal{P}}(intro_{\mathcal{P}}(g)){:}name_{\perp}(g) \tag{30}$$

In other words, $\mathcal{Q}$ determines the semantics of $\mathcal{P}$ in terms of instances of the metamodel. However, the denotation of the program text, i.e. the mapping from names to instances, cannot be determined only with unqualified names in $\mathcal{P}$—mediation of the current module $\mathcal{M}$ is required. This will be discussed and exemplified in the following section.

So far, besides names, $\mathcal{Q}$ and $\perp$ are exactly isomorphic. This will change when considering local properties, since $\perp$ is empty, whereas local properties are required to make $\mathcal{Q}$ a real program. The local properties of a class are defined as the most $\lhd$-specific local properties of all local classes of the considered global class:

$$L^{\mathcal{Q}}_{gc} \overset{\text{def}}{=} \min_{\lhd}\Big(\biguplus_{lc \in glob^{-1}_{\mathcal{P}}(gc)} L^{def_{\mathcal{P}}(lc)}_{lc} \Big) \ . \tag{31}$$

Of course, this is a sound local property set only if there are no two local properties belonging to the same global property—i.e. the restriction of the function $glob_{\mathcal{Q}}$ on each $L^{\mathcal{Q}}_{gc}$ must be injective (Const. 3.6). Otherwise, a conflict occurs, which must be solved, and this solution involves defining some *glue code* in the bottom module. This will be discussed at length in Section 5.3. The whole program is constructed from these sets $L^{\mathcal{Q}}_{gc}$, as in Definition 3.10.

Finally, $\mathcal{Q}$ also determines the run-time behavior of $\mathcal{P}$—method invocation is defined as in Section 3.3, except for the call to `super` and method combination which will be discussed in Section 5.3.5.

5.3 Multiple Import and Inheritance

As a result of the previous definitions, refining a subclass automatically induces a kind of multiple inheritance, through class specialization ($\prec^{\mathcal{M}}$) and class refinement ($\ll^{\mathcal{P}}$). Hence, it is impossible to ignore the problems caused by multiple inheritance. Moreover, the dependence relationship between modules ($\prec^{\mathcal{P}}$) can itself be 'multiple'. In this section, we analyze these new conflicts and investigate their treatment in the light of the metamodel.

A convention similar to that of Figure 1 applies to all figures: local classes appear as small named inner boxes, and modules are larger numbered enclosing boxes. Only specialization relations ($\prec^{\mathcal{M}}$) in a module and dependence between modules ($\prec^{\mathcal{P}}$) are drawn. Local classes in the same global class have the same name—hence, the refinement relation ($\ll^{\mathcal{P}}$) between classes remains implicit. Moreover, relevant implicit classes and implicit specializations are drawn in dashed lines. In the following, $p, q, r \ldots$ denote property names, $A, B, C \ldots$ denote class names, $1, 2, 3 \ldots$ denote modules, and A_1 denotes the local class in module 1 corresponding to the global class named A.

5.3.1 *Multiple Import.* Global and local class import yields the same conflicts as global and local property inheritance.

A *global class conflict* occurs when a module imports two homonymic global classes from two different supermodules—Fig. 12-a. It is isomorphic to global property conflict, has the same definition (Def. 3.4) and it can be solved in the same way. As modules are usually name-spaces, fully qualified names are the most natural solution here (Section 4.1)—hence, in the example, `1:Bow` and `2:Bow` unambiguously denote the two conflicting global classes. However, a class renaming

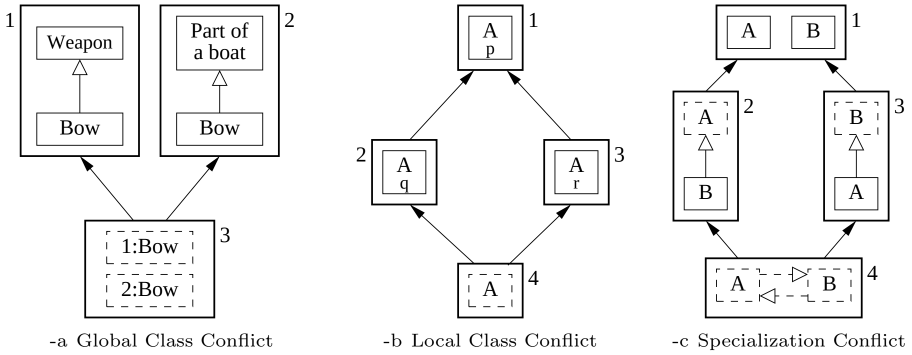


Fig. 12. Multiple Class Import and Conflict

mechanism, as in Eiffel and its configuration language Lace [Meyer 1997], would be an alternative.

In a similar way, a *local class conflict* would occur when a module imports several most specific classes from its supermodules, i.e. when $|spec_{\mathcal{P}}(c, m)| > 1$ for a module m and a global class c (Def. 3.11). However, the assumption that all imported classes are refined with a default empty local class (Const. 5.3) makes such a situation impossible. Let us consider, for a while, that this constraint does not hold. Figure 12-b illustrates this conflict configuration: module 4 imports the global class A introduced by module 1 and explicitly refined by the local classes A_2 and A_3 in the corresponding supermodules. Nevertheless, apart from the structural analogy, such a situation is quite different from *local property conflicts*. Whereas a local property is generally required to be unique and atomic in a class, local classes can be understood as sets of local and global properties. Hence, unlike general local property conflicts, the solution here is straightforward—namely, *refining* A by an empty local class A_4 which *combines*, through refinement-based inheritance (Const. 5.5), the two conflicting classes, A_2 and A_3. One must however note that the solution still resorts to re(de)fining and combining, like general local property conflicts. Therefore, besides being a technical facility, default empty local classes are also the solution to local class conflicts. Of course, some property conflicts may follow, that we shall examine hereafter.

5.3.2 *Cyclic Specialization and Class Unification.* A new conflict configuration, named *specialization conflict*, appears in a module when two local classes specialize each other, after importing the specialization from supermodules (Const. 5.4). Figure 12-c illustrates this case. It creates a cycle in the specialization relation which is no longer a partial order, but only a preorder. A natural solution would be to forbid such conflicts, which means that for each module $\mathcal{M} \in X^{\mathcal{P}}$, the part of $\prec^{\mathcal{M}}$ declared by the programmer should not entail such cycles in conjunction with the part inherited from supermodules by Constraint 5.4. This is, however, not enough since the cycle may follow only from the specialization inherited from two unrelated supermodules, as in the example. Moreover, such a cycle does not only appear in an explicit module—it could also occur in the implicit *bottom module*. Hence, if cycles are forbidden, modules that yield them cannot be gathered in the

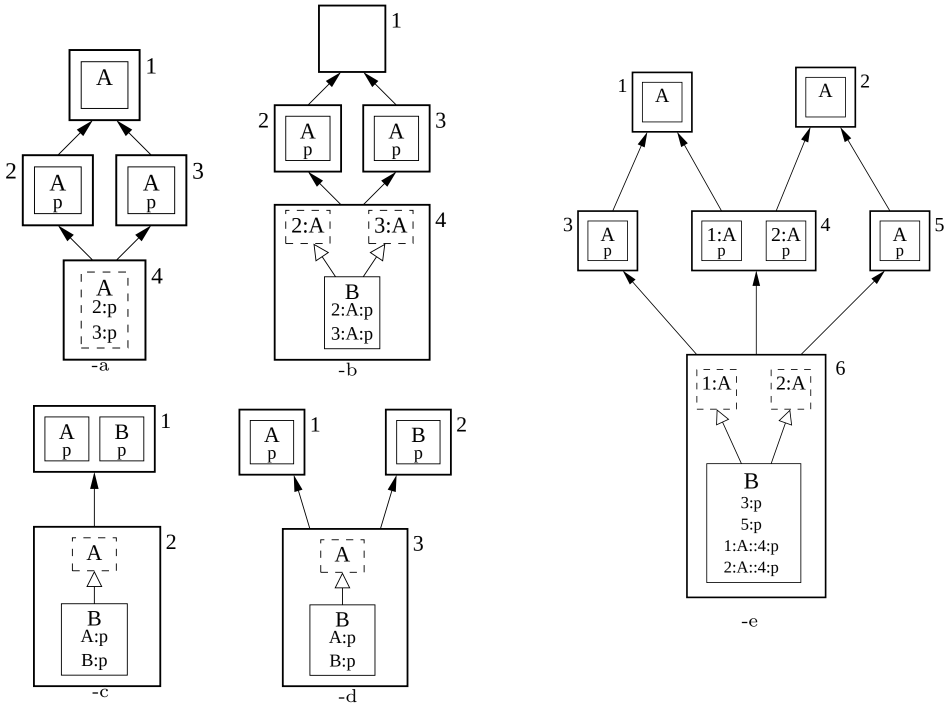


Fig. 13. Global Property Conflicts

same program.

An alternative involves considering that a cycle in the specialization relationship is an equivalence class, since specialization implies inclusion of instance sets and property sets. Therefore, the solution would be to unify all classes on the cycle. Moreover, class unification could be an explicit feature of class refinement languages. We shall no longer formalize this idea, which seems quite feasible. Of course, variants of the usual property conflicts might occur when a name denotes several global properties in the cycle classes, and when several local properties are defined in the cycle classes, for the same global property.

5.3.3 *Global Property Inheritance.* Like usual classes, a local class resulting from specialization and refinement consists of sets of global and local properties.

A local class has the global properties of the classes that it specializes and refines (21-23). This entails the usual global property conflicts (Def. 3.4) together with their generalization to refinement, by replacing $\prec_d^{\mathcal{M}}$ by $\ll_d^{\mathcal{P}}$, or by union of both. In Figure 12-b, A_2 and A_3 inherit the global property `p` introduced into A_1 while A_4 inherits the global properties `p`, `q` and `r` introduced into the classes it refines. Therefore, global property inheritance behaves, with modules and class refinement, as for usual class hierarchies. Technically, given the bottom module $\perp$, each global class constitutes a class hierarchy:

Definition 5.9 (Global class hierarchy). Let $lc \in L^{\perp}$ be a local class in the bottom module and $gc = glob_{\mathcal{P}}(lc)$ be its corresponding global class, then

$$\mathcal{H}_{gc} \overset{\text{def}}{=} \langle glob_{\mathcal{P}}^{-1}(gc), \ll^{\mathcal{P}}, G_{lc}^{\perp}, L_{gc}^{*}, N^{\mathcal{P}}, name_{\mathcal{P}}, glob^{*}, intro_{\mathcal{M}}, def^{*} \rangle$$

is a class hierarchy, where $L_{gc}^{*} = \bigcup_{lc \in glob_{\mathcal{P}}^{-1}(gc)} L_{lc}^{def_{\mathcal{P}}(lc)}$ is the set of all local property definitions in the local classes of gc, and $glob^{*}$ and def^{*} are defined accordingly.

There is no need to go into further detail. Just note that these hierarchies are somewhat dual of modules—i.e. intra-module specialization-based inheritance must be added here to usual inheritance, whereas, in a module, extra-module refinement-based inheritance must be added.

