Coloring, a Versatile Technique for Implementing Object-Oriented Languages

ROLAND DUCOURNAU
LIRMM – CNRS and Université Montpellier II, France

RR LIRMM 06001, 2006
revised September 27, 2010

This report is a preliminary version of an article to appear in *Software, Practice & Experience* in 2011.

Abstract

Late binding and subtyping create run-time overhead for object-oriented languages. Dynamic typing and multiple inheritance create even more overhead. Static typing and single inheritance lead to two major invariants—of reference and position—that make the implementation as efficient as possible. Coloring is a technique that preserves these invariants for dynamic typing or multiple inheritance at minimal spatial cost.

Coloring has been introduced and applied, more or less independently, to method invocation—under the name of *selector coloring*—to subtype tests under the name of *pack encoding* and to attribute access. This paper reviews a number of uses of coloring for optimizing object-oriented programming and generalizes them. It specifies several variations of coloring, such as bidirectional coloring and n-dimensional coloring. Coloring is NP-hard, so compilers that use it depend on heuristics. The paper describes some experimental results which indicate that bidirectional coloring gives the best results.

Keywords: object-oriented programming, graph coloring, conflict graph, downcast, graph coloring, late binding, message sending, multiple inheritance, object layout, pack encoding, selector coloring, single inheritance, subtype test, virtual function tables

1 Introduction

One of the implementation issues characteristic of object-oriented languages is *late binding*, which is also referred to as the *message sending* metaphor. The underlying principle is that the address of the actually called procedure is not statically determined at compile-time, but depends on the dynamic type of a distinguished parameter known as the *receiver*. In dynamically typed languages, the receiver's dynamic type is completely unknown and message sending can result in a run-time type error. However, static typing ensures only that the receiver's dynamic type is a subtype of its static type—the actual dynamic type is bounded, thus avoiding run-time errors, but remains statically unknown. An issue similar to message sending arises with attributes (aka *instance variables*, *slots*, *data members* according to the languages), since their position in the object layout may depend on the object's dynamic type. Furthermore, subtyping introduces another original feature, i.e. run-time subtype checks, which amounts to testing whether the value of x is an instance of some class C or, equivalently, whether the dynamic type of x is a subtype of C. This is the basis for so-called *downcast* operators.

Message sending, attribute access and subtype testing need specific implementations, data structures and algorithms. In statically typed languages, late binding is usually implemented with tables, called *virtual function tables* in C++ jargon. These tables reduce method calls to function calls, through a small fixed number—usually 2—of extra indirections. It follows that object-oriented programming yields some overhead, as compared to usual procedural languages. When static typing is combined with single inheritance—this is *single subtyping*—two major invariants hold: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position* of attributes and methods in the tables does not depend on the dynamic type of the object. These invariants allow direct access to the desired data and optimize the implementation. Otherwise, dynamic typing or multiple inheritance make it harder to retain these two invariants. Actually, the most commonly used language with multiple inheritance, i.e. C++, does not keep these invariants—hence its implementation is hampered by both significant overhead and ill-specified features [Ellis et Stroustrup, 1990 ; Lippman, 1996 ; Ducournau, 2002a].

Implementation is thus not a problem with single-subtyping languages. However, there are almost no such languages. The few examples, such as OBERON [Mössenböck, 1993], MODULA-3 [Harbinson, 1992], or ADA 95, result from the evolution of non-object-oriented languages and object orientation is not their main feature. In static typing, commonly used pure object-oriented languages, such as C++ or EIFFEL [Meyer, 1992 ; Meyer, 1997], offer the programmer plain multiple inheritance. More recent languages like JAVA and C# offer a limited form of multiple inheritance, whereby classes are in single inheritance and types, i.e. *interfaces*, are in *multiple subtyping*. Furthermore, the absence of multiple subtyping was viewed as a deficiency of the ADA 95 revision, and this feature was incorporated in the next version [Taft *et al.*, 2006]. This is a strong argument in favour of the importance of multiple inheritance. So there is a real need for efficient object implementation in the context of multiple inheritance and static typing. The multiple inheritance requirement is less urgent in the context of dynamic typing—an explanation is that the canonical static type system corresponding to a language like SMALLTALK [Goldberg et Robson, 1983] would be that of JAVA, i.e. multiple subtyping. Anyway, dynamic typing gives rise to implementation issues which are similar to that of multiple inheritance, even though the solutions are not identical, and the combination of both, as in CLOS [Steele, 1990], hardly worsens the situation.

Coloring can be defined as an optimization technique that retains these two invariants at minimal spatial cost. It has been introduced and applied, more or less independently, to method invocation—under the name of *selector coloring* [Dixon *et al.*, 1989]—to attribute access [Pugh et Weddell, 1990 ; Ducournau, 1991] and to subtype testing—under the name of *pack encoding* [Vitek *et al.*, 1997] and with the notion of *meet incompatible antichain partition* [Fall, 1995].

This article presents coloring as the versatile generalization of these three techniques, which amounts to coloring the graph of specific entities, respectively methods, attributes or classes. Coloring makes multiple inheritance as efficient as single inheritance implementations, so it should be envisaged for implementing languages but it is essential, first, to present theoretical and experimental evidence of its tractability. A theoretical analysis, following the first results by [Pugh et Weddell, 1990 ; Pugh et Weddell, 1993], is first presented. It turns out that optimal coloring is almost always NP-hard, so the paper proposes heuristics, together with a general scheme for using coloring with separate compilation, in spite of its non-incrementality. Finally, the tractability of coloring is assessed by its simulation on several large-scale benchmarks commonly used in the object-oriented language implementation community.

Overall, the contribution of this article is manyfold: (i) a survey of coloring as a generalization of the various proposals dedicated to each of the three aforementioned mechanisms; (ii) an analysis of its practical use in object-oriented implementation; (iii) a graph-theoretic formalization which stresses the role of a *conflict graph* and presents a synthesis of complexity results; (iv) a variety of heuristics; (v) a large-scale experiment on real-size benchmarks which show that bidirectional coloring is both efficient and tractable. Besides marginal usage in the YAFOOL language [Ducournau, 1991], we are not aware of any actual use of coloring in a real language. We are currently developping an experimental language, PRM [Privat et Ducournau, 2005 ; Privat, 2006], that uses coloring, but the main objective of this paper is to convince implementers that coloring is a worthwhile alternative to current object-oriented implementations.

Structure of the paper

Section 2 presents the usual object implementation in the static typing and single inheritance context. The problem with dynamic typing or multiple inheritance is stated. Section 3 presents the principles and briefly reviews the various contributions to coloring. Several variations are considered, according to the colored entities (methods, attributes, or classes) and the minimization criterion (color number or table size). A generalization to bidirectional and n-dimensional coloring is proposed and the practical use of all of these variants is examined. Finally, putting coloring into practice is discussed from the standpoint of compilers, linkers and loaders. In Section 4, two theoretical analyses are undertaken. The first one regards the structure of the class hierarchy, namely the so-called *conflict graph*, which is the main target of the coloring problem. The second one reports various contributions to the complexity of the problem, which is akin to *minimum graph coloring.* As expected, it is NP-hard in all but a few cases. Section 5 describes tractable heuristics and presents the results of their experiments. It follows that coloring is tractable, from the standpoint of computation time, at compile or link time, and also from that of memory occupation, at run-time. Section 6 presents some works closely related to coloring, together with some alternatives. A conclusion and perspectives are presented at the end of the paper.

2 From single subtyping to multiple inheritance

Single subtyping, i.e. single inheritance and static typing, allows for a straightforward constant-time implementation of the basic object-oriented mechanisms. We first present this implementation and explain why it does not easily generalize to multiple inheritance or dynamic typing. The reader is referred to [Ducournau, 2002a] for a review of the implementation techniques for object-oriented languages.

2.1 Single subtyping

2.1.1 Method call and object layout

In separate compilation of statically typed languages, late binding is generally implemented with tables called *virtual function tables* in C++ jargon. Method calls are then reduced to

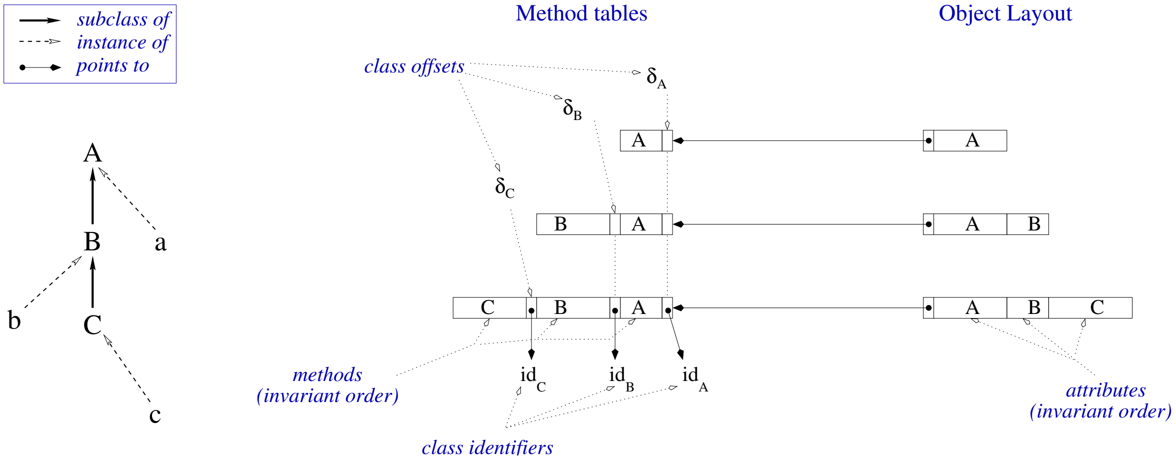


Figure 1: Single subtyping implementation. A class hierarchy with 3 classes A, B and C with their respective instances, a, b and c (left). Method tables (center)—including Cohen's display—and object layout (right). This diagram respects the following conventions. In method tables, A (resp. B, C) represents the set of addresses of methods *introduced* by A (resp. B, C). Method tables are drawn from right to left to reduce edge crossing. In object layouts, the same convention applies to attributes but the tables are drawn from left to right.

function calls through a small fixed number (usually 2) of extra indirections. On the other hand, an object—e.g. the receiver—is laid out as an attribute table, with a header pointing at the class table and possibly some added information, e.g. for garbage collection. With single inheritance, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of that of its single superclass (Figure 1). Figure 2 presents a diagram of the object layout and method table in this setting, together with the corresponding code sequences in an intuitive pseudo-code. This pseudo-code is borrowed from [Driesen, 2001]. Contrary to previous works, instruction-level parallelism will not be considered, as it has no effect here. So the role of these code sequences is mostly paraphrasing the corresponding diagrams.