So global property conflicts are still possible for specialization, in each module, and for refinement, in each global class hierarchy. They must be individually solved by the techniques of Section 4.1. Figure 13-a describes such a conflict, where conflicting properties named `p` are introduced in two local classes corresponding to the same global class `A`—hence, they must be disambiguated by the module name, e.g. by `2:p` and `3:p`. Of course, a global property conflict originating from conflicting global classes requires double qualification—e.g. `2:A:p` and `3:A:p` in Figure 13-b. The global property conflict can also originate from an extra specialization link, as a combination of specialization and refinement (Fig. 13-c and 13-d). Finally, in the worst case, fully qualified local classes may be required for disambiguating property names, when both global property and global class conflicts occur—e.g. `1:A::4:p` and `2:A::4:p` in Figure 13-e. This fully qualified notation corresponds to $name_{\mathcal{Q}}$ (30). Disambiguation is complete as $gid^{\mathcal{P}}$, $lid^{\mathcal{P}}$ and all $gid^{\mathcal{M}}$ are injective.

Global property generalization. However, other apparent conflicts resulting from the combination of refinement and specialization deserve discussion. As the programmer of a module $\mathcal{M}$ knows the supermodules of $\mathcal{M}$, i.e. classes and properties which are defined in these supermodules[42], it seems relevant to also consider subclasses when analyzing class refinement. Thus, in Figure 14-a, when the programmer defines `p` in A_2, it is interpreted[43] as the *generalization* of the property `p` introduced in B_1.

Definition 5.10 (Property generalization). Let g be a global property introduced in some local class in module $\mathcal{M}'$ and a submodule $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$. Property generalization occurs when $id_{\mathcal{M}}^{\mathcal{M}'}(intro_{\mathcal{M}'}(g)) \prec^{\mathcal{M}} intro_{\mathcal{M}}(g)$.

For the sake of simplicity, this feature was not considered in Constraint 5.5, but the modification is straightforward—namely, $intro_{\mathcal{M}}(g) = id_{\mathcal{M}}^{\mathcal{M}'}(intro_{\mathcal{M}'}(g))$ must be replaced by $id_{\mathcal{M}}^{\mathcal{M}'}(intro_{\mathcal{M}'}(g)) \preceq^{\mathcal{M}} intro_{\mathcal{M}}(g)$. During module construction (Def. 5.7, step iv), when defining a local class c, property names must be compared not only to inherited global properties ($G_{\uparrow c}^{\mathcal{M}} \cup G_{\Uparrow c}^{\mathcal{M}}$) but also to global properties introduced in refined subclasses in supermodules, i.e. in $G_{\Uparrow + c'}^{\mathcal{M}}$, for all $c' \prec^{\mathcal{M}} c$. So,

[42] The reader must keep in mind that visibility restrictions are not considered in this paper—in practice, the programmer of $\mathcal{M}$ would only partially see the classes and properties defined in supermodules, and property generalization only applies to the visible ones.

[43] We are only specifying a model, not an actual programming language. In an actual language, specific keywords might be added to ensure that there is no confusion.

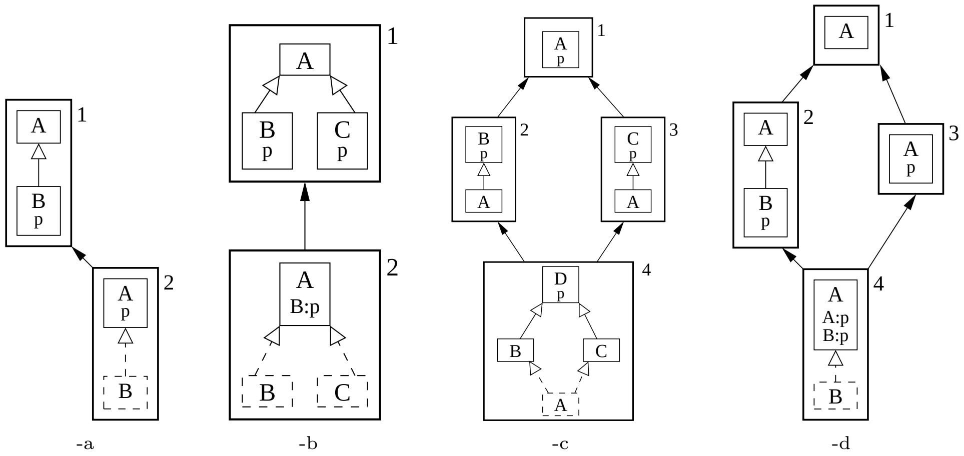


Fig. 14. Global Property Generalization

in practice, when a local property *lp* is defined in *c*, its name must be compared to all of these property sets.

Property generalization can in turn yield some ambiguities akin to global property conflicts. For instance, in Figure 14-b, generalizing `p` in A_2 requires disambiguating it, by `B:p` or `C:p`. An alternative would be to include unification of global properties in our specifications, but we do not consider this extension to our model here. However, unifying both properties would here be questionable since the programmer of module 1 could have done it alone. Figure 14-c depicts another example, where the same property `p` is generalized in two different classes, B and C, of two unrelated modules 2 and 3. Therefore, in a common submodule 4, `p` must be in turn generalized in a common superclass D of B and C—this ensures that `p` is introduced by a single class. Moreover, such a conflict does not appear only in an explicit module—it could also occur in the implicit *bottom module.* However, in both cases, the extra class can be added automatically. Finally, a conflict may occur when the considered name has been independently introduced in refined classes of both superclasses and subclasses (Fig. 14-d).

5.3.4 *Local Property Inheritance.* Local property inheritance can present different kinds of apparent conflicts, when there are several most specific inherited local properties for a given class and a given global property:

(1) usual local property conflict in a single module, when specificity is related to $\prec^{\mathcal{M}}$ and $\ll^{\mathcal{M}}$ (Fig. 7);

(2) the analogue when specificity is related to $\ll^{\mathcal{P}}$ and $\lhd$ (Fig. 15-a);

(3) a special case of the preceding one, where submodule 4 is not explicit (Fig. 15-b);

(4) a mixed situation, when the two conflicting properties are inherited respectively through $\prec^{\mathcal{M}}$ and $\ll^{\mathcal{P}}$ (Fig. 15-c);

In the first two cases, the conflict must be solved in the same way as ordinary local property conflicts, i.e. redefining, selecting and/or combining. In the third

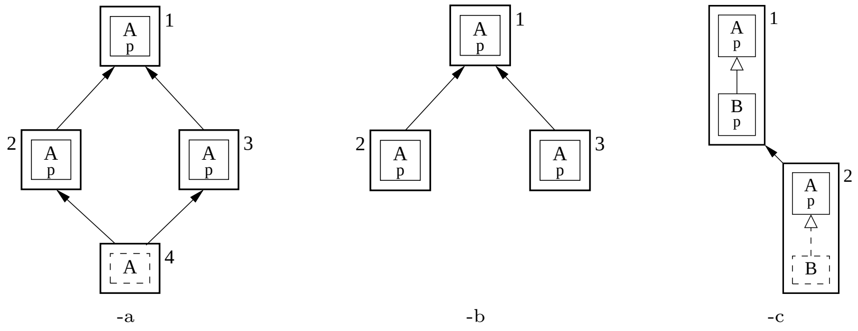


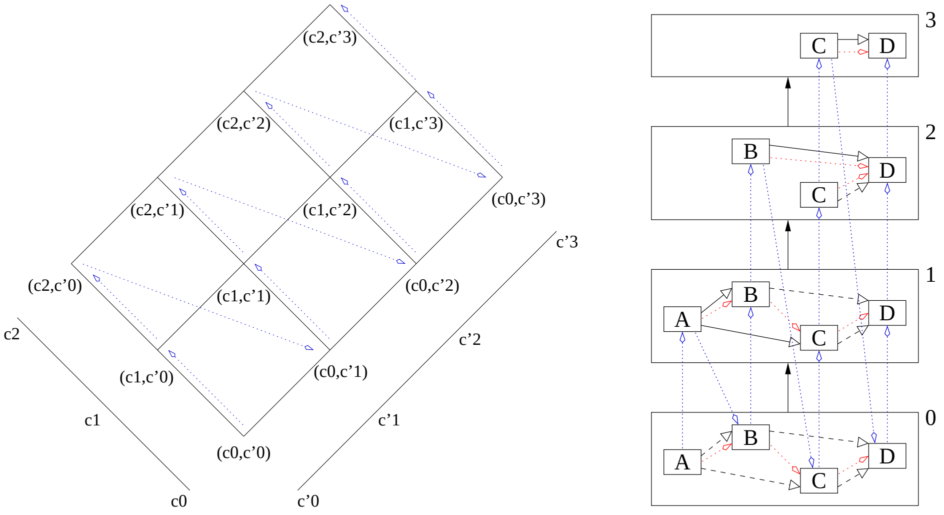
Fig. 15. Local Property Conflicts

case, when modules 2 and 3 are included in some program $\mathcal{P}$ but have no explicit submodule, the conflict must be solved in the implicit bottom module $\perp$. This is finally the only glue code which is required for building $\mathcal{Q}$ and the only case which requires making $\perp$ explicit. The difference is that in case (2), the conflict solution is done when defining and compiling module 4, whereas in case (3) it is done at link-time, when gathering the code of all modules. Figure 15-c illustrates the latter case and raises the question of local property inheritance of the global property `p` in the local class B_2. The local classes B_1 and A_2 are incomparable by $\prec^{\mathcal{M}}$, $\ll^{\mathcal{P}}$ or even combinations of both, but the intuitive vision considering refinement as an incremental modification of classes gives to B_2 the method p defined in B_1. This intuition agrees with the definition of $\mathcal{Q}$, where the local property `p` defined in B_1 is the local property for B (31).

5.3.5 *Call to `super` and method combination.* With class refinement, call to `super` becomes much more ambiguous than usual. It could as well denote a redefined local property in a superclass ($\ll^{\mathcal{M}}$), in a refined class ($\lhd$), or in a combination of both. However, the same approaches as with multiple inheritance are still possible—`call-next-method` and linearizations and qualified or unqualified use of `super`. We do not consider the alternative of static calls.

Bidimensional linearizations. Let us first examine linearizations. The situation is somewhat 2-dimensional, with local properties depending on both specialization and refinement orderings. This resembles a common situation in Clos, when the selection of *methods* (the Clos word for local properties) in a *generic function* (the Clos word for global properties) involves the dynamic types of two parameters. This is an original feature of Clos, also called *multimethods*, which implies selecting methods in the cartesian product of two hierarchies, one for each parameter. Each hierarchy is a partial order and this amounts to single parameter selection in the product of these partial orders[44]. In this situation, Clos linearizes the product of both hierarchies, by linearizing the product of their linearizations. Moreover, among a large set of possible linearizations, Clos selects the following. Let c_0, c'_0 be the dynamic types of the two parameters, $clin(c_0) = (c_0, c_1, c_2, .., c_k)$

[44] The product of two posets $(E, <_E)$ and $(F, <_F)$ is the poset $(E \times F, <_{E\times F})$ defined by $(x, y) <_{E\times F} (x', y') \iff x <_E x' \wedge y <_F y'$.



-a Multiple selection in Clos

-b Specialization and refinement

Fig. 16. Bidimensional linearizations in Clos and when combining class specialization and refinement—linearizations are in dotted lines. The Clos example (left) illustrates the product of two linearizations, (c_0, c_1, c_2) and (c'_0, c'_1, c'_2, c'_3), whereas the class refinement example (right) depicts a module linearization $clin_0^{\mathcal{P}} = (0, 1, 2, 3)$, where 0 is the bottom module, a class linearization in $\perp$, $clin^{\perp}(A_0) = (A_0, B_0, C_0, D_0)$, and the local class linearizations $llin^{\mathcal{P}}(A, \perp) = (A_0, A_1)$, $llin^{\mathcal{P}}(B, \perp) = (B_0, B_1, B_2)$, $llin^{\mathcal{P}}(C, \perp) = (C_0, C_1, C_2, C_3)$ and $llin^{\mathcal{P}}(D, \perp) = (D_0, D_1, D_2, D_3)$. The linearizations are monotonic w.r.t. refinement—i.e. $clin^1(A_1)$ includes $id_1^2(clin^2(B_2))$.

and $clin(c'_0) = (c'_0, c'_1, c'_2, ..., c'_{k'})$. Then, the linearization of the product is the ordering $((c_0, c'_0), .., (c_k, c'_0), (c_0, c'_1), .., (c_k, c'_1), .., (c_0, c'_{k'}), .., (c_k, c'_{k'}))$ (Fig. 16-a).

So, we propose a similar approach for class refinement. Let $c^0 = c_0 \in X_{\perp}^{\mathcal{P}}$ be the dynamic type of the considered receiver, and let the two linearizations be:

(1) $clin^{\perp}(c_0) = (c_0, c_1, .., c_l)$ is the linearization of $supc^{\perp}(c_0)$, the superclasses of c_0 in $(X^{\perp}, \prec^{\perp})$—or, isomorphically, the global classes in $(X^{\mathcal{Q}}, \prec^{\mathcal{Q}})$;

(2) $llin^{\mathcal{P}}(gc, \perp) = (c^0, c^1, ..., c^k)$, with $gc = glob_{\mathcal{P}}(c^0)$ is the linearization of c^0 in $(glob_{\mathcal{P}}^{-1}(gc), \ll^{\mathcal{P}})$, the set of local classes refined by c^0. Note that $llin^{\mathcal{P}}(gc, \perp)$ and $clin^{\mathcal{H}_{gc}}(c^0)$ (Def. 4.2 and 5.9) are total orders on the same set and can be made equal.

In contrast with Clos, we no longer consider the product of linearizations, but the concatenation of successive $llin^{\mathcal{P}}(gc_i, \perp) = (c_i^0, c_i^1, ..., c_i^{k_i})$ where $c_i^0 = c_i$ and all $gc_i = glob_{\mathcal{P}}(c_i^0)$ are ordered by $clin^{\perp}(c_0)$. Therefore, the resulting linearization is $(c_0^0, c_0^1, .., c_0^{k_0}, c_1^0, .., c_1^{k_1}, .., c_l^1, .., c_l^{k_l})$ (Fig. 16-b). Once again, this ordering is consistent with the definition of $L^{\mathcal{Q}}$ (31), hence linearization can serve for selection as well.

This new framework raises multiple monotonicity issues. (i) With modules, a programmer is in charge of a whole module, not of a single class—so, he/she can modify this module until the required monotonicity is reached. Therefore, monotonicity would never be a problem if modules were isolated. However, a module depends on some other modules and (ii) specialization import entails a new kind of monotonicity, which amounts to importing linearizations—namely, if $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$, $clin^{\mathcal{M}}(id^{\mathcal{M}'}_{\mathcal{M}}(c))$ must include $id^{\mathcal{M}'}_{\mathcal{M}}(clin^{\mathcal{M}'}(c))$. Though the only required linearization is $clin^{\perp}$, the programmer does not foresee this bottom module when he/she develops module $\mathcal{M}$ but his/her contribution to $clin^{\perp}$ is the definition of $clin^{\mathcal{M}}$. So refinement-based monotonicity is as desirable as usual, i.e. specialization-based, monotonicity, and both combine for classes introduced in $\mathcal{M}$—i.e. if $c' \prec^{\mathcal{M}} id^{\mathcal{M}'}_{\mathcal{M}}(c)$, then $clin^{\mathcal{M}}(c')$ must include $id^{\mathcal{M}'}_{\mathcal{M}}(clin^{\mathcal{M}'}(c))$. Figure 16-b exemplifies this in a simple diamond case. Thus, multiple import makes monotonicity as hard to meet as with usual multiple inheritance hierarchies and partial linearizations $clin^{\mathcal{M}}$ are again better, since they are less constraining for future submodules. (iii) On the import dimension, the monotonicity of $clin^{\mathcal{P}}$ is also desired and partial linearizations can also be considered. However, $clin^{\mathcal{P}}$ and $llin^{\mathcal{P}}$ are isomorphic, because of empty local classes, so partial linearizations must be considered at the local property level, i.e. in $llin^{\mathcal{H}_{gc}}$ (Def. 5.9)—two $\prec^{\mathcal{P}}$-unrelated modules must be ordered iff both define a local property for the same global property and class couple.

Qualified `super`. To overcome potential linearization drawbacks, we also examine a qualified use of `super`, with two different usages. Let lp be the considered local property which is calling `super`, $gp = glob(lp)$ the corresponding global property, $lc = def(lp)$ the local class which defines lp, $gc = glob_{\mathcal{P}}(lc)$ the corresponding global class, and $\mathcal{M} = def_{\mathcal{P}}(lc)$ the current module, where lc and lp are defined:

(1) Let $\mathcal{M}'$ be a supermodule of $\mathcal{M}$: `super`$\langle\mathcal{M}'\rangle$ calls the local property belonging to gp and defined in, or $\ll^{\mathcal{P}}$-inherited by, module $\mathcal{M}'$; this is not exactly a static call because the local property can be defined in a supermodule of $\mathcal{M}'$;

(2) Let B be the name of a superclass lc' of lc in $\mathcal{M}$, and gc' the corresponding global class: `super`$\langle$`B`$\rangle$ calls the local property defined, in $\mathcal{Q}$, 'in the direction of' gc', i.e. in the most specific class gc'' such that $gc \prec^{\mathcal{Q}} gc'' \preceq^{\mathcal{Q}} gc'$ or $gc \prec^{\mathcal{Q}} gc' \preceq^{\mathcal{Q}} gc''$.

In both cases, the compiler can statically check for the local property. With `super`$\langle\mathcal{M}\rangle$, it can also check for uniqueness, like `Precursor` in EIFFEL. However, with `super`$\langle$`B`$\rangle$, uniqueness cannot be checked at compile-time, since gc' or other conflicting classes can be introduced in submodules. Therefore, a robust specification of this feature might use a linearization of the superclasses of gc which are also sub- or super-classes of B. This resembles the *point of view* notion proposed by Carré and Geib [1990].

Unqualified use of `super` would be rather unsafe, since there is no way to ensure, in the current module, that a class refinement will not yield a conflict.

Example. Let us consider the very simple example of Figure 15-c. Suppose that `B`$_1$`::p` (resp. `A`$_2$`::p`) calls `super` or `super`$\langle$`A`$\rangle$ (resp. `super`$\langle$`1`$\rangle$). Then, in the final program, `B::p` (resp. `A::p`) will be `B`$_1$`::p` (resp. `A`$_2$`::p`) and will call `A::p=A`$_2$`::p` (resp. `A`$_1$`::p`). When substituting `call-next-method` to `super`, the two-dimensional linearization (B_2, B_1, A_2, A_1) would lead to the same result.

Mixin-like modules. We do not consider alternatives such as mixins, since we excluded them when discussing multiple inheritance. Let us just note, however, that the analogy between class level and module level can also apply to mixins. So an analogous definition of mixin-like modules would be straightforward.

5.4 Static Typing

Modules and class refinement slightly changes the point with static typing. Above all and contrary to classes, modules are not types (see Section 5.1.3). Hence, they are not directly concerned by subtyping and one must only examine the effects of modules and class refinement on ordinary types: (i) compilation units are now modules instead of classes; (ii) inheritance import is related to subtyping; (iii) class refinement also induces extra property conformance and changes the way instance construction is done.

5.4.1 *Modular compiling and global linking.* The overall proposition involves a subtle balance between closed and open world assumptions. Each module is compiled under the OWA, irrespective of the way it will be used and refined in submodules. In contrast, modules are linked together to form the program $\mathcal{Q}$ under the CWA.

Modular compiling. The fact that compilation units are now modules instead of classes has only a few effects. Actually, modules cannot pretend to solve any problem occurring in usual class hierarchies since any class hierarchy can be canonically transformed into a module hierarchy, by simply defining a module per class and translating multiple inheritance into multiple import. However, in practice, each module can introduce a whole set of related classes which are defined in a consistent way by a single programmer. Coupled with partial linearizations, this leads us to expect, for instance, that monotonicity will be easier to meet than with usual class hierarchies and total linearizations. From a type perspective, in a module $\mathcal{M}$, the class hierarchy $(X^{\mathcal{M}}, \prec^{\mathcal{M}})$ is globally and statically known, as all $clin^{\mathcal{M}}$ and $llin^{\mathcal{M}}$ linearizations within the module. Hence, the type of $cnm^{\mathcal{M}}(l)$ is statically known. However, method combination also implies refining classes in submodules. Even if refinement is type invariant, a submodule may define a method in a class which only inherits it in $\mathcal{M}$ and supermodules—e.g. in Figure 16-b, the method introduced in D_3 may be redefined in B_2 and C_1, making `call-next-method` with explicit parameters in B_2 unsafe if redefinition is not type invariant (including virtual types). On the contrary, `call-next-method` without explicit parameters remains type safe in a type unsafe covariant framework.

Modular compiling can also help to handle *type conflicts* (Section 4.4.3). This is an error that an intersection type $t \cap t'$ has no corresponding class, e.g. when the two classes t and t' have no common subclass. However, with usual class hierarchies under the OWA, it is not possible to forbid this situation since all classes are not known. With modular compiling, it may be sensible to require such common subclass to be known in the current module. This somewhat amounts to mixing CWA and OWA. The relation between modules remains under the OWA, hence compiling a module does not make any assumption on the way this module will be used and reused. In contrast, the relation between classes inside a module might be ruled by the CWA—a module should be a consistent whole.

Global linking. At end, modules are linked under the CWA. Therefore, it should be theoretically possible to check whether expressions which had been considered as unsafe at compile-time are still unsafe. It only requires that modular compiling keeps some trace of the unsafe expressions. THis will be examined in Section `para:mcgo`.

5.4.2 *Types and subtyping.*

Classes vs. types. In Section 4.4, we placed ourselves in a framework where classes were identified to types, and specialization to subtyping. Now, modules involve two kinds of classes, hence two kinds of types. However, there is no need to formalize these two kinds. When statically analyzing a module $\mathcal{M}$, each occurrence of a type (class name) in $\mathcal{M}$ is considered as the corresponding local class in $X^{\mathcal{M}}$ and the subtyping relationship is identified to $\prec^{\mathcal{M}}$. It follows from specialization import (Const. 5.5) that all subtyping relationships which are valid in some module remain valid in all submodules. So, modular compile-time type checking remains unchanged and all class names can be interpreted as the corresponding classes in $X^{\perp}$ or, equivalently, in $X^{\mathcal{Q}}$. At end, when modules are linked together to form a program ($\mathcal{Q}$), all class names reference the corresponding classes in $X^{\perp}$, and the subtypings of all modules are still valid in the final program.