The resulting implementation respects two essential *invariants*: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position* of attributes and methods in the tables does not depend on the dynamic type of the object. Therefore all accesses to objects are straightforward, but this simplicity requires both static typing and single inheritance. From a spatial standpoint, the object layout is clearly optimal, since there is one field per attribute, with a single extra pointer at the method table, which shares all data common to all direct instances of the considered class. The method tables are not very small, but they are also, in some sense, optimal. If one assumes that the method introduction is uniformly distributed over all classes, the total size of method tables is linear in the size of the class specialization relationship, which is assumed to be reflexive and transitive. In the worst case, this is however quadratic in the number of classes.

2.1.2 Subtype tests

This feature resorts to dynamic typing, hence it does not depend on static types. So there is no need, here, to distinguish between class and type, or between specialization and subtyping.

```
// attribute access
load [object + #attColor], attVal

// method invocation
load [object + #tableOffset], table
load [table + #methColor], methAddr
call methAddr

// subtype test
load [object + #tableOffset], table
load [table + #colorC], idC
comp idC, #targetId
bne #fail
// succeed
```

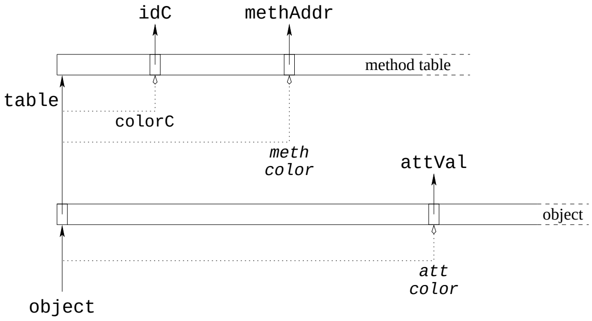


Figure 2: Single subtyping implementation. Code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. Pointers and pointed values are in Roman type with solid lines, and offsets are italicized with dotted lines.

In the following, the class specialization relationship is denoted $\preceq$. It is transitive, reflexive and antisymmetric.

Subtype testing is less straightforward than method invocation and access to attributes. In the single inheritance context, several techniques have been proposed and are commonly used. We present only one of the simplest ones that provides a basis for further generalizations. The technique is known as *Cohen's display* and was first described by [Cohen, 1991] as an adaptation of the 'display' originally proposed by [Dijkstra, 1960].

Cohen's display consists of assigning two integers to each class, a unique ID and a non-unique color. Let the ID of class C be id_C and the color of C be $\chi(C)$. Each subclass of a class C has id_C stored in its method table at $\chi(C)$. Thus, an object is a direct or indirect instance of a class C if and only if offset $\chi(C)$ in its method table (tab) contains id_C. This is equivalent to testing that the class, say D, that has instantiated the considered object is a subtype of C:

$$D \preceq C \Leftrightarrow tab_D[\chi(C)] = id_C \tag{1}$$

Originally, Cohen's method required a set of tables separate from the method tables, and $\chi(C)$ was the depth of C in the class hierarchy. However, in a statically typed language, these tables can be merged with the method tables. Class offsets are ruled by the same position invariant as methods and attributes. This can be thought of as giving each class C a method for checking whether an object is an instance of C, i.e. such that its instances can check that they are. The test fails when this pseudo-method is not found.

Inlining the Cohen display in the method table requires that a class ID must be distinct from a method address. As addresses are even numbers (due to word alignment), coding class IDs with odd numbers avoids any confusion between the two types. In theory, $tab_D[\chi(C)]$ is sound only if $\chi(C)$ is in tab_D. If one assumes that class offsets are positive, a comparison of $\chi(C)$ with the length of tab_D seems required, together with memory access to the length itself—this would hinder efficiency. Fortunately, there is a simple way to avoid this test, i.e. by ensuring that, in some specific memory area, the value id_C always occurs at offset $\chi(C)$ of the table tab_D of some class D. This can be ensured if method tables contain only method addresses and class IDs—which cannot be confused—and if the specific memory area contains only method tables and is padded with some even number, to a length corresponding to the maximum tab size. In this way, Cohen's test preserves linear-space tables. If method tables contain more data than addresses and IDs, i.e. something that might take any half-word or word value—even though we did not identify what—a more complex coding or an indirection might be required. Anyway, if class IDs are gathered within specific tables, distinct from method tables and allocated in the same contiguous way, this extra indirection will have the same cost as access to length—apart from cache misses—but the test itself will be saved.

A frequently proposed way to save on this bound check is to use fixed-size tab_D. [Click et Rose, 2002] attribute the technique to [Pfister et Templ, 1991]. However, statistics on a set of benchmarks commonly used in the object-oriented language implementation community (Table 4) show that, on these benchmarks, the maximum superclass number may be 5-fold greater than its average. Hence, fixed size tables would entail a large space overhead and a physical limitation, with a tradeoff between both. Actually, a fixed size always entails this tradeoff. However, limitation for limitation, the overhead is quite a bit larger when the fixed size concerns many, possibly thousands, small tables rather than a single or a few large areas.

2.2 Multiple inheritance

2.2.1 Semantics of inheritance

The meaning of multiple inheritance is a subject of endless debates which are far beyond the scope of this paper. Therefore, we first present a simple abstraction which will be sufficient for the present paper and make things clear—interested readers should refer to a more formal in-depth presentation in [Ducournau et Privat, 2008]. Inheritance involves two basic entities, *classes* and *properties*, with the latter standing for *methods* and *attributes*. Classes have a *name* and properties have a *signature*, i.e. this accounts for parameter types and static overloading. Classes are organized in a *specialization* hierarchy, i.e. a partial order denoted $(X, \prec)$. $B \prec A$ means that A is a superclass of B. The relationships between classes and properties can be characterized by 4 verbs—a class *knows*, *defines*, *introduces* and *inherits* properties:

- a class A *knows* a property p—this means that instances of A can answer the message p, i.e. they have a value (attribute) or a function (method) of the same signature as p;
- a class A *defines* a property p if the definition of A includes a definition of the signature of p;
- a subclass B *inherits* all properties of its superclass A—i.e. if A knows p, then B knows p;
- a class A *introduces* p if A defines a property p with a new signature, i.e. a signature which does not correspond to any inherited properties; of course, if A introduces p, then A knows p.

The term 'introduction' is used with this meaning throughout the paper. We must now distinguish two cases, according to whether the typing is static or dynamic.

In static typing, we assume that all signatures are *fully qualified*, i.e. they include the name of the *introducing* class. Moreover, each property is introduced by a single class—[Pugh et Weddell, 1990] call these properties *class-based*. When considering the property `foo` introduced in class `A`, we must think of `A@foo` either in `A` or in all subclasses of `A`. So the pair formed by the signature and the introduction class is unambiguous—e.g. in the code fragment

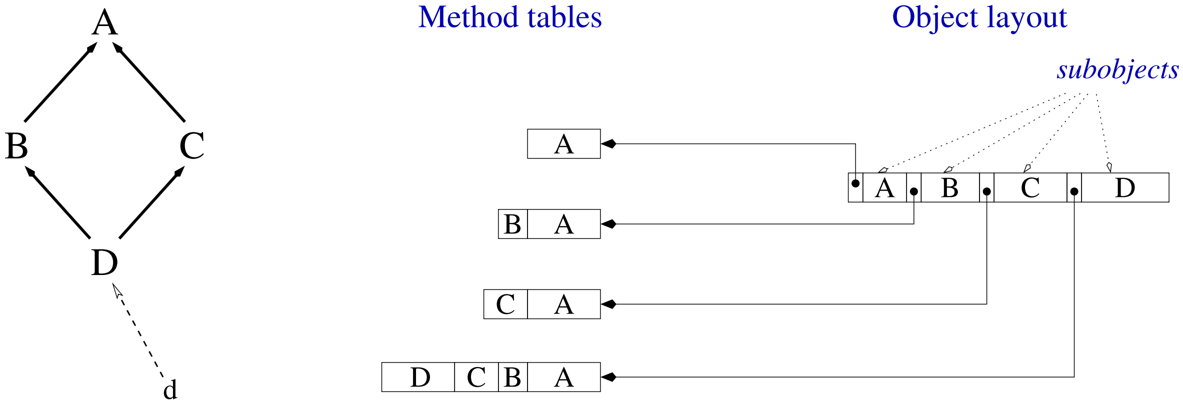


Figure 3: Subobject-based implementation. The object layout of a single instance d of the class D is depicted, with its 4 subobjects and the corresponding method tables.

{`x:T; x.A@foo;`}, where `T` is some subclass of `A`. In a real language, the qualification would be inferred from the static type and `x.foo` would be unambiguous in the same context. In the diamond example of Figure 3, if both `B` and `C` introduce a property with the same unqualified signature `bar`, they actually respectively introduce the signatures `B@bar` and `C@bar`, and their common subclass `D` inherits both as distinct properties. In contrast with the preceding example, the fragment {`x:T; x.bar;`}, where `T` is some subclass of `D`, is ambiguous and the explicit qualification would be mandatory in a real language.

In dynamic typing, we cannot rely on the static type, hence there is no way to distinguish properties with the same signature when they are introduced by different classes. Therefore, a property has an unqualified signature and can be introduced by several classes—e.g. class `D` inherits a single property of signature `bar`.

Without loss of generality, implementation of method invocation and attribute access is only concerned by allocating a position to properties, i.e. to unqualified signatures in dynamic typing, or to pairs of signatures and introduction classes in static typing. The precise property definition does not matter—this is the method address which will fill in the method table entry at the considered position.