Property Conformance w.r.t. refinement. Both metamodels are fully compatible with all subtyping policies—usual type safe policy as in most languages, or covariant policy as in Eiffel or Beta.

However, the point is that refinement slightly changes the way the conformance of a property must be checked. Usually, given the redefinition $lp' \ll^{\mathcal{H}} lp$, one must check that lp' conforms to lp, e.g. by respecting the contravariance rule. Refinement imposes the same conformance check with the redefinition $lp' \lhd lp$. But this is not enough—one should also check that lp' conforms with the definition of the considered global property in the subclasses of the considered class. In the example in Figure 15-c, it must be checked that the local property `p` defined in B_1 and inherited by B_2 conforms with the one defined in A_2. This must be checked when defining A_2, by looking in its *refining subclasses*—i.e. in the subclasses of A_2 which refine subclasses of A_1. Note that even with an invariant parameter type policy, such a check is required since the point may be the return type, not only the parameter types.

Nevertheless, the property conformance associated with class refinement is unclear. Class specialization entails the aforementioned covariant vs. contravariant controversy and there are strong but contradictory arguments in favor of both covariance and type safety (see Section 4.4). But refinement is *not* specialization, and this is unclear whether arguments in favor of covariant specialization still hold for refinement. Overall, though other type policies should also be investigated, it might be better to adopt a type invariant policy w.r.t. refinement—thanks to virtual types, this was also the conclusion about specialization. The point with virtual types similar. Of course a final virtual type $T = D$ cannot be redefined in a refining class. Should we allow it for $T <: D$? It may entail unrecoverable errors at link-time, if two modules with conflicting refinements are linked together. Hence, a conservative policy would be to keep also virtual types invariant w.r.t. refinement.

5.4.3 *Instantiation.* Instances only appear at run-time, i.e. in $\mathcal{Q}$. So, strictly speaking, local classes have no direct instances and the code in modules can only create instances of global classes. This has some effects on the notions constructors and abstract classes.

Constructors. Usually, in statically typed languages without class refinement, polymorphism does not apply to instance *constructors* (see also Note 11). The used static type predetermines the dynamic type of the created instances. Thus, in the language specifications, the particular methods with instance construction roles are either not inherited (JAVA or C++) or inherited but without their instance construction roles (EIFFEL).

With class refinement, it is a little bit different since the dynamic type of the new instances (i.e. their *global type*) is statically unknown—indeed, the local classes statically handled by the module can be refined in possible submodules. Therefore, the constructor which will be actually invoked is unknown. Thus, on the one hand, constructors must be fully $\ll^{\mathcal{P}}$-inheritable and $\lhd$-redefinable during class refinement, while on the other, the refining classes must make sure that the constructors introduced into the refined classes remain coherent—possibly by redefining them.

Actually, a similar issue occurs with generics. With homogeneous implementations (see Note 26, page 32), e.g. in JAVA 1.5, `new T` is prohibited when `T` is a formal type parameter, since it has been erased. With heterogeneous implementations, or hypothetical midterm implementations, `new T` is accepted. However, as the usual C++ or JAVA syntax confuses the name of the class and the name of the constructor, and because constructors are not inheritable, the programmer is required to check that a constructor with the precise signature has been defined in the considered class.

Finally, the reader must remember that there is no constructor at the module level because modules are not classes—i.e. they have no instance and no existence at run-time (Section 5.1.3).

Abstract classes. A related topic concerns *abstract* (aka *deferred*) classes, i.e. classes which cannot be instantiated, and their complement, *concrete classes.* Usually, an abstract class is either explicitly declared as such, by some keyword `abstract`, or it has at least one *abstract method*, i.e. a method without implementation. Conversely, a class with a constructor is presumed to be concrete. In common object-oriented languages, there is no need for a neutral position.

Actually, class refinement changes many things. First, it yields a distinction concerning abstract methods—their implementation can be provided by further refinement or subclassing. In the first case, the class can be concrete, but only in submodules. So, the language might offer two distinct keywords to declare a method abstract. The second point concerns the refinement of abstract classes. First, can an abstract class be refined as a concrete one? This would be possible but it does not seem necessary, since the abstract class is not instantiated in supermodules—hence, submodules can ever specialize it for instantiation. Conversely, can a concrete class be refined as an abstract one? This would be unsafe, since the concrete class can be instantiated in supermodules. So concrete is a dominant feature, which cannot be overridden in further refinements. Hence, abstract methods cannot be defined in concrete class refinements.

Let us now consider the possibility of a neutral position. A *neutral class* would have no constructor and no abstract method. It could be refined by either an abstract or a concrete class. However, this would yield a new kind of conflict, when both abstract and concrete refining classes are imported by some common submodule. Of course, the only way to solve this conflict is to consider that the resulting class is concrete. Altogether, it is simpler to consider that neutral classes are abstract, and that abstract classes can be refined by concrete classes. Of course, this forces the programmer to give, in the refining class, concrete definitions for all abstract methods defined in the refined abstract class.

6. A CASE STUDY: PRM AND ITS MODULAR COMPILER

PRM is a modular and object-oriented programming language designed for (i) testing a new compilation scheme, (ii) serving as a testbed for testing various object-oriented implementation and optimization schemes. Originally, we planned to reuse an existing language and its compiler, namely EIFFEL and SMART EIFFEL [Zendra et al. 1997]. As it turned out that it was almost as difficult as starting from scratch, we finally decided to develop a new language, while adding two goals: (iii) a clean specification of multiple inheritance, by implementing the meta-model, and (iv) modules and class refinement. The latter were soon found to be necessary to fulfil (ii). So, PRM is a fully-fledged object-oriented programming language, with static typing, multiple inheritance (mostly conformable to Section 4), genericity, modules and class refinement (mostly conformable to Section 5). There is no place here to present the language in detail, so the reader is referred to [Privat 2006b; Privat and Ducournau 2005] and [Privat 2006a, in French].

This section first presents the original compilation scheme which yields an efficient implementation of module and class refinement. Then the modular architecture of the PRM modular bootstrapped compiler is sketched, as a real-size example of how to use modules and class refinement.

6.1 Modular compilation and global optimizations

The notion of module is closely related to modular separate compilation which operates under the OWA. Each module must be statically analyzed and compiled irrespective of possible future usages. However, object-oriented languages present an original implementation issue, related to late binding and worsened by multiple inheritance. It makes the CWA and global compilation far more efficient, as the knowledge of the complete program allows the compiler to apply many optimizations like type analysis, dead code elimination and *devirtualization* of method calls. For instance, the aforementioned SMART EIFFEL compiler is based on such optimizations. For an analysis of object-oriented implementations, the reader is referred to [Ducournau 2002a]. So reconciling modular compilation with global optimizations looks like squaring of the cycle—appealing but likely difficult. Actually, some first attempts had already been made in a functional programming framework [Boucher 2000].

Figure 17 depicts the overall scheme. Each module is separately compiled and this compilation yields three different results: (i) the *external schema* roughly corresponds to the model of the module, as defined in Section 5, (ii) the *internal schema* describes the flow of types inside the module methods, (iii) the generated

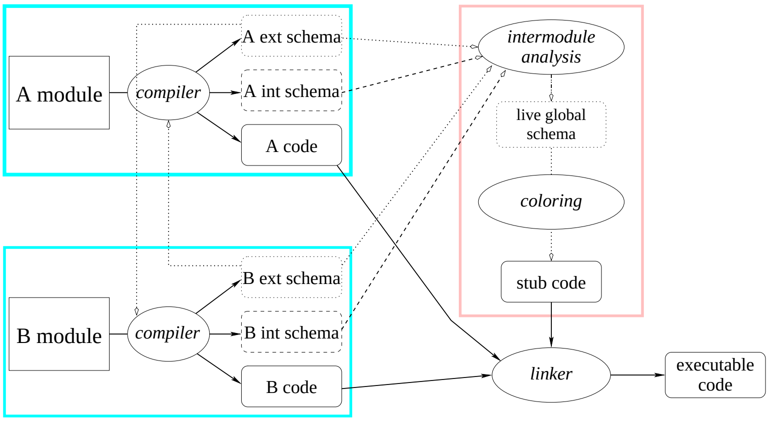
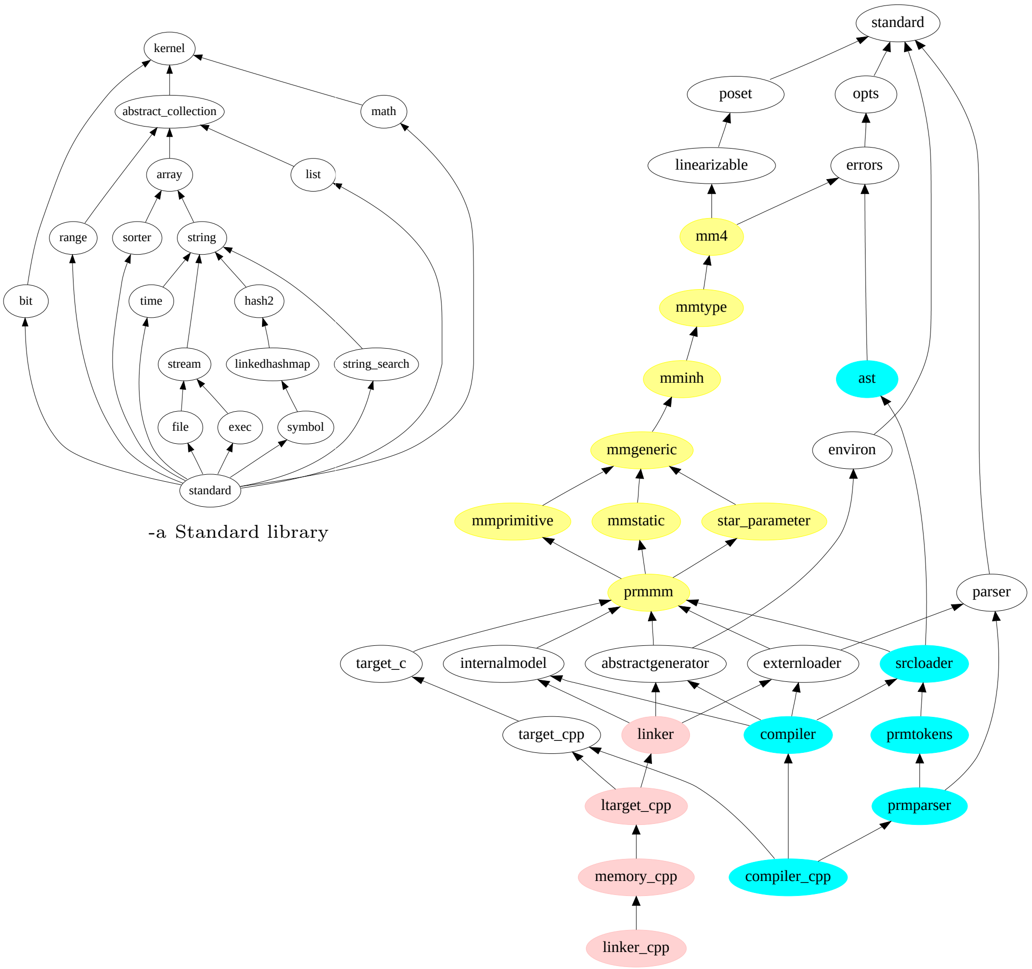


Fig. 17. The PRM compilation scheme, with local modular separate compilation phases (left) and a final global linking phase (right)

code—in some target language, namely C in the PRM case—is quite common but it includes more unresolved symbols than usual. So a module delivery consists of both external and internal schemas, plus the machine code file produced by the final `gcc` compilation. An additional benefit of separate compilation is that the source code does not need to be included in the delivery. Compiling a module generally requires some information about related modules—this information is contained in the external schema, which therefore stands for the module interface (or `.h` files in C++). In the module definition (Section 5), the dependence relationship is constrained to be acyclic. This is, however, a simplifying assumption only required for the relationship which supports class refinement and class specialization, and modules could be 'related' without being 'dependent'. Figure 17 illustrates such a cyclic relationship, with each module requiring the external schema of the other module. This is of course not a problem, as generating the external schema does not depend on any other module. Once all modules are separately compiled, the global phase generates the executable. This global phase first gathers all external schemas which represent $\mathcal{P}$, yielding a *global schema*, which is equivalent to $\mathcal{Q}$. An inter-module analysis based on the *internal schemas* produces the *live* global schema. In this live global schema, possible conflicts may force the programmer to make the *bottom module* explicit and insert some glue code.