2.2.2 Subobject-based implementation

With multiple inheritance, both invariants of reference and position cannot hold together, at least if compilation—i.e. computation of positions—is to be kept separate. For instance, in the diamond hierarchy of Figure 3, if the implementations of B and C simply extend that of A, as in single inheritance, the same offsets will be occupied by different properties in B and C, thus prohibiting a sound implementation of D. Therefore, the 'standard' implementation of multiple inheritance (SMI) in a static typing and separate compilation setting—i.e. that of C++—is based on *subobjects*. The object layout is composed of several subobjects, one for each superclass of the object's class. Each subobject contains attributes *introduced* by the corresponding class, together with a pointer to a method table which contains the methods *known* by the class. Both invariants are dropped, as both reference and position depend on the current static type. This is the C++ implementation, when the keyword `virtual` annotates each superclass [Ellis et Stroustrup, 1990 ; Lippman, 1996 ; Ducournau, 2002a]. It is time-constant and compatible with dynamic loading, but method tables are no longer space-linear.

```
// pointer adjustment
load [object + #tableOffset], table
load [table + #castOffset], delta1
add object, delta1, tobject

// private attribute access
load [object + #attOffset], attVal

// inherited attribute access
// lines 1-3
load [tobject + #attOffset], attVal

// method invocation
load [object + #tableOffset], table
load [table + #methOffset+1], delta2
load [table + #methOffset], methAddr
add object, delta2, self
call methAddr
```

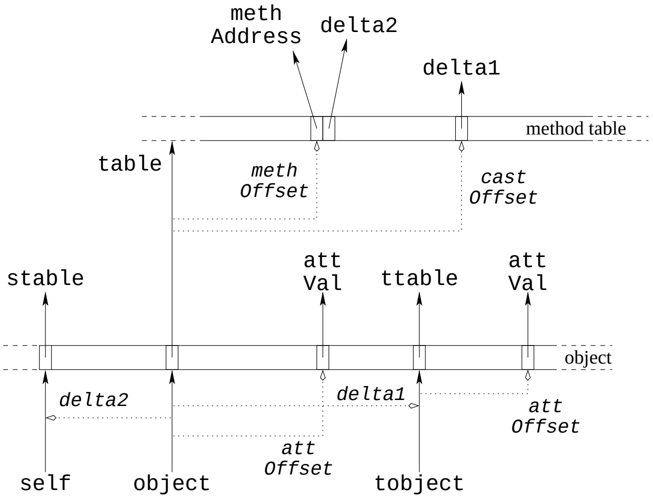


Figure 4: Standard multiple inheritance implementation—code sequences for all the basic mechanisms, but subtype testing, and the corresponding diagrams of object layout and method table.

The number of method tables is exactly the size of the specialization relationship. In the worst case, the total table size of a class is itself quadratic in the number of superclasses—so the total size for all classes is cubic in the number of classes. Furthermore, all polymorphic object manipulations—i.e. assignments and parameter passing, when the source type is a subtype of the target type—which are quite numerous, require *pointer adjustments* between source and target types, as they correspond to different subobjects. These pointer adjustments are purely mechanical and do not bring any semantics. They are also safe—i.e. the target type is always a supertype of the source type—so they are implemented more efficiently than subtyping tests. They can be done with explicit pointers, called VBPTRs, in the object layout or with offsets in the method tables. There are, however, a lot of variants, according to whether compiler-generated fields are allocated in the object layout, like VBPTRs, or in the method tables, as in Figure 4. [Sweeney et Burke, 2003] analyse this variety, from the ARM implementation, where all fields are allocated in the object layout, to the ALL implementation, where all fields are allocated in the method tables. We only consider ALL implementation here, which is closest to most actual implementations, but our conclusions hold for the whole implementation family. Furthermore, contrary to single inheritance, there is no known way to derive a subtype test from the technique used for method invocation. It is no longer possible to consider that testing if an object is an instance of some class C is a kind of method introduced by C because this pseudo-method would not have any known position other than in static subtypes of C.

Figure 4 displays the code sequence for basic SMI mechanisms—apart from subtype testing—together with the corresponding diagram. At method invocation, `self`-adjustment[1] is required since the static type of the receiver in the callee is not known in the caller—it is done, here, by an offset included in the 2-fold entries of the method table. The diagram

[1] `Self` denotes the current receiver in Smalltalk and corresponds to `this` in C++ and Java and to `current` in Eiffel. Here we follow the Smalltalk usage, which is closer to the original metaphor. `Self` can be considered as a reserved formal parameter of the method, and its static type—even in dynamic typing!—is the class within which the method is defined.

displays three different pointers at the same object: `object` is the original reference to the object in the caller, with a certain static type C, `self` is the reference to the subobject of some type D, unrelated with C, which has defined the callee method, and `tobject` is the reference to the subobject corresponding to a supertype of C which introduces the desired attribute.

More details can be found in the aforementioned references. The overall complexity of the implementation is clear but instruction-level parallelism partly reduces the actual overhead for method invocation [Driesen, 2001]. Finally, the main drawback of this implementation family is that its overhead remains even when multiple inheritance is not used. Therefore, language designers have provided alternative specification and implementation, known as *non-virtual inheritance*, when omitting the `virtual` keyword. Non-virtual inheritance gives exactly the same implementation as single inheritance in the case of single inheritance hierarchies, but it is ill-specified for general multiple inheritance hierarchies, hence preventing sound reusability—for instance, in the diamond example of Figure 3, the attributes introduced by A would be duplicated in the object layout of D.

2.3 Dynamic typing

Replacing multiple inheritance by dynamic typing leads to similar problems regarding method invocation and access to attributes—however, the subtype test does not depend on static or dynamic typing. The same property `foo`—i.e. properties with the same name `foo`—can be introduced in unrelated classes, therefore at different places, in such a way that an access to `foo` on an entity `x`, e.g. `x.foo()`, cannot directly determine the position of `foo` in the table of the current value of `x`. Thus, the pioneer of object-oriented languages, SMALLTALK [Goldberg et Robson, 1983], finds an efficient solution in a strict *encapsulation* of attributes—the language reserves to `self` all accesses to attributes. Although SMALLTALK is dynamically typed, the position invariant holds for attributes because an object's attributes can only be accessed by its methods. Regarding method invocation, it mainly involves a combination of non-constant time techniques, e.g. hashing and caching [Conroy et Pelegri-Llopart, 1983 ; Deutsch et Schiffman, 1984 ; Hölzle *et al.*, 1991].

3 Principle, history and applicability of coloring

The general idea underlying coloring is to keep the two invariants of single inheritance—i.e. reference and position. There is a trivial but inefficient solution, i.e. give each attribute, method and class a unique number. This will result in a lot of empty slots, so a lot of wasted space. Coloring is a way of minimizing the wasted space. This cannot be done separately for each class, but requires complete knowledge of the class hierarchy. Figure 5 depicts a possible implementation of the example of Figure 3. The section first presents a short history of coloring and the combinatorics of all variants. Putting coloring to work is then examined by successively analyzing (i) the role of typing, either static or dynamic, (ii) the implications on compilers, linkers and run-time systems, then (iii) the n-directional variants.

3.1 Short history

Coloring, as a general implementation technique, has been independently discovered in the specific case of each basic mechanism. It follows from Section 2.1 that all three mechanisms

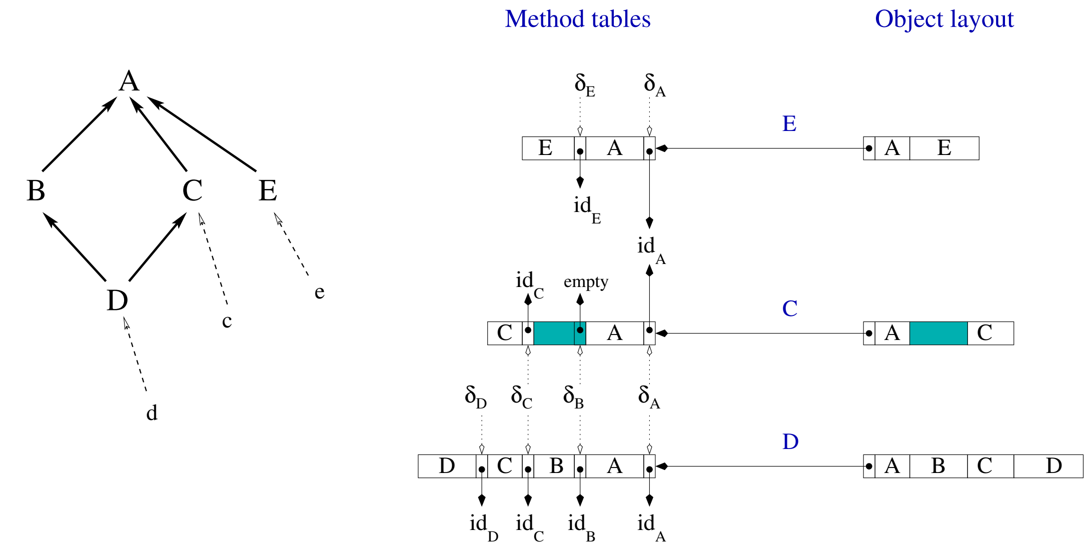


Figure 5: Unidirectional coloring applied to classes, methods and attributes. A and B classes are presumed to have the same implementation as in Figure 2 and there are holes in the C tables since some space must be reserved for B in the tables of D. In contrast, class E can occupy the same positions as B because both classes have no common subclass.

are basically reducible to each other—indeed, a subtype test can be reduced to a kind of special method but the method table layout can also be considered as an object layout at the meta level and attributes can be accessed through accessor methods. Therefore, this is not surprising that all three mechanisms can be generalized in a single one, i.e. coloring[2]. However, object layout and method table have memory requirements of their own.

3.1.1 Method coloring and minimization of color number

Coloring was first mentioned as an implementation technique for object-oriented languages by [Dixon *et al.*, 1989]. The technique, called *selector*[3] *coloring*, is applied to method invocation. This coloring problem is an instance of the well known *graph coloring* problem since determining an offset—called a *color*—for each selector amounts to coloring the undirected graph, that here we call *selector coexistence graph*, whose vertices are selectors, whereby there is an edge between two selectors when both are *known* by some class. The optimization criterion is the color number. From the complexity standpoint, as an instance of the classic *minimum graph coloring problem*, this is an NP-hard problem [Garey et Johnson, 1979 ; Toft, 1995 ; Jensen et Toft, 1995]. From the spatial standpoint, this is equivalent to minimizing the size of a large matrix, whose rows are classes and columns are selector colors.

A first experiment with real-size hierarchy—the Smalltalk hierarchy—was reported by [André et Royer, 1992]. As they explicitly computed the coexistence graph, which was quite large, before using heuristics to color it (see Section 5.2), the conclusions were rather negative and coloring was long considered to be an intractable technique[4]. Later, selector coloring was

[2] This common abstraction is however not possible with all implementations, as we have noted that subtype testing was irreducible to method invocation in subobject-based implementations.

[3] Selector is the term used in Smalltalk for what we call *signature* (Section 2.2.1).

[4] This critique is not directed at André and Royer—the author is actually deeply indebted to them for

integrated in a general framework for method dispatch by [Holst et Szafron, 1997].

3.1.2 Attribute coloring and minimization of table size

At about the same time as selector coloring, [Pugh et Weddell, 1990] proposed a similar technique, but applied to attributes. The *attribute coexistence graph* is defined in an analogous way, by substituting attribute to selector in the *selector coexistence graph* definition. The point is no longer to minimize the number of colors, i.e. the matrix size, but rather the total size of all of the tables associated with the different classes. Equivalently, this amounts to minimizing the number of *holes*, i.e. empty entries of tables. A coloring without hole is said to be *perfect*.

From an historical standpoint, [Pugh et Weddell, 1990] cite [Dixon *et al.*, 1989] but the authors do not explicitly recognize their proposition as an analogue of selector coloring. They actually consider graph coloring, but only in order to decide on the existence of bidirectional perfect coloring or to prove the complexity of bidirectional coloring (Section 4.5). However, [Cheung et Grogono, 1992] make a similar but rather underdeveloped proposition, which is applied in the Dee system—both [Dixon *et al.*, 1989] and [Pugh et Weddell, 1990] are cited and recognized as similar. Independently, attribute coloring was also used, on a rather small scale, in the Yafool language [Ducournau, 1991].

Of course, minimizing the total size of a set of variable-size tables was obviously also applicable to methods—this was done later, in a French-language review paper on message sending in dynamically typed languages [Ducournau, 1997]. Moreover, this paper presented experimental evidence of the existence of tractable heuristics.

3.1.3 Bidirectional and n-dimensional coloring

Besides applying coloring to attributes and object layout, Pugh and Weddell provide a very interesting generalization, *bi-directionality*, which reduces the size. Bidirectional coloring involves both positive and negative colors, contrary to unidirectional coloring, which involves only positive colors. Generalization to n-directional or n-dimensional coloring was further proposed by [Pugh et Weddell, 1993] and rediscovered by [Zibin et Gil, 2003]—[Gil *et al.*, 2008] is a synthesis. Besides coloring, the bidirectionality of method tables or object layout has been widely reused [Myers, 1995 ; Krall et Grafl, 1997 ; Gil et Sweeney, 1999 ; Gagnon et Hendren, 2001]. Bidirectional coloring must not be confused with *two-way coloring* [Huang et Chen, 1992], which is a generalization of selector coloring that merges both rows and columns of the dispatch matrix.

Whereas [Pugh et Weddell, 1990 ; Pugh et Weddell, 1993] are key works on coloring—all considerations were already there apart from the application to method invocation and subtype testing—strangely enough the importance of these papers has been overlooked. In the 90s, it seems that the first paper was only cited by [Myers, 1995] and [Gil et Sweeney, 1999]. It also appears in the reference list of Driesen's PhD thesis [2001], but we could not find any comments on this reference in the text.

initially inciting him to focus more closely on this issue—but at computer scientists, collectively. A negative experiment is not a counter-example of the tested approach, but only of the applied protocol.

		optimization criterion				colored entities			typing		systems			
paper		#colors	unidir.	bidir.	*k*-dir.	method	attribute	class	static	dynamic	compiler	linker	loader	libraries
Dixon et al.	[1989]	×				×			×	×	×			
Pugh and Weddell	[1990]		×	×			×			×	×	×		
Ducournau	[1991]		×				×			×				
André and Royer	[1992]	×				×				×	×		×	
Cheung and Grogono	[1992]		×				×					×		
Pugh and Weddell	[1993]			×	×	×	×			×	×	×		
Fall	[1995]	×						×						
Ducournau	[1997]		×			×				×	×			
Vitek et al.	[1997]	×						×			×			
Ducournau	[2002b]		×	×		×	×	×	×	×	×	×		
Palacz and Vitek	[2003]	×						×	×				×	
Zibin and Gil	[2003]				×		×		×	×	×			
Privat and Ducournau	[2005]		×	×		×	×	×	×			×		
Privat and Morandat	[2008]		×	×		×	×	×	×			×		×

Table 1: Coloring bibliography

3.1.4 Class coloring and pack encoding

Class coloring was proposed by [Vitek *et al.*, 1997] under the name of *pack encoding.* This is the direct extension of Cohen's test to multiple inheritance. *Class coexistence graph* is now defined as the undirected graph whose vertices are classes such that there is an edge between two classes iff they have a common subclass, particularly if one of them is a subclass of the other. [Fall, 1995] anticipated this idea with the notion of *meet incompatible antichain*, but he did not look into it closely when he recognized that the problem was NP-hard. Vitek et al. use fixed-size tables, so the technique amounts to minimizing the color number. However, they do not make any reference to graph coloring—the word 'color' is actually not used—and the paper does not cite [Dixon *et al.*, 1989] nor [Pugh et Weddell, 1990].

Finally, though coloring is inherently non-incremental, [Palacz et Vitek, 2003] propose to use it for subtype tests in JAVA, when the target type is an interface.

3.2 Combinatorics of coloring

All of these various works can be abstracted in a general technique, here called *coloring*, which has different variants (Table 1):

- it can apply to classes, methods or attributes; actually, as methods and classes should likely be colored in the same tables, it should apply jointly to classes and methods; moreover, class coloring can also serve for accesses to attributes through *accessor simulation* (Section 3.3.2);
- the minimization criterion may be the color number or the total table size or, equivalently, the hole number; in the case of attributes, the size should be weighted by the number of instances;
- typing may be static or dynamic;

```
// attribute access
load [object + #tableOffset], table
load [table + #colorC+1], attC
add object, attC, attgr
load [attgr + #attOffset], attVal
```

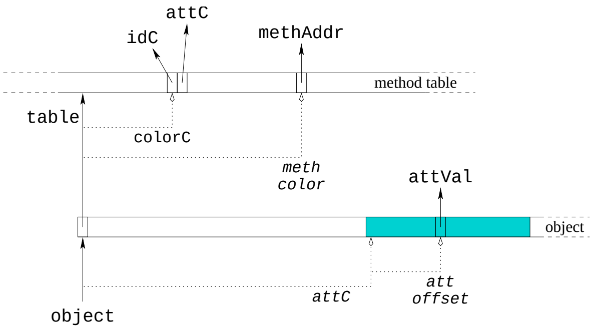


Figure 6: Accessor simulation in static typing—code sequence for access to attributes and diagram of object layout and method table. The group of attributes introduced by class C is in grey.

- coloring may be unidirectional, bidirectional, or even n-dimensional.
- and, finally, coloring is preferably used in a global setting rather than with dynamic loading, but incremental versions have also been proposed for Smalltalk [André et Royer, 1992] or Java [Palacz et Vitek, 2003].

All combinations are possible, and their respective efficiency and complexity must be studied (Section 4.5). The optimization problem is exactly minimum graph coloring—when fixed-size tables are used—or akin to it. So, it is likely NP-hard in most cases. Therefore heuristics are needed, along with some experiments to check their tractability and evaluate the size of the resulting data structures (Section 5).

3.3 Coloring and typing

3.3.1 Application to static typing

In a static typing setting, uni- and bi-directional coloring are direct extensions of single inheritance implementation, and all three mechanisms use the exact same code and implementation as that used for single subtyping, hence providing the same time efficiency. However, colors must be globally computed, whereas single inheritance allows an incremental computation. Moreover, coloring results in tables that may include some holes, i.e. empty slots. This is not too expensive for method and class coloring, since class tables are usually a small percentage of the total size of a program. However, attribute coloring can be expensive—if a class with only a few attributes but many instances has a few holes, the wasted space might be large. One solution is to profile a program, determine which classes have the most instances, and make sure that they are not classes with holes in their attributes. Another solution is accessors, which eliminates the need for profiling at a slight run-time cost.

3.3.2 Accessors and accessor simulation

An accessor is a method that either reads or writes an attribute. Suppose that all accesses to an attribute are through an accessor. Then the attribute layout of a class does not have to be the same as the attribute layout of its superclass. A class will redefine the accessors for an attribute if the attribute has a different offset in the class than it does in the superclass. True accessors require a method call for each access, which can be inefficient. However, a class can simulate accessors by replacing the method address in the method table with the attribute's offset. This approach is called *field dispatching* by [Zibin et Gil, 2003]. Another improvement is to group attributes introduced by the same class together in the method table. Then one can substitute, for their different offsets, the single relative position of the attribute group, stored in the method table at the class color—i.e. at an invariant position (Figure 6).

Accessor simulation is a generic approach for access to attributes, which works with any method invocation technique—only grouping can be conditioned by static typing, since attributes must be partitioned by the classes which introduce them.