An implementation technique is then applied to this global schema. For each class in $\mathcal{Q}$, It must compute the object layout, generate indexes and method tables, if any, and so on. Various techniques can be considered, like *binary tree dispatch* as in SMART EIFFEL, or *coloring* [Ducournau 2006]. Anyway, for each method call site, a small stub function (aka a *thunk*) is generated which implements late binding according to the considered technique and call site. The approach is also used for accesses to attributes and subtype tests. Finally all module target codes, which have been previously separately compiled into machine code, and stub functions are linked together.

With this compilation scheme, class refinement only involves computing $\mathcal{Q}$, i.e. flattening the global schema coupled with method tables (or any alternative) which



-a Standard library

-b PRMC architecture

Fig. 18. PRM standard library and PRMC modular architecture. Both charts are generated by PRM `dot` compiler, another tool not presented in the diagram.

are filled with addresses of the right local properties. Method combination makes it slightly more complicated—when a global property g uses `call-next-method`, for each $c \prec intro(g)$, a local property linearization $llin(g, c)$ is computed as a static array which is passed to methods as an extra parameter.

Altogether, the implementation of class refinement does not entail any run-time overhead—the run-time implementation and code are almost the same as for an ordinary class hierarchy and the PRM compilation scheme has been showed very efficient in previous tests [Privat and Ducournau 2005]. Moreover, the cost of multiple inheritance and method combination is only supported by classes and methods that use these features.

```
abstract class Named {
    String name;}

abstract class Aentity
         <E extends Aentity<E,G,L>, G extends Aglobal<E,G,L>, L extends Alocal<E,G,L>>
         extends Named {
    Set<G> knows;
    Set<L> defines;
    Set<E> spec_d;}

abstract class Aglobal
         <E extends Aentity<E,G,L>, G extends Aglobal<E,G,L>, L extends Alocal<E,G,L>>
         extends Named {
    E intro;
    Set<L> localset;}

abstract class Alocal
         <E extends Aentity<E,G,L>, G extends Aglobal<E,G,L>, L extends Alocal<E,G,L>>
         extends Named {
    G glob;
    E def;
    Set<L> redef_d;}

abstract class Aclass extends Aentity<Aclass,Gproperty,Lproperty>{}
class Gproperty extends Aglobal<Aclass,Gproperty,Lproperty> {}
class Lproperty extends Alocal<Aclass,Gproperty,Lproperty> {}

class Module extends Aentity<Module,Gclass,Lclass> {}
class Gclass extends Aglobal<Module,Gclass,Lclass>{}
class Lclass extends Alocal<Module,Gclass,Lclass> {}

class Mclass extends Aclass, Lclass {}
```

Fig. 19. Definition of the metamodel classes, in module `mm4`

6.2 PRMc modular architecture

PRMc is made of two main executables, which correspond to both phases in the compilation scheme (Fig. 17)—namely, the *modular compiler* and the *global linker*. So the PRM code which implements the whole is a single module hierarchy $\mathcal{P}$, and each program corresponds to a subhierarchy, i.e. to a single bottom module and all its dependent modules. This is, however, just a partial view on PRMc. As the first goal of PRM is to test various implementation techniques, each program is proposed in several versions, with each version corresponding to a specific bottom module.

Figure 18 depicts the module hierarchy dedicated to the PRMc version which generates C++ code. The diagram highlights three groups of modules:

(1) yellow (light grey) modules, from `mm4` to `prmmm`, represent the PRM metamodel, including types, inheritance, generics and so on; this group is of course common to all PRM tools;
(2) cyan (dark grey) modules, from `ast` to `compiler_cpp` represent the PRM to C++ compiler, with its parser;
(3) pink (middle grey) modules, from `linker` to `linker_cpp` represent the PRM linker, in the C++ version;
(4) other modules have various specific technical roles such as poset implementation, abstract code generation and so on.

6.2.1 *Metamodels.* The modules for the metamodel are quite numerous, almost more numerous than the corresponding classes. This was, of course, not required,

Table I. Statistics on class and property—numbers of global class introductions, class refinements and global property introduction.

module	C. intro	C. ref	P. intro	module	C. intro	C. ref	P. intro
standard	84	0	413	ast	85	0	213
opts	9	1	68	parser	3	1	19
poset	2	0	38	target_c	2	21	52
linearizable	5	3	21	srcloader	21	64	131
errors	4	2	16	abstractgenerator	2	0	10
mm4	10	0	131	prmtokens	0	1	88
mmtype	6	3	45	target_cpp	0	4	3
mminh	6	9	59	compiler	2	11	26
mmgeneric	2	10	49	prmparser	22	75	107
mmprimitive	8	4	5	compiler_cpp	1	66	38
star_parameter	0	3	7	linker	10	5	45
mmstatic	2	5	5	ltarget_cpp	0	17	32
prmmm	2	3	33	memory_cpp	0	2	5
internalmodel	2	0	17	linker_cpp	0	0	0
				total	290	310	1676

since all Prm programs are based on the same metamodel, but it was a convenient way to separate concerns. So each module is rather small and dedicated to a single concern: `mm4` introduces the main classes for the metamodel, and other modules refines them for introducing types and signatures (`mmtype`), inheritance and import (`mminh`), generics (`mmgeneric`), and so on. Regarding generics, the language will implement full F-bounded polymorphism, like Java 1.5, likely with an implementation which will keep the advantages of both homogeneous and heterogeneous implementations (see also Note 26, page 32). However, in the current state of the compiler, generics are handled in a coarse way that does not allow us to directly implement the metamodel isomorphism with generics. F-bounded polymorphism is a powerful tool for implementing isomorphic models and, in further versions, the basic metamodel classes will be defined by the code in Figure 19, which is presented with a more familiar Java syntax. `Aentity`, `Aglobal` and `Alocal` define the abstract triangle common to both metamodels and implements the whole protocol of Definition 3.10. `Aclass`, `Gproperty` and `Lproperty` define the metamodel of Figure 4, and `Module`, `Gclass` and `Lclass` the metamodel of Figure 11. Finally, `Mclass` is the class of local classes as they are defined in module definitions, at compile-time. In the linker modules, `Gclass` is refined to inherit `Aclass`—this defines the class of classes in $\mathcal{Q}$. All are legal Java definitions, except the last one which involves multiple inheritance.

6.2.2 *Compiler and linker.* The compiler modules involve first a set of modules dedicated to parsing, among which the *abstract syntax tree* of the language (`ast`), and a set of modules dedicated to code generation, either independent from the target language (`abstractcompiler`, `compiler`) or dedicated to C (`target_c`) and C++ (`target_cpp`, `compiler_cpp`). Whereas usual compiler architectures are based on different variants of the *Visitor* pattern (e.g. [Nystrom et al. 2003]), PrmC uses class refinement, which makes it very easy to add methods to the AST classes, e.g. in modules `srcloader`, `prmparser` and `compiler_cpp`, which all have a high number of class refinements (Table I). The linker also presents general modules (`linker`) and modules dedicated to C++ (from `ltarget_cpp` to `linker_cpp`).

7. RELATED WORKS

We group related works according to whether they are markedly different or similar to our proposal. [Bergel et al. 2006] is a survey of the notion of modules in an object-oriented framework. Class refinement corresponds to *local rebinding* in their terminology. So the first group presents different 'module' systems that we consider are not very close to our proposal, and the second one presents module systems with some form of class refinement.

7.1 Generally related works

This quick survey is not meant to be exhaustive—there are many object-oriented module systems.

Java *packages and inner classes.* Java proposes two different features which are closely related to the notion of modules and give good examples of what a module is or is not [Gosling et al. 2005]. Java *packages* represent both name-spaces and directories and their hierarchical organization is also used for finding a class along a class path. These packages have almost nothing to do with our modules, since they are unclosed—any programmer can add classes in any package—so they are not units of compilation and reuse. Moreover, there is nothing close to class refinement. *Package sealing* could address the former point and has been discussion issue in the Java community (e.g. [Biberstein et al. 2001]). On the contrary, *inner classes* make usual, i.e. enclosing, classes true modules. However, an inner class can be specialized but not refined.

Uml *package merge.* This mechanism is proposed by Uml 2.0 [OMG 2004]. It permits an incremental definition of packages, independently from package import. As for multiple inheritance, multiple merge is not clearly specified and inherent conflicts are not discussed. So the intention of Uml specifications is likely close to modules and class refinement, but it is too underspecified to allow any precise comparison. Clark et al. [2002] propose a metamodel for package extension with renaming but its primary postulate is that "on package extension, the default is for elements of the same name to get merged". Local renaming is possible but it is as unconstrained as in Eiffel.

Other module systems for object-oriented languages. Many module systems have taken Modula-family languages as a model. However, in Modula-3 [Harbinson 1992] and Oberon [Mössenböck 1993], modules do not accept class refinement.

Several works propose module systems for object-oriented languages where the dependence relationship is only implicit and derived from explicit class import clauses—e.g. JavaMod [Ancona and Zucca 2001] which accepts to specialize an imported class in a mixin-like way. Ocaml modules are similar, but the dependence relationship is expressed with signatures and module types [Leroy et al. 2004]. However, in all of these module systems, there is no trace of refinement.

Some other module definitions are substantially out of line with our view on modules. For instance, in the Jigsaw programming language [Bracha 1992; Bracha and Lindstrom 1992], the primary construct is the 'module' and inheritance is characterized as a module manipulation mechanism. This however appears to be a misnomer—i.e. these modules can be understood as classes or mixins.

Aspect-oriented programming. AOP is a general fine-grained answer to the cross-cutting concern issue [Kiczales et al. 1997]. As such, one might consider that modules and class refinement present a form of aspect-oriented programming[45]. Modules would be aspects—which can also be organized by a dependence relationship, called `extends` in AspectJ—and refinement would be a way to add some cross-cutting aspect to the refined class. However, we do not take up this position because we consider that AOP focuses on concerns of a much finer grain. Class refinement works at the class-property interface—it proceeds by defining local properties, which are considered as atoms in the metamodel. On the contrary, AOP is intended to work inside the methods, or even inside the basic object-oriented mechanisms, such as method invocation or class instantiation, by specifying *pointcuts* which provide the programmer with some hooks for aspect-specific code. However, it might be possible that some AOP proposal includes higher-level mechanisms similar to class refinement. Actually, an AOP system like AspectJ [Kiczales et al. 2001] provides some atomic primitives which allow the programmer to modify 'imported' classes in a way similar to refinement, but this is much more akin to metaprogramming than to common object-oriented programming. Of course, possible conflicts are not considered. To our knowledge, there is no proposal which combines fine-grained AOP with some class-level refinement closer to our proposal.