3.3.3 Application to dynamic typing

Coloring is also applicable to dynamic typing, with some key differences regarding attribute and method coloring. The lack of static typing forces the compiler to generate, for each call site, the code allowing to check at run-time that the invoked method is actually known by the current receiver—i.e. the receiver might know another method with the same color. A simple way to do this is to add an extra parameter to each method—i.e. the selector itself—and to check it in the method prologue[5]. Furthermore, bound checking is mandatory and empty entries are filled with the address of a function which signals an exception—`doesNotUnderstand` in SMALLTALK. Regarding attributes, an analogue would require an image of the object layout in the method table, whereby each field contains the name of the corresponding attribute. This would not be efficient and the SMALLTALK solution is surely the best one—namely *encapsulation* which reserves attributes to `self`. Access to attributes of other entities is mediated by true *accessors*, which may be generated by the compiler or defined by the programmer—i.e. only the latter in SMALLTALK. Coupled with single inheritance, encapsulation makes attribute access to `self` as efficient as with single subtyping. Coupled with multiple inheritance, encapsulation allows for both attribute coloring and accessor simulation, that would apply as in static typing, apart from grouping which is no longer possible since an attribute can be introduced by more than one class. However, with accessor simulation, the efficiency of attribute access to `self` would be partially lost. Despite this relative inadequacy, attribute coloring was mostly proposed and studied in dynamic typing settings—either FLAVORS [Pugh et Weddell, 1990] or YAFOOL [Ducournau, 1991]. Finally, as noted above, subtype testing does not depend on static types, so class coloring can be applied in dynamic typing without any change.

3.4 From compilation to class loading

The single subtyping implementation is truly modular and incremental—it can be applied in separate compilation and dynamic loading. This is still the case for subobject-based implementations, but not for coloring.

3.4.1 Link-time coloring

The main defect of coloring is that it requires complete knowledge of all classes in the hierarchy. This complete knowledge is usually achieved by global compilation. However, giving up

[5] As this extra parameter is used only in the prologue, it can be passed in a register, i.e. more efficiently than other parameters.

the modularity provided by separate compilation may be considered too high a price for program optimization. An alternative was already noted by [Pugh et Weddell, 1990]. Coloring does not require knowledge of the code itself, but only of the 'model' (aka 'schema') of the classes, all of which is already needed by separate compilation of object-oriented programs, namely the model of the superclasses of the currently compiled class, together with other classes which are used for typing the code of the current class. This class model is included in specific header files (in C++) or automatically extracted from source or compiled files (in Java). Therefore, the compiler can separately generate the compiled code without knowing the value of the colors of the considered entities, representing them with specific symbols. At link time, the linker will collect the models of all classes and color all the entities, before substituting values to the different symbols, as a linker commonly does. The linker can also be enriched by optimizations, e.g. type analysis and dead code elimination [Privat et Ducournau, 2005], but this is far beyond the scope of this paper.

The *double compilation* scheme proposed by [Myers, 1995] provides a final optimization of accessor simulation, when attributes are encapsulated. It involves assuming that each class has one *primary superclass* plus some *secondary superclasses*. Each class is compiled in two versions. The efficient version considers that the current class is always specialized as a *primary superclass* and all accesses to attributes of `self` are compiled in the usual position-invariant way. On the contrary, the second version considers that the current class is sometimes specialized as a *secondary superclass*—hence, the position invariant does not hold and accesses to attributes must use accessor simulation, i.e. extra memory access. The appropriate version is selected at link-time. When classes are only used in single inheritance, the efficient version is always chosen, so accessors entail no overhead, at least for all encapsulated accesses.

[Privat et Morandat, 2008] examine the possibility of using coloring with shared libraries and global linking. In their proposition, only the library code is shared. Method tables and colors are proper to each program and the actual implementation requires extra memory access for all colors and class IDs.

3.4.2 Load-time coloring

Finally, a definitive defect will remain—i.e. coloring is not incremental, hence apparently not suitable for dynamic loading. However, as aforementioned, some authors have attempted to use coloring at load-time. Actually, any global technique can be used in a dynamic loading setting at the expense of dynamic data structures, hence extra indirections, and of possibly complete recomputations. When the considered technique is inherently non-incremental, using it in a dynamic setting might make it lose all its desired qualities that are ensured in a global setting. So, the point with using coloring at load-time is to examine the overhead entailed at load-time—recomputation cost—and at run-time—extra indirection cost. From the load-time standpoint, specific incremental heuristics must be designed since coloring is NP-hard and near-optimal[6] global heuristics are roughly cubic (see Section 5). These incremental heuristics will likely favour load time to the detriment of run-time space. At run-time, the generated code also requires memory access to colors—i.e. colors cannot be replaced by values at link-time. Nevertheless, the main point would be that the method table of exist-

[6] This is an informal usage of the term. We only mean that these heuristics give apparently good results, but we do not mean that they always give good results. In the general case, minimum graph coloring problem is non-approximable within a constant factor [Bellare *et al.*, 1998].

```
// preamble
load [object + #tableOffset], table
load [table + #ctableOffset], ctable
load [classColor], colorC
add ctable, colorC, entry

// attribute access
load [entry + #1], attC
add object, attC, attgr
load [attgr + #attOffset], attVal

// method invocation
load [entry + #2], methC
add table, methC, methgr
load [methgr + #methOffset], methAddr
call methAddr

// subtype test
load [entry], idC
comp idC, #targetId
bne #fail
// succeed
```

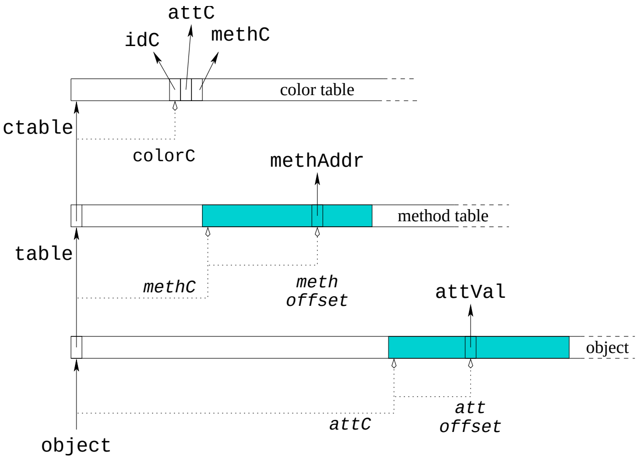


Figure 7: Load-time coloring in static typing—all three mechanisms begin with the same preamble

ing instances would have to be updated—this is easy to do, but at the expense of an extra indirection. Moreover, a dynamic method table might well make bound checks mandatory in Cohen's test. It would also be unreasonable for attribute coloring—the layout of existing objects might change—unless an expensive feature like the CLOS `change-class` generic function is accepted [Steele, 1990]. Furthermore, since incremental heuristics will likely be less space-optimal than global ones, the number of holes in the object-layout will be significant. Of course, load-time coloring would still be compatible with accessor simulation. Overall, coloring can be envisaged at load-time in a static typing setting, as follows (Figure 7):

- class coloring is the basis of all mechanisms;
- attributes are implemented with accessor simulation, i.e. the offset `attC` of the attribute group is stored at the class color `colorC`;
- methods are implemented in an analogous way, i.e. all methods introduced by a class are grouped together in the method table and the offset `methC` of the method group is stored at the class color `colorC`.

The class coloring table is made of 3-fold entries: the class identifier `idC`, for subtyping test, the attribute offset `attC` and the method offset `methC`. This allows one to make object-layout and method tables invariant and reducing load-time computation to class coloring. However, the resulting run-time efficiency is far from that of link-time coloring, especially for attributes, since they require four extra `loads`, in three unrelated memory areas, hence with possible cache misses.

JAVA-like languages present a more realistic application field. In these languages, classes are in single inheritance but types, i.e. classes and interfaces, are in multiple subtyping. Therefore, attributes are not concerned and incremental coloring could be applied to subtype testing when the target type is an interface—this is the proposition of [Palacz et Vitek, 2003]—but also to method invocation when the receiver is typed by an interface.

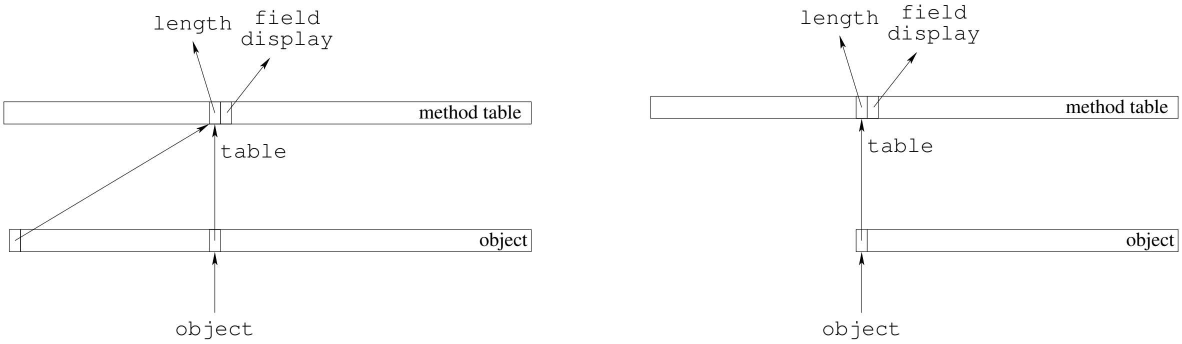


Figure 8: Bidirectional object layout for garbage collection. When an object layout uses negative colors, an extra pointer to the method table is added. The length of the object-layout is included in the method table, together with the required information to decide which fields are pointers that must be followed during garbage collection.

Dynamic typing would make things even more difficult since attributes and methods cannot be grouped according to their introduction class. So dynamic typing requires method coloring at load-time but this would be quite expensive.

3.5 Application of n-dimensional coloring

Let us first specify the terminology. Let n be an arbitrary positive integer, then n-directional coloring consists of coloring with n tables of positive offsets—each entity is assigned both a table and a color in this table. So the technique described up to now is *unidirectional*, i.e. 1-directional, coloring. When $n = 2$, both tables can be interpreted as a single 1-dimensional array, with negative and positive offsets—this is *bidirectional coloring*. One generalizes with k-dimensional coloring, which involves k bidirectional arrays, i.e. $2k$-directional coloring[7]. Conversely, n-directional coloring involves $\lceil n/2 \rceil$ dimensions. In all generality, increasing n must reduce the hole number—if the coloring of a class in n tables produces some holes, these holes could be avoided with an extra table. For instance, in Figure 5, coloring C in negative offsets would yield perfect coloring.

Unidirectional Unidirectional coloring directly applies to all three cases of attributes, classes and methods—preferrably, of classes and methods jointly. However, the resulting space may be considered as less than optimal and far from desired, so an improved technique may be preferred.