Reverse inheritance. A notion akin to *property generalization* has been proposed under the name of *reverse inheritance* [Chirila et al. 2004] or *exheritance* [Sakkinen 2002]. It involves generalizing classes by defining superclasses of preexisting classes. Some properties defined in the generalized classes can move up in the generalizing class. However, there is no module in the proposal, which seems to resort to re-engineering rather than to normal software development.

7.2 Closely similar works

All proposals discussed hereafter involve a notion of modules coupled with some class refinement mechanism.

Modular open classes. MultiJava [Clifton et al. 2000], an extension of Java, proposes compilation units, similar to the modules presented in the present article. They are provided with a dependence relation via the keyword `require`. MultiJava makes it possible to extend existing classes by adding functions by an ad hoc syntax whereas, in opposition to our approach, method redefinition, attribute addition or declaration of Java interface implementation are not allowed. Nevertheless, MultiJava is compatible with separate compilation and dynamic loading. It also proposes an implementation of multimethods.

Classboxes. This notion of 'box of classes' is a kind of module first introduced for Smalltalk [Bergel et al. 2003; Bergel et al. 2005]. *Classbox/J* applies the same approach to Java and static typing [Bergel et al. 2005]. Classes can be extended by adding or redefining methods and attributes in classboxes while controlling the visibility of these additions since these changes only have local impacts: message sending answers are determined by both the receiver and the *classbox*. This method

[45] "[..] it is perhaps hard to find cross-cutting software implementation techniques that would *not* qualify as 'aspect-oriented' " [Smaragdakis and Batory 2002].

dispatch mechanism is called *local rebinding* and seems to yield marked overhead in the present implementation of Classbox/J, since it is akin to Clos multiple selection. Thus, contrary to our proposal, class amendments made in a *classbox* are applied only to this *classbox* and to *classboxes* which depend on it. Hence, message sending from other *classboxes* are not affected by the modification. Another difference is multiple inheritance. Classbox/J allows multiple import but the potential conflicts are not discussed. However, in a conflict—presumed to concern local properties—involving both refinement and specialization, refinement takes precedence over specialization, as in our proposal.

Virtual classes and higher-order hierarchies. Virtual classes have been introduced in the language Beta [Madsen and Møller-Pedersen 1989; Madsen et al. 1993]. They combine *virtual types* and *inner classes* and have multiple usages—e.g. they can serve as inner classes, as formal types in parametrized classes [Thorup and Torgersen 1999], as virtual types or Eiffel anchored types. They can also combine these different usages. As inner classes, they differ from those of Java by their 'virtuality'—like other usages of 'virtual' (see Notes 6, page 9, and 31, page 36) a virtual class is 'redefinable', hence submitted to late binding and depending on the dynamic type of some 'receiver'. They also introduce a form of class refinement and outer classes are similar to modules. However, original virtual classes also have limitations—they cannot be specialized [Ernst 2003]. Another difference with our proposal is that instances of a virtual class and instances of its redefinition are distinct and can simultaneously exist in the same program. *Higher-order hierarchies* provide nested class hierarchies, based on an extension of virtual classes, in the Gbeta language [Ernst 2003; Ernst et al. 2006]. Outer classes are close to our modules, and order 1 inner classes are close to our classes, but the proposal generalizes this at any order. Technically, both approaches are quite different concerning multiple inheritance, which is unavoidable as we have seen, and method combination. Higher-order hierarchies propose a combination of completely ordered *mixins* whereas our proposal relies on the interpretation of multiple inheritance in the metamodel.

Nested Inheritance. This proposal is quite close to our view of modules and refinement, though restricted to single inheritance or mixins [Nystrom et al. 2004]. In a recent work, its extension to *Multiple Nested Inheritance* [Nystrom et al. 2006] follows a similar analysis of multiple inheritance, though the lack of metamodel makes the authors uniformly speak of *name conflicts.* Global property conflicts are solved by fully qualified syntax[46], whereas local property conflicts make the

[46] The authors assign to Borning and Ingalls [1982] the paternity of this 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], the authors consider that their view of multiple inheritance derives from *intersection types* [Compagnoni and Pierce 1996; Reynolds 1996] (see Section 4.4). We did not find, in the referenced papers on

considered class *abstract*—hence, they require further redefinitions. Besides these similarities, there are some differences. Nested inheritance is recursive, like higher-order hierarchies—however, the first two levels are called *packages* and *classes.* It also provides implicit dependence, via *static virtual types.*

Layers, collaboration-based design, feature-oriented programming. Mixins are used in several proposals, in an approach called *collaboration-based design* [VanHilst and Notkin 1996]. *Mixin layers* [Smaragdakis and Batory 1998; 2002] are sets of mixins—akin to modules—which make the composition of the reusable parts much easier. In some sense, they can be understood as higher-order or nested mixins. So, mixin layers are akin to higher-order hierarchies. Actually, 'layer' is another word for 'module', with more operational semantics, i.e. stack-wise or linearized like mixins. Mixin layers have been used for implementing *feature-oriented programming* [Apel et al. 2005], which looks like another synonym for all of these approaches—i.e. AOP at the 'feature' (property in our terminology) level.

Difference-based modules. MixJuice [Ichisugi and Tanaka 2002] is a language based on Java which proposes modules in dependence relation and refinement of classes authorizing method (re)definition, attribute addition and declaration of Java interface implementation. Therefore, it functionally corresponds to adapting to Java points (1-3), Section 2.1. In the case of multiple dependence between modules, global property conflicts are solved by fully qualified names and local property conflicts are solved by linearization. The approach is compatible with separate compilation but not with dynamic loading. So MixJuice is close to our proposal. The main difference is that, due to Java single inheritance of classes, adding superclasses to existing classes is not possible—though this is theoretically possible in the restricted case where the new superclass specializes the old one. Property generalization is also not provided. In contrast with our proposal, MixJuice does not reveal the structural analogy between classes and modules—therefore, their proposition, though very similar to ours—may appear as an ad hoc solution.

7.3 Comparison and discussion

Overall, our proposal is akin to all these proposals based on some module and class refinement notions. Comparing them involves examining the following points:

(1) *Run-time instances.* At run-time, different versions of a class can coexist in the same program, as with *classboxes*, *virtual classes* and *nested inheritance.* On the contrary, in MixJuice and in our proposal, only the most refining classes, and their related instances, remain at run-time.

(2) *Higher-order.* The proposal can rely on two distinct notions, classes and modules, as in MixJuice and our proposal; on the contrary, it can rely on a single,

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 [1984] must also be ruled out—it proposes a theory of record types without classes, hence, without any notion of introduction. Altogether, to our knowledge, this idea of distinguishing properties according to the class which introduces them goes back to some studies on the notion of *point of view*, in a framework of dynamic typing and knowledge representation [Carré and Geib 1990; Dugerdil 1991]. We are not aware of any prior mention of it.

recursive notion of class, like *inner classes*, *virtual classes* or *higher-order hierarchies*. It can also rely on both approaches, with an explicit notion of module and nested classes, like *nested inheritance*.

(3) *Multiple inheritance and import.* The proposal can take full multiple inheritance into account, like *multiple nested inheritance* and our proposal. In a Java multiple subtyping setting, it can take multiple import into account, and deal with conflicts in a sound way, like MixJuice. Finally, it can deal with multiple import and inheritance conflicts in an ad hoc way, e.g. with mixins.

(4) *Metamodeling.* The proposal may rely on an explicit metamodel, or use some other formalizations, e.g. calculi, which rely only on names.

Point (1) makes the proposals functionally different. They do not address the same need. Compared to classboxes, our proposal does not allow the programmer to keep, in the same run-time program, different versions of the same class. Both solutions are actually not comparable. If c refines c' (in our notations, $c \ll^{\mathcal{P}} c'$), having an instance of c' instead of an instance of c at run-time is not a gain in expressiveness but only a different behaviour. Indeed, having only c instances may appear at first glance to be less powerful, but these instances can be created by instructions in the module which defines c'. The other solution is unable to do it, at least directly[47]. The overall complexity can be analyzed in the following way. Suppose that we have a hierarchy of m modules refining the same hierarchy of n classes. In our proposition, the final program contains only n classes, whereas there are mn classes at run-time in the compared propositions—they can be understood as the third view on program semantics (Section 5.1.2), with the essential difference that all local classes would remain concrete.

However, besides the questionable design and the program complexity that such an expressiveness brings, the implementation cost seems too high—i.e. in *Classboxes*, all method invocations are akin to multiple selection on two parameters, the current receiver and the current classbox. This differs markedly from all analyses carried out on multiple selection (e.g. [Kiczales and Rodriguez 1990; Dujardin et al. 1998]), which stress that, in a multiple selection framework, about 95% of method invocations are single selection. In comparison, our proposal incurs no run-time overhead in the Prm compilation framework[48]. However, in return, keeping different run-time versions of the same class is mandatory in a dynamic-loading framework. Indeed, what is the meaning of class refinement for already existing instances? This is a well-known issue in languages like Clos which allow the programmer to dynamically redefine classes.

Point (2) makes our proposition less expressive but simpler, and we promote this simplicity. Modules and classes are distinct notions, with specific roles and,

[47] Of course, this is likely possible in a less direct way. Suppose that modules do have instances, like in the compared propositions. Then, if `new` C were constructing an instance of the local class C of the current module, it would be possible to create an instance of C in the bottom module with `new ctx.`C, where `ctx` would be a global variable typed by the current module and denoting the single instance of the bottom module. Without loss of generality, each module should define such a variable and all of the variables would be inititialized in the bottom module.

[48] We mean here that the compilation framework does not add any overhead when substituting modules to classes as compilation units. When classes are compilation units, the efficiency of the compilation framework have been showed by previous experiments [Privat and Ducournau 2005].

Table II. Comparison. MST (resp. SI, MIX, FMI) stands for *multiple subtyping* (resp. *single inheritance*, *mixins*, *full multiple inheritance*).

	class boxes	higher-order hierarchies	mixin layers	diff.-based modules	nested inheritance	modules & class refin.
diff. run-time instances	yes	yes	yes	**no**	yes	**no**
higher-order	**no**	yes	yes	**no**	yes	**no**
explicit modules	**yes**	no	no	**yes**	**yes**	**yes**
inheritance	SI/MST	MIX	MIX	MST	FMI	FMI
multiple import	**yes**	MIX	MIX	**yes**	**yes**	**yes**
metamodel	no	no	no	no	no	**yes**

though recursive higher-order has a nice theoretical flavour, we do not think that it will help programmers to build comprehensive modular programs. Other proposals are top-down higher-order and classes can be deeply nested. From a historical standpoint, class nesting comes from SIMULA [Birtwistle et al. 1973], the first object-oriented language and a close parent of both C++ and BETA—so, the confusion goes back the very beginning of object-oriented programming. [Szyperski 1992] contains an insightful critique of this confusion between modules and classes (or even types), which is common in many object-oriented module systems, especially with class nesting and even in the MODULA family. A class that contains inner classes can be interpreted either as a module or as a class which implements some aggregation, akin to a container. Only class *leaves*—i.e. without inner classes—are really unambiguous and, among higher-order proposals, only *nested inheritance* provides explicit unambiguous modules. In contrast, our proposal is bottom-up second-order—classes are at ground level and modules are order 2—and we stop there. Of course, metamodel isomorphism could be easily generalized to higher-order, but we think that the improved expressiveness is not really useful and does not offset the lack of conceptual simplicity. Actually, among many other arguments, modules are not classes, given that they do not have instances[49]. Moreover, one should not confuse module dependence with module nesting. Modules—i.e. our view of modules—are separate code units, that must be separately analyzed and compiled. They cannot be nested—we feel that an inner module is nonsense.