Bidirectional The application of bidirectional coloring is also straightforward in the case of methods and classes. Bidirectional method tables—i.e. negative offsets—are not a problem. They are actually rather common, at least in a research setting [Myers, 1995 ; Krall et Grafl, 1997 ; Gil et Sweeney, 1999 ; Gagnon et Hendren, 2001]. In contrast, bidirectional object layout might make memory management difficult, as garbage collection requires some information regarding memory allocation at the beginning—or at least at a fixed offset from

[7] This terminology is slightly different from that of [Zibin et Gil, 2003]—i.e. our dimension is their *layer*. Conversely, their two *dimensions* follow from the fact that an entry is determined by two coordinates, the table index (among 1..k) and the offset in the bidirectional table—hence, the actual color is the pair index-offset.

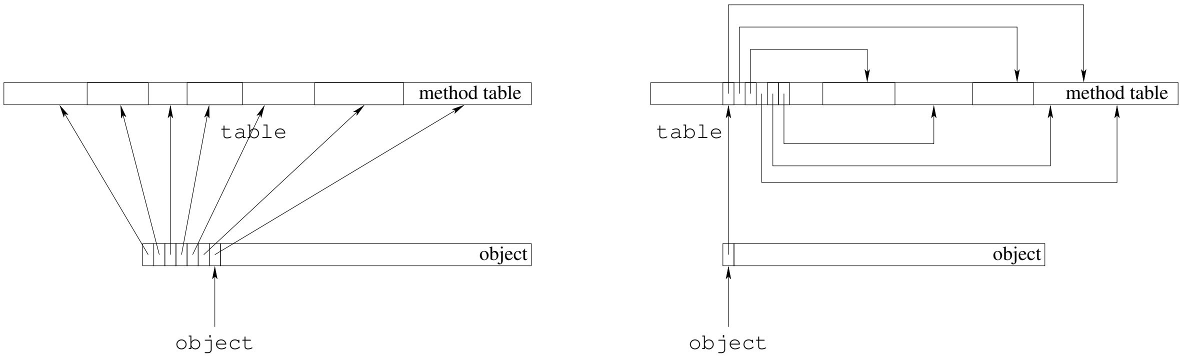


Figure 9: n-dimensional method tables, with an n-pointer header in the object layout (left) or with indirections in the method table (right).

the beginning—of each memory block. Therefore, an extra pointer to method table might be required, at least when negative offsets are used [Desnos, 2004] (Figure 8). The memory gain w.r.t. unidirectional coloring will be reduced accordingly.

[Gagnon et Hendren, 2001] propose a bidirectional layout designed especially for optimizing garbage collection, by assigning negative offsets to references and positive offsets to immediate values. In a copying garbage collection, this incurs no overhead as all fields must be scanned before being copied and an extra pointer is not required. This is, however, not applicable to bidirectional coloring, since negative offsets are no longer dedicated to references.

Multi-dimensional The aim of n-dimensional coloring, when $n > 1$, is to reduce the hole number, which should decrease when n increases—though heuristics cannot always ensure this obvious mathematical fact. The gain is, however, questionable. In the case of methods, it yields multiple method tables which can be, however, contiguous. A possible object layout involves an n-word header, with each word pointing at the corresponding bidirectional table (Figure 9, left). Of course, in a static typing setting, the header can be left-truncated when there is no method in the upper dimensions. With such a layout, method invocation incurs no overhead at all, however this is to the detriment of object layout. An alternative involves a single pointer at the method table in the first dimension—while the other tables are pointed by the first one (Figure 9, right). This incurs no overhead in object layout, but rather a constant overhead in method invocation for all methods which are not in the first dimension. In a global compilation setting, the overhead will likely concern only a small subset of method calls. However, in a separate compilation and global linking setting, it might concern all method calls. Regarding class coloring, the same arguments apply—however, the left truncation would require an extra bound check. Now, it is not mandatory to improve on bidirectional coloring for methods and classes since it is already quite good—the expected gain would be low and counterbalanced by the multi-dimensional drawbacks. Therefore, it is likely useless to further consider n-dimensional coloring for classes and methods.

The real point is attribute coloring which presents the same alternative as method coloring (Figure 10). Each dimension represents, in the object layout, a kind of bidirectional subobject. These subobjects might be explicitly linked with $n-1$ pointers in the first subobject, but it would contradict the intention of saving on space—this would generate the same drawbacks as VBPTRs (Section 2.2.2). Therefore, the alternative proposed by [Zibin et Gil, 2003] would

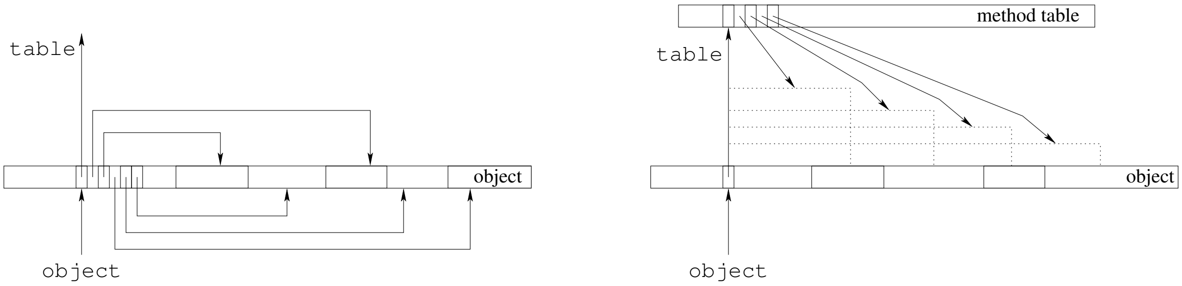


Figure 10: n-dimensional object layout, with an n-pointer header in the object layout (left) or with indirections through offsets in the method table (right).

be preferred—i.e. the relative positions of $n-1$ subobjects are stored in the method table. The argument is now the same as with methods. Most attributes are presumed to be in the first dimension, which incurs no overhead in a global compilation setting. Other dimensions require an indirection in the method table, as with accessor simulation or subobject-based implementation, when the attribute is not introduced by the static type of the receiver. Hence, perfect n-dimensional attribute coloring may be understood as an optimization of *accessor simulation* (Section 3.3.2), but it optimizes only with global compilation, not with global linking which cannot avoid indirection for the first dimension. Double compilation optimization applies, but in a slightly different way, namely *multiple compilation*, since several versions can be considered. For instance, if a class is always specialized as a primary superclass, attributes introduced by the class can be laid out in the first dimension. This is, however, not always true for inherited attributes. An alternative is to consider the case where all attributes of the class can be laid out in the first dimension. Overall, n-dimensional attribute coloring might be a slight improvement of accessor simulation, but at the expense of a somewhat significant complication and it should be reserved to global compilation.

Conclusion Anticipating the results of our experiments (Section 5.2), it follows from this analysis that language implementers should use bidirectional coloring for classes and methods. Regarding attributes, the choice is mainly between accessor simulation and bidirectional coloring—profiling applications would be a major improvement for coloring.

4 Formal definitions and Theoretical results

This section presents formal definitions and surveys theoretical results. Though this section can be mostly skipped, the definitions introduced in Section 4.2 are essential for understanding the heuristics proposed in Section 5.

4.1 Notations and definitions

Definition 4.1 ((Hierarchy))**.** *A* hierarchy *(aka* inheritance graph*) is an directed acyclic graph* $(X, \prec_d)$*, where* X *is the set of classes and* $\prec_d$ *is the direct specialization relationship—i.e.* $B \prec_d A$ *iff* A *is a direct superclass of* B*.* $\prec$ *denotes the transitive closure of* $\prec_d$ *and* $\preceq$ *is the reflexive closure of* $\prec$*.*

We assume that $\prec_d$ includes no transitivity arcs—i.e. $\prec_d$ is the transitive reduction of $\prec$. $(X, \preceq)$ is a *partial order* (aka *poset*), since $\preceq$ is reflexive, transitive and antisymmetric. *Maximal* (resp. *minimal*) elements of a subset of X refer to $\preceq$: x is maximal (resp. minimal) in $X' \subset X$ iff there is no $x' \in X'$ such that $x \prec x'$ (resp. $x' \prec x$). $\max_\prec(X')$ and $\min_\prec(X')$ denote, respectively, the sets of maximal and minimal elements of X'. Unqualified uses of max and min denote the single *maximum* or *minimum* element of an integer set. The reader is referred to [Cormen *et al.*, 2001] for a definition of all other notions of graph and order theories that might be used hereafter but are not essential. Finally, given two sets E and F, $E \uplus F$ asserts that the sets are disjoint and denotes their union. Moreover, given a function $f : E \rightarrow F$ and a subset $E' \subset E$, $f(E')$ denotes the set $\{f(x) \mid x \in E'\}$.

P is the set of the considered properties, i.e. either attributes or methods. Moreover, h is a subset of $X \times P$ which formalizes the relationship between classes and properties—$h(x, y)$ holds iff the class x *knows* the property y. The function $p : X \rightarrow 2^P$ associates with each class x the set $p(x) \overset{\text{def}}{=} \{y \in P \mid h(x, y)\}$ of properties known by x. The following property characterizes inheritance:

$$x' \prec x \Rightarrow p(x) \subseteq p(x') \tag{2}$$

Conversely, the function $c : P \rightarrow 2^X$ associates with each property y the set of classes which *introduce* the property:

$$c(y) \overset{\text{def}}{=} \max_{\prec}\{x \in X \mid h(x, y)\} \tag{3}$$

As aforementioned, a property is *class-based* iff $c(y)$ is a singleton and, in static typing, all properties are class-based.

In the following, we consider a hierarchy $(X, \prec_d)$, a set of properties P, together with h, p and c. Y stands for X, P or $X \uplus P$—the latter in the case of class and method joint coloring.