The differences can be stated in a different way. Our classes are neither virtual classes nor virtual types. Contrary to properties, they are actually not concerned by late binding.

Multiple inheritance (3) is another distinctive feature. *Multiple nested inheritance*, our proposal and, to a lesser extent, MIXJUICE are based on a sound analysis of multiple inheritance and import. They have the same behaviour at the global entity level. At the local entity level, method combination remains a difficulty and the linearizations used by the different proposals should be compared in detail.

Finally, meta-modeling (4) is the distinctive feature of our proposal. It allows us to get rid of names and instead to consider reified entities. The metamodel is defined at the class and property level to model our intuitive—one might say 'Aristotelian'—view on classes, specialization and inheritance. It it then isomorphically applied to modules, yielding a precise specification—close to multiple nested

[49] Modules and programs have the same dimension—so, they can have a single instance, corresponding to the static data of an execution, this is a kind of *singleton* class. However, if the metaphor is continued, what would be an order 3 class?

inheritance, but not recursive, or close to MixJuice, but applied to full multiple inheritance—based on formal and intuitive arguments. When comparing with *multiple nested inheritance*, which roughly gets to the same conclusion at the expense of intricate formulations on method names, the metamodel is a great improvement.

8. CONCLUSION AND PERSPECTIVES

In this paper, we have proposed two coupled notions of *modules* and *class refinement* as an extension of the object-oriented model, in the particular context of statically typed languages with multiple inheritance. This proposal improves modularity—by grouping in the same module closely related classes—and reusability—by allowing a module to modify classes imported from its supermodules. Hence, this provides a simple solution to the so-called *expression problem*—adding a new property, by refinement, becomes as easy as adding a new class, by specialization [Findler and Flatt 1999; Torgersen 1994; Zenger and Odersky 2004].

Functionally, our proposal is very close to *Difference-based modules* [Ichisugi and Tanaka 2002], which are restricted to Java-like languages and single inheritance. All other related works such as *Multiple nested inheritance* [Nystrom et al. 2006] adds two undesired features, namely higher-order nesting and different instances at run-time. In contrast, our proposal follows the analysis of Szyperski [1992] and strongly stresses the different roles played by classes and modules by avoiding any confusion. We think that this simplicity should help programmers.

The main originality of our proposal is its meta-modeling approach, with a twofold contribution: (i) a class and property metamodel and its application to multiple inheritance, and (ii) its adaptation to modules and class refinement. Although the use of modules and classes are fundamentally different, our proposal is based on a strict structural analogy between these two concepts since they are described by isomorphic metamodels.

The class and property metamodel clarifies the notion of property and its relation to classes and names. Looking back, it is quite strange to note that a whole scientific field had no words for such basic notions. Being the field of 'reification', it is even stranger that there were no objects underlying these missing words. In a closely related field, the ObjVlisp model was a decisive step in the understanding of classes, meta-classes and instantiation [Cointe 1987]. Besides the understanding of the property concept, the first benefit of the present metamodel is to shed light on multiple inheritance, in a way which is in line with both recent and older works [Ducournau et al. 1995; Nystrom et al. 2006]. Regarding multiple inheritance conflicts, the conclusion of this analysis is twofold: (i) *global property conflicts* could be easily solved when they are met, in a modular way, with a very simple syntactic mechanism, i.e. *fully qualified names*; (ii) *local property conflicts* can also be solved either by an explicit modular redefinition, or with a more controversial mechanism, i.e. linearization. Actually, *method combination* is the single remaining issue in multiple inheritance—linearization is a solution, but it is not perfect.

Metamodels enhance the distinction between local and global entities for both properties and classes. This distinction allows consistent conflict management which respects the semantics of class specialization and module dependence. Our proposal requires only a light syntactic addition—i.e. a fully qualified syntax for class and property names and a rudimentary module language to express that a module depends on another module[50]. Moreover, our proposal is truly modular in the sense that any module hierarchy can be linked within a single program, without any need to modify the preexisting modules. The resulting multiple inheritance conflicts can be fixed in the bottom module when they are met at link-time.

The prospects of this work are manifold. First of all, the current model will be intensively used in PRM and PRMC for testing and benchmarking various implementation techniques, with each technique being a separate module. Several extensions must be considered regarding the module system itself. Here we did not examine the question of *visibility* (aka 'protection' or 'static access control'), i.e. an `export` feature. This is a shallow but important feature that must be carefully designed since it merges usual module export with class encapsulation. We do not feel that the keywords commonly used in JAVA and C++ (`private`, `public` and `protected`) are very relevant, besides their numerous ambiguities, and we shall likely adopt the EIFFEL approach which generalizes SMALLTALK encapsulation—see [Ardourel and Huchard 2002] for an analysis of these mechanisms at the class level. Analogous mechanisms should be specified at the module level and the interaction of both levels should be carefully examined. Another point to examine is the possibility of extending module dependence to implicit ones, e.g. by defining module interfaces and true `import` clauses, as in many module systems. There also remains a kind of conflict without a modular solution in the current state of the model—i.e. *specialization conflicts* caused by specialization import (Fig. 12-c). The solution goes through *class unification*, which has to be specified. Of course, all these features must be added in the metamodels and an interesting question concerns the possibility of adding them while preserving the isomorphism. This may, however, be pointless since we claim that modules and classes have different roles—for instance, visibility might be reserved to the module level. Finally, the adaptation of our proposal to a dynamic loading framework remains to be investigated.

Regarding the programming language itself, and the associated class and property metamodel, several extensions could be considered, which surely interfere with class refinement. As aforementioned, method combination is likely the only remaining issue with multiple inheritance. In common statically typed object-oriented languages, it has been poorly specified in comparison with dynamic languages like CLOS. So introducing :`before`, :`after` and :`around` methods—the so-called 'daemons'—in a statically typed language with class refinement could be considered. Linearizations are now well known but they should be made more flexible, (i) by only computing partial linearizations, thus preserving the chances for further monotonicity, (ii) by allowing the programmer to customize them at the class or global property level, (iii) by extending both to bidimensional linearizations. Furthermore, apart from the aforementioned works on GBETA, linearization-based method combination has been mainly applied in a dynamic typing framework—as we have seen, static typing yields interesting and paradoxical conclusions regarding type safety. *Multi-methods* or *multiple selection*, i.e. the dynamic selection of a local property based on the dynamic types of all parameters, could be also considered. There are basically two different ways of doing this. The *overloaded functions* pro-

[50] However, if the required keywords are very few, a complete language might gain by providing more keywords in order to help programmers to specify their intention.

posed by Castagna [1997] would only require a small change in the metamodel. On the contrary, Clos *generic functions*, which are defined across classes, would require more in-depth changes. Interaction with class refinement should be carefully examined, especially method combination which would get a third dimension.

Finally, we have presented modules and class refinement at the programming level but they could also be considered at the design and modeling level. Obviously, our proposal closely interferes with Uml. The class and property metamodel can be considered as a more accurate version of a very small part of the Uml metamodel, whereas the module and class metamodel is a more accurate version of package merge. However, besides this simple shift from programming to modeling, the processes illustrated in this paper might be generalized to mechanisms such as *model composition* [Clarke 2002], which differ from *module import*, but which present some structural analogy. To paraphrase Szyperski [1992], “modules are not models, why [do] we need both”!

ACKNOWLEDGMENTS

This work has been partially supported by grants from Région Languedoc-Roussillon (034750).

REFERENCES

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

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

Ancona, D., Drossopoulou, S., and Zucca, E. 2001. Overloading and inheritance. In *8th Intl. Workshop on Foundations of Object-Oriented Languages (FOOL8).* `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL8.html`.

Ancona, D. and Zucca, E. 2001. True modules for Java-like languages. See Knudsen [2001], 354–380.

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

Apel, S., Leich, T., Rosenmüller, M., and Saake, G. 2005. Combining feature-oriented and aspect-oriented programming to support software evolution. In *Proc. 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05).* 3–16.

Ardourel, G. and Huchard, M. 2002. Access graphs, another view on static access control for a better understanding and use. *J. of Object Technology*, 95–116.

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

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., Shalit, A. L. M., and Withington, P. T. 1996. A monotonic superclass linearization for Dylan. See OOPSLA [1996], 69–82.

Bergel, A., Ducasse, S., and Nierstrasz, O. 2005. Classbox/J: controlling the scope of change in Java. In *Proc. OOPSLA'05*, R. Johnson and R. P. Gabriel, Eds. SIGPLAN Notices, 40(10). ACM Press, 177–189.

Bergel, A., Ducasse, S., and Nierstrasz, O. 2006. Analyzing module diversity. *J. Universal Computer Science 11,* 10, 1613–1644.

Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R. 2005. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures 31,* 3-4, 107–126.

Bergel, A., Ducasse, S., and Wuyts, R. 2003. Classboxes: A minimal module model supporting local rebinding. In *Proc. Joint Modular Languages Conf. (JMLC'03)*. LNCS 2789. Springer, 122–131.

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

Biberstein, M., Gil, J., and Porat, S. 2001. Sealing, encapsulation and mutability. See Knudsen [2001], 28–52.

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

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

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

Boucher, D. 2000. GOld: a link-time optimizer for Scheme. In *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, M. Felleisen, Ed. 1–12.

Bracha, G. 1992. The programming language Jigsaw: Mixins, modularity and multiple inheritance. Ph.D. thesis, University of Utah.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. See OOPSLA/ECOOP [1990], 303–311.

Bracha, G. and Lindstrom, G. 1992. Modularity meets inheritance. In *Proc. IEEE Int. Conf. Comp. Lang.* 282–290.

Bruce, K. B., Odersky, M., and Wadler, P. 1998. A statically safe alternative to virtual types. See Jul [1998], 523–549.

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

Cardelli, L. 1984. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. McQueen, and G. Plotkin, Eds. LNCS 173. Springer-Verlag, 51–67.

Cardelli, L. 2004. Type systems. In *The Computer Science and Engineering Handbook*, 2nd ed., A. B. Tucker, Ed. CRC Press, Chapter 97.

Carré, B. and Geib, J. 1990. The point of view notion for multiple inheritance. See OOPSLA/ECOOP [1990], 312–321.

Castagna, G. 1997. *Object-oriented programming: a unified foundation.* Birkhaüser.

Chailloux, J. 1993. *LeLisp version 15.25: reference manual.* Ilog.

Chiba, S. 1998. Javassist—a reflection-based programming wizard for Java. In *Proc. ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java.*

Chirila, C.-B., Crescenzo, P., and Lahire, P. 2004. A reverse inheritance relationship for improving reusability and evolution: The point of view of feature factorization. In *Proc. MASPEGHI'04 workshop at ECOOP'04*. 9–14.

Clark, T., Evans, A., and Kent, S. 2002. A metamodel for package extension with renaming. In *Proc. UML'02*, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds. LNCS 2460. Springer, 305–320.

Clarke, S. 2002. Extending standard UML with model composition semantics. *Sci. Comput. Program. 44,* 1, 71–100.

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. 2000. MultiJava: Modular open classes and symetric multiple dispatch for Java. See OOPSLA [2000], 130–145.

Cointe, P. 1987. Metaclasses are first class: the ObjVlisp model. In *Proc. OOPSLA'87*. SIGPLAN Notices, 22(12). ACM Press, 156–167.

Compagnoni, A. B. and Pierce, B. C. 1996. Higher order intersection types and multiple inheritance. *Mathematical Structures in Computer Science 6,* 5, 469–501.

Cook, W., Hill, W., and Canning, P. 1990. Inheritance is not subtyping. In *Proc. POPL'90*. ACM Press, 125–135.

DeMichiel, L. and Gabriel, R. 1987. The Common Lisp Object System: An overview. See Bezivin et al. [1987], 201–220.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and 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. (submitted to *ACM Comp. Surv.*; revised July 2005).

Ducournau, R. 2002b. "Real World" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, J.-M. Bruel and Z. Bellahsène, Eds. LNCS 2426. Springer, 3–12.

Ducournau, R. 2005. Perfect hashing as an almost perfect subtype test. Tech. Rep. 05-058, LIRMM, Université Montpellier 2. (submitted to acm toplas, revised Dec. 2006, Aug. 2007).

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Tech. Rep. 06-001, LIRMM, Université Montpellier 2. (submitted to acm toplas, Nov. 2006).

Ducournau, R. and Habib, M. 1987. On some algorithms for multiple inheritance. See Bezivin et al. [1987], 243–252.

Ducournau, R. and Habib, M. 1991. Masking and conflicts, or to inherit is not to own. See Lenzerini et al. [1991], Chapter 14, 223–244.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1992. Monotonic conflict resolution mechanisms for inheritance. In *Proc. OOPSLA'92.* ACM Press, 16–24.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1994. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA'94.* SIGPLAN Notices, 29(10). ACM Press, 164–175.

Ducournau, R., Habib, M., Huchard, M., Mugnier, M.-L., and Napoli, A. 1995. Le point sur l'héritage multiple. *Technique et Science Informatiques 14,* 3, 309–345.

Dugerdil, P. 1991. Inheritance mechanism in the Objlog language: multiple selective and multiple vertical with points of view. See Lenzerini et al. [1991], Chapter 15, 245–256.

Dujardin, E., Amiel, E., and Simon, E. 1998. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. Program. Lang. Syst. 20,* 1, 116–165.

Ernst, E. 1999. Propagating class and method combination. See Guerraoui [1999], 67–91.

Ernst, E. 2001. Family polymorphism. See Knudsen [2001], 303–326.

Ernst, E. 2002. Safe dynamic multiple inheritance. *Nord. J. Comput 9,* 1, 191–208.

Ernst, E. 2003. Higher-order hierarchies. In *Proc. ECOOP'2003*, L. Cardelli, Ed. LNCS 2743. Springer, 303–329.

Ernst, E., Ostermann, K., and Cook, W. R. 2006. A virtual class calculus. In *Proc. POPL'06.* ACM Press, 270–282.

Findler, R. B. and Flatt, M. 1999. Modular object-oriented programming with units and mixins. In *Proc. ICFP'98.* SIGPLAN Notices, 34(1). ACM Press, 94–104.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1998. Classes and mixins. In *Proc. POPL'98.* ACM Press, 171–183.

Forman, I. R. and Danforth, S. H. 1999. *Putting Metaclasses to Work.* Addison-Wesley.

Goldberg, A. and Robson, D. 1983. Smalltalk*: the language and its implementation.* Addison-Wesley.

Goldberg, D. S., Findler, R. B., and Flatt, M. 2004. Super and inner: together at last! See Vlissides and Schmidt [2004], 116–129.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The JAVA Language Specification*, Third ed. Addison-Wesley.

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 *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2: Nonmonotonic Reasoning*, D. Gabbay and C. Hogger, Eds. 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., and Ducournau, R. 1991. Towards a unique multiple inheritance linearization. In *Proc. EurOOP'91*, A. Mrazik, Ed. Bratislava.

Ichisugi, Y. and Tanaka, A. 2002. Difference-based modules: A class-independent module mechanism. In *Proc. ECOOP'2002*, B. Magnusson, Ed. LNCS 2374. Springer, 62–88.

Igarashi, A. and Pierce, B. C. 1999. Foundations for virtual types. *Lecture Notes in Computer Science 1628*, 161–??

ILOG 1996. *Power Classes reference manual, Version 1.4.* ILOG, Gentilly.

Jul, E., Ed. 1998. *Proceedings of the 12th European Conference on Object-Oriented Programming, ECOOP'98.* LNCS 1445. Springer.

Kiczales, G., des Rivières, J., and Bobrow, D. 1991. *The Art of the Meta-Object Protocol.* MIT Press.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An overview of AspectJ. See Knudsen [2001], 327–355.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proc. ECOOP'97*, M. Aksit and S. Matsuoka, Eds. LNCS 1241. Springer, 220–242.

Kiczales, G. and Rodriguez, L. 1990. Efficient method dispatch in PCL. In *Proc. ACM Conf. on Lisp and Functional Programming.* 99–105.

Knudsen, J. L., Ed. 2001. *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'2001.* LNCS 2072. Springer.

Knuth, D. E. 1973. *The art of computer programming, Sorting and Searching.* Vol. 3. Addison-Wesley.

Lalande, A. 1926. *Vocabulaire technique et critique de la philosophie.* Presses Universitaires de France.

Lenzerini, M., Nardi, D., and Simi, M., Eds. 1991. *Inheritance Hierarchies in Knowledge Representation and Programming Langages.* John Wiley & Sons.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. 2004. The Objective Caml system, release 3.09. documentation and user's manual. Tech. rep., INRIA. `http://caml.inria.fr/`.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6, 1811–1841.

Madsen, O. L. and Møller-Pedersen, B. 1989. Virtual classes. A powerful mechanism in object-oriented programming. In *Proc. OOPSLA'89.* ACM Press, 397–406.

Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. 1993. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley.

Meyer, B. 1992. *Eiffel: The Language.* Prentice-Hall.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. Prentice-Hall.

Meyer, B. 2001. Overloading vs. object technology. *J. Obj. Orient. Program. 14,* 5, 3–7.

Moon, D. 1986. Object-oriented programming with Flavors. See OOPSLA [1986], 1–8.

Mössenböck, H. 1993. *Object-Oriented Programming in Oberon-2.* Springer.

Nierstrasz, O., Ducasse, S., and Schärli, N. 2006. Flattening traits. *J. Object Technology 5,* 4, 129–146.

Nystrom, N., Chong, S., and Myers, A. C. 2004. Scalable extensibility via nested inheritance. See Vlissides and Schmidt [2004], 99–115.

Nystrom, N., Clarkson, M. R., and Myers, A. C. 2003. Polyglot: An extensible compiler framework for Java. In *Proc. 12th Int. Conf. on Compiler Construction (CC'03).* LNCS 2622. Springer, 138–152.

Nystrom, N., Qi, X., and Myers, A. C. 2006. $\mathcal{JE}$: Nested intersection for scalable software composition. In *Proc. OOPSLA'06*, P. L. Tarr and W. R. Cook, Eds. SIGPLAN Notices, 41(10). ACM Press, 21–35.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. 2004. An overview of the Scala programming language. Tech. Rep. IC/2004/64, EPFL, Lausanne, CH.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*. ACM Press, 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*. 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 2000. *Proceedings of the 15th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'00*. SIGPLAN Notices, 35(10). ACM Press.

OOPSLA/ECOOP 1990. *Proceedings of the 5th ACM Conference on Object-Oriented Programming, Languages and Applications, 3rd European Conference on Object-Oriented Programming, OOPSLA/ECOOP'90*. SIGPLAN Notices, 25(10). ACM Press.

Privat, J. 2006a. De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc. Ph.D. thesis, Université Montpellier II.

Privat, J. 2006b. PRM—the language. 0.2. Tech. Rep. 06-029, LIRMM, Montpellier.

Privat, J. and Ducournau, R. 2005. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*. 20–27.

Rayside, D. and Campbell, G. 2000a. An aristotelian introduction to classification. In *ECOOP'2000 Workshop on Objects and Classification, a natural convergence*, M. Huchard, R. Godin, and A. Napoli, Eds. RR LIRMM 2000-095.

Rayside, D. and Campbell, G. 2000b. An aristotelian understanding of object-oriented programming. See OOPSLA [2000], 337–353.

Rayside, D. and Kontogiannis, K. 2001. On the syllogistic structure of object-oriented programming. In *Proc. of ICSE'01*. 113–122.

Reynolds, J. 1996. Design of the programming language Forsythe. In *Algol-like languages*, P. O'Hearn and R. Tennent, Eds. Birkhauser.

Sakkinen, M. 2002. Exheritance—class generalisation revived. In *Proc. ECOOP'2002 Inheritance Workshop*, A. P. Black, E. Ernst, P. Grogono, and M. Sakkinen, Eds. University of Jyväskylä, 76–81.

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? `http://www.visviva.com/transframe/papers/covar.htm`.

Smaragdakis, Y. and Batory, D. 1998. Implementing layered designs with mixin layers. See Jul [1998], 550–570.

Smaragdakis, Y. and 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. See Lenzerini et al. [1991], Chapter 10, 153–171.

Steele, G. 1990. *Common Lisp, the Language*, Second ed. Digital Press.

Stefik, M. and 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 *Proc. ECOOP'92*, O. L. Madsen, Ed. LNCS 615. Springer, 19–32.

Thorup, K. and Torgersen, M. 1999. Unifying genericity: Combining the benefits of virtual types and parameterized classes. See Guerraoui [1999], 186–204.

Torgersen, M. 1994. The expression problem revisited—four new solutions using generics. In *Proc. ECOOP'94*, M. Tokoro and R. Pareschi, Eds. LNCS 821. Springer, 123–143.

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., and Hølzle, U. 1991. Organizing programs without classes. *Lisp and Symbolic Computation 4,* 3, 223–242.

van Rossum, G. and Drake, J. F. L. 2003. *The Python Language Reference Manual.* Network Theory Ltd.

VanHilst, M. and Notkin, D. 1996. Using role components to implement collaboration-based designs. See OOPSLA [1996], 359–369.

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

Vlissides, J. M. and Schmidt, D. C., Eds. 2004. *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'04*. SIGPLAN Notices, 39(10). ACM Press.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proc. OOPSLA'97.* SIGPLAN Notices, 32(10). ACM Press, 125–141.

Zenger, M. and Odersky, M. 2004. Independently extensible solutions to the expression problem. Tech. Rep. IC/2004/33, EPFL, Lausanne, CH.

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