Definition 4.2 ((Coloring)). *A* unidirectional *(resp.* bidirectional, k-directional*) coloring is a function $\chi : Y \rightarrow Z$, with $Z = \mathbf{N}$ (resp. $\mathbb{Z}$, $\mathbf{N} \times [1, k]$), such that $\forall x, y \in Y$,*

$$\chi(x) = \chi(y) \Rightarrow \begin{cases} x, y \textit{ have no common subclass} & x, y \in X \\ x, y \textit{ do not belong to the same class} & x, y \in P \\ y \textit{ does not belong to a subclass of } x & x \in X, y \in P \end{cases} \tag{4}$$

Of course, $\mathbf{N}$ (resp. $\mathbb{Z}$) and $\mathbf{N} \times [1, 1]$ (resp. $\mathbf{N} \times [1, 2]$) are isomorphic—so, unidirectional (resp. bidirectional) is an abbreviation for 1-directional (resp. 2-directional). For all $i = 1..k$, $Y_i = \{x \in Y \mid \chi(x) = (n, i)\}$ and $\chi_i : Y_i \rightarrow N$ is the function such that $\chi(x) = (\chi_i(x), i)$. We do not consider n-dimensional coloring—$\mathbf{N}$ should just be replaced by $\mathbb{Z}$ in k-directional coloring.

Definition 4.3 ((Color table)). *Given a coloring $\chi : Y \rightarrow Z$, the function $\chi^* : X \rightarrow 2^Z$ is defined as follows:*

$$\chi^*(x) \overset{\text{def}}{=} \chi(Y_x) \quad \textit{with} \quad Y_x = \begin{cases} \{y \mid x \preceq y\} & Y = X \\ p(x) & Y = P \\ \textit{union of both} & Y = X \uplus P \end{cases} \tag{5}$$

$\chi^*(x)$ is called the color table *of x. When $Z = \mathbf{N} \times [1, k]$, for all $i \in [1, k]$, the projection $\chi_i^* : X \rightarrow \mathbf{N}$ is defined by: $\chi_i^*(x) = \{n \mid (n, i) \in \chi^*(x)\}$.*

Alternatively, one defines $\chi^+, \chi^\uparrow : X \to 2^Z$, such that $\chi^+(x)$ (resp. $\chi^\uparrow(x)$) denotes the set of colors *introduced* (resp. *inherited*) by x:

$$\chi^+(x) \overset{\text{def}}{=} \begin{cases} \{\chi(x)\} & Y = X \\ \{\chi(z) \mid x \in c(z)\} & Y = P \\ \text{union of both} & Y = X \uplus P \end{cases} \tag{6}$$

$$\chi^\uparrow(x) \overset{\text{def}}{=} \bigcup_{x \prec_d y} \chi^*(y) = \biguplus_{x \prec y} \chi^+(y) \tag{7}$$

$$\chi^*(x) = \chi^\uparrow(x) \uplus \chi^+(x) \tag{8}$$

Proposition 4.4. *A function $\chi : Y \to Z$ is a coloring iff χ is injective on Y_x, for all $x \in X$.*

This alternative characterization of coloring amounts to note that a color table cannot contain two different things at the same place. The restriction to all $x \in \min_\prec(X)$ keeps the condition sufficient. The proof is trivial.

Definition 4.5 ((Optimal coloring))**.** *A coloring $\chi : Y \to Z$ is* optimal *if it minimizes* $\begin{cases} \max_{y\in Y} \chi(y) & Z = \mathbf{N},\ \textit{min. color number} \\ \sum_{x\in X} \max(\chi^*(x)) & Z = \mathbf{N},\ \textit{minimum size} \\ \sum_{x\in X} (\max(\chi^*(x)) - \min(\chi^*(x) \cup \{0\})) & Z = \mathbb{Z},\ \textit{minimum size} \\ \sum_{i=1..k} \sum_{x\in X} \max(\chi_i^*(x)) & Z = \mathbf{N} \times [1,k],\ \textit{min. size} \end{cases}$

In the bidirectional case, the minimum must be negative to keep the isomorphism with 2-directional coloring. Furthermore, the argument used in Section 2.1.2 for saving the bound check in Cohen's test can be applied to bidirectional coloring, under the condition that color 0 is inside each table. In practice, this is likely for all rooted hierarchies, as 0 is the natural root color. This is also true for property coloring—color 0 is occupied by the pointer at method table, which is considered as an implied attribute introduced in the hierarchy root, and is used for information about memory allocation, in the method table.

Alternatively, when the minimization criterion is the size, *holes* can be defined:

Definition 4.6 ((Hole))**.** *Given a coloring χ, a* hole *is an empty entry in the color table of some class x, i.e. an element of the set*

$$H_\chi(x) \overset{\text{def}}{=} \begin{cases} [0, \max(\chi^*(x))] \backslash \chi^*(x) & Z = \mathbf{N} \\ [\min(\chi^*(x) \cup \{0\}), \max(\chi^*(x))] \backslash \chi^*(x) & Z = \mathbb{Z} \\ \biguplus_{i=1..k}(([0, \max(\chi_i^*(x))] \backslash \chi_i^*(x)) \times \{i\}) & Z = \mathbf{N} \times [1,k] \end{cases} \tag{9}$$

$h_\chi(x)$ is the number of holes in the color table of x, i.e. the cardinality of $H_\chi(x)$.

Minimizing the total size is equivalent to minimizing the total hole number, i.e. $\sum_{x\in X} h_\chi(x)$.

Definition 4.7 ((Optimal weighted coloring))**.** *Given a* weight *function $w : X \to \mathbf{N}$, a coloring $\chi : Y \to Z$ is* optimal *if it minimizes*

$$\begin{cases} \sum_{x\in X} w(x).\max(\chi^*(x)) & Z = \mathbf{N} \\ \sum_{x\in X} w(x).(\max(\chi^*(x) - \min(\chi^*(x) \cup \{0\}) & Z = \mathbb{Z} \\ \sum_{i=1..k} \sum_{x\in X} w(x).\max(\chi_i^*(x)) & Z = \mathbf{N} \times [1,k] \end{cases} \tag{10}$$

or, equivalently, the weighted number of holes, $\sum_{x\in X} w(x).h_\chi(x)$.

4.2 Structure of the hierarchy

4.2.1 Regular coloring

Property coloring is largely reducible to class coloring. Indeed, two attributes or methods ‘conflict’, i.e. they cannot have the same color, if they have been introduced in ‘conflicting’ classes, which cannot have the same color, e.g. when they are $\preceq$-related. The condition is however sufficient only when properties are class-based. Intuitively, a class-based property can be colored in the same condition as the single class which introduces it. The analogy is exact when each class introduces exactly one property—class coloring is a special case of property coloring. In order to formalize this intuition, we introduce the notion of *regular coloring*:

Definition 4.8 ((Regular coloring))**.** *A* regular coloring *is either a* class coloring *($Y = X$) or a* property coloring *($Y = P$ or $Y = X \uplus P$) where all properties are* class-based*, i.e.* $|c(y)| = 1, \forall y \in P$*.*

In static typing, all colorings are regular. Anyway, in all cases, an analysis of the class hierarchy is mandatory.

4.2.2 Core and crown

The single inheritance approach applies to regular colorings when and where classes are in single inheritance. Given a regular coloring of a class and all its superclasses, this coloring can be simply extended to subclasses which are only specialized in single inheritance. This leads to distinguishing parts of the hierarchy which are in single or multiple inheritance.

Definition 4.9 ((Core))**.** *The* core *of the hierarchy is the subset X_{co} of classes in multiple inheritance, i.e. which have more than one direct superclasses or such a subclass:*

$$X_{co} \overset{\text{def}}{=} \{x \in X \mid \exists y, z_1, z_2 \in X : y \preceq x, y \prec_d z_1, y \prec_d z_2, z_1 \neq z_2\} \tag{11}$$

Definition 4.10 ((Crown))**.** *The* crown *of the inheritance graph is the subset X_{cr} of classes in single inheritance, that are only specialized in single inheritance:* $X_{cr} \overset{\text{def}}{=} X \backslash X_{co}$*.*

In the poset terminology, the core is a *filter* of $(X, \preceq)$ and the crown is an *ideal*[8]. Moreover, $(X_{cr}, \prec_d)$ is a *forest*, i.e. a set of disjoint directed trees. An interesting subset of the core is the border:

Definition 4.11 ((Border))**.** *The* border *of the inheritance graph is the subset X_{bo} of $\prec$-minimal classes in the core, that have several direct classes but whose subclasses are all in single inheritance:*

$$X_{bo} \overset{\text{def}}{=} \{x \in X_{co} \mid y \prec x \Rightarrow y \in X_{cr}\} = \min_{\prec}(X_{co}).$$

Figure 11 presents an example of the IDL hierarchy, with its core, crown and border. Intuitively, a regular coloring of X amounts to coloring the core X_{co}, then to extending this coloring to the crown X_{cr} in the same algorithmic way as in single inheritance, except that there may be some holes in the superclass table, which must be filled in the subclass. This intuition is formalized as follows:

[8] $F \subset X$ is a filter of $(X, \preceq)$ iff $x \in F$ & $x \preceq y$ implies $y \in F$. Conversely, I is an ideal iff $x \in I$ & $y \preceq x$ implies $y \in I$.

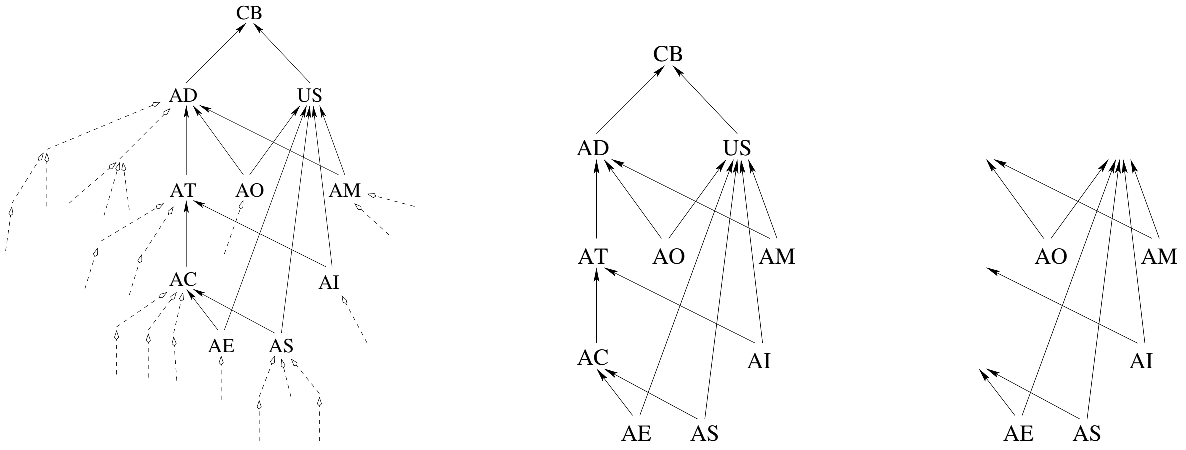


Figure 11: Core, crown and border. The pictures depict a part of the IDL hierarchy (left), its core (middle) and border (right)—dashed lines represent edges originating from the crown.

Definition 4.12 ((Partial coloring)). *A* partial coloring *of a filter X' of X, is a function $\chi : Y' \to Z$, where Y' is X', the subset $P' \subset P$ of properties introduced by classes in X', or $X' \uplus P'$, such that (4) holds for all $x, y \in Y'$.*

Definition 4.13 ((Partial coloring extension)). *A partial coloring χ' of X'* extends *a partial coloring χ'' of X'' if $X'' \subset X'$ and $\chi'/Y'' = \chi''$.*

As coloring is essentially non-incremental, any partial coloring cannot be extended into a complete coloring. However,

Proposition 4.14. *Any partial regular coloring of the core can be extended in a coloring of the whole hierarchy.*

The proof follows from the fact that a class in the crown cannot induce a conflict between two superclasses. □

However, this proposition does not mean that any optimal regular coloring of the core can be extended to an optimal coloring of the whole hierarchy. Indeed, the crown plays a role in the minimization criterion. Conversely, the restriction to the core of an optimal regular coloring may be nonoptimal.

4.2.3 Conflict graph

Coloring amounts to graph coloring, for some *coexistence graphs* of classes, attributes or methods.

Definition 4.15 ((Coexistence graph)). *The* coexistence graph *of Y is the undirected graph $(Y, \diamond)$, where $x \diamond y$ iff*

$$x \neq y \text{ and } \begin{cases} x, y \text{ have common subclasses} & x, y \in X \\ x, y \text{ belong to the same class} & x, y \in P \\ y \text{ belongs to a subclass of } x & x \in X, y \in P \end{cases}$$

This condition is a logical complement of equation (4) in Definition 4.2. Hence, an equivalent characteristics of colorings is that, for all k, $\chi^{-1}(k)$ is an *independent set* of the coexistence graph—i.e. the restriction $\diamond/\chi^{-1}(k)$ is empty.

The $\diamond$ relationships account for the informal use of the term 'conflict' in Section 4.2.1. However, even the *class coexistence graph* (when $Y = X$) is too large to be useful—for instance it includes $\prec$—and a better formulation of regular colorings requires defining the following *conflict graph*, which greatly simplifies the problem:

Definition 4.16 ((Conflict graph))**.** *The* conflict graph *of a hierarchy is the undirected graph* $(X_{co}, \leftrightarrow)$*, where* $x \leftrightarrow y$ *iff* $x \not\preceq y$*,* $y \not\preceq x$ *and* $\exists z \in X_{co}, z \prec x, z \prec y$*.*

Two classes conflict iff they are $\prec$-incomparable and have a common subclass. In other words, the conflict graph is the *incomparability graph* of $(X, \preceq)$, restricted to classes with common subclasses. The class coexistence relation $\diamond$ is the disjoint union of both $\prec$ and $\leftrightarrow$ relationships. All edges in the conflict graph are between two classes in $X_{co} \backslash X_{bo}$—however, Proposition 4.14 does not hold if the border is not considered. From now on, the term 'conflict' is related to the conflict graph. Two classes conflict when they are $\leftrightarrow$-related.

The conflict graph can also serve for regular property coloring. Indeed, two class-based properties conflict iff their introducing classes conflict. This is a simplification of the conflict graph defined on the set P of all properties—attributes or methods:

Definition 4.17 ((Property conflict graph [Pugh et Weddell, 1990]))**.** *The* property conflict graph *is the undirected graph* $(P, \leftrightarrow)$*, where* $x \leftrightarrow y$ *iff* x *and* y *coexist in some class and there are classes with* x*, without* y*, and classes with* y*, without* x*:*

$$y \leftrightarrow y' \overset{\text{def}}{\iff} c(y) \not\subset c(y') \ \& \ c(y') \not\subset c(y) \ \& \ (\exists x \in X, h(x, y) \ \& \ h(x, y')) \tag{12}$$

Actually, this set-wise condition can be implemented in a more efficient way, by simply counting the corresponding entities and comparing the resulting numbers [Pugh et Weddell, 1990].

Proposition 4.18. *Let* $y, y' \in P$ *be two class-based properties, with* $c(y) = \{x\}$ *and* $c(y') = \{x'\}$*. Then* $y \leftrightarrow y'$ *iff* $x \leftrightarrow x'$*.*

The proof is straightforward. □

4.3 Perfect regular coloring

Single subtyping implementation is a special perfect case of coloring, where there is no hole at all. [Pugh et Weddell, 1990] generalizes this remark by defining *perfect coloring*:

Definition 4.19 ((Perfect coloring))**.** *A* perfect coloring χ *is a coloring without any holes, i.e.* $h_\chi(x) = 0, \forall x \in X$*.*

A perfect coloring is optimal. Proposition 4.14 extends to perfect colorings:

Proposition 4.20. *Any perfect partial regular coloring of the core can be extended in a perfect coloring of the whole hierarchy.*

Proof. The natural extension of core coloring involves applying the single inheritance algorithm to the crown, i.e. a top-down class ordering with the selection of the next free color (Section 5.1.1). It cannot introduce new holes since there are no conflicts between classes in the crown. □

Pugh and Weddell have shown that the existence of perfect coloring is closely related to the property conflict graph colorability. The following proposition generalizes their results to regular coloring.

Proposition 4.21. [Pugh et Weddell, 1990, Th. 1 and 2, for $k \leq 2$]*There is a* perfect k-directional regular coloring *if the conflict graph can be colored in k colors.*

This characterization of perfect regular colorings involves two-level coloring. Each color of the conflict graph determines a *direction*—in the following, we shall use 'direction' instead of 'color'. Then each direction is separately colored—in the second meaning.

Proof. A k-coloring of the conflict graph partitions X_{co} in k *independent sets* $X_i, i = 1..k$, i.e. for all i, the restriction of $\leftrightarrow$ to X_i is empty. Then each $(X_i, \prec)$ is a forest, which can be colored as in single inheritance. □

The condition is sufficient but not necessary. Necessary conditions may be added in the following way:

Proposition 4.22. *There is a* perfect k-directional class coloring *iff the conflict graph can be colored in k colors.*

Proof. Let us prove that the directions give a coloring of the conflict graph, i.e. that each direction is an independent set. Let x and y be two classes colored in the same direction i and suppose that $x \leftrightarrow y$. Then $\chi_i(x) \neq \chi_i(y)$ and, for instance, $\chi_i(x) < \chi_i(y)$. Then $\chi_i(x)$ is obviously a hole in the y table color, since x and y have a common subclass. □

Corollary 4.23. *There is a uni- (resp. bi-) directional class coloring iff the conflict graph is edgeless (resp. bipartite).*

Finally, note that it is hopeless to expect a local characterization of perfect colorings, for instance based on the restriction of the conflict graph on the superclasses of each class. Any graph is the conflict graph of some hierarchy—a common subclass just has to be added to each connected pair. In the resulting hierarchy, the conflict graph restriction to each class is 2-colorable.

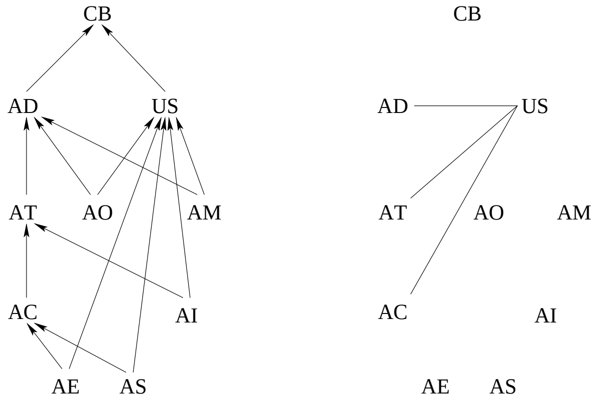
Proposition 4.22 holds also for class and method joint coloring when methods are class-based. In the case of property regular coloring, the necessary condition holds only if the conflict graph is restricted to classes introducing properties:

Proposition 4.24. *There is a* perfect k-directional method *(resp.* attribute) regular coloring *iff the conflict graph restricted to classes introducing methods (resp. attributes) can be colored in k colors.*

Proof. The proof of Proposition 4.22 works, but x and y must be constrained to introduce some properties—obviously, a class which introduces no properties has no effect at all on regular property coloring. □

Example

Figure 12 presents an example of the IDL hierarchy. The core and conflict graph are depicted. It appears that the latter is bipartite—hence there is a perfect bidirectional coloring. Class coloring is displayed in the unidirectional case (with 3 holes) and in the perfect bidirectional case, together with the specific memory area which gathers all color tables and avoids bound checking.



	-1	0	1	2	3	4
CB		CB				
AD		CB	AD			
US	US	CB				
AT		CB	AD	AT		
AC		CB	AD	AT	AC	
AO	US	CB	AD	AO		
AM	US	CB	AD	AM		
AE	US	CB	AD	AT	AC	AE
AS	US	CB	AD	AT	AC	AS
AI	US	CB	AD	AT	AI	

	0	1	2	3	4	5
CB	CB					
AD	CB	AD				
US	CB		US			
AT	CB	AD		AT		
AC	CB	AD		AT	AC	
AO	CB	AD	US	AO		
AM	CB	AD	US	AM		
AE	CB	AD	US	AT	AC	AE
AS	CB	AD	US	AT	AC	AS
AI	CB	AD	US	AT	AI	

	US
US ⟶	CB
CB ⟶	CB
AD ⟶	CB
	AD
AT ⟶	CB
	AD
	AT
AC ⟶	CB
	AD
	AT
	AC
	US
AO ⟶	CB
	AD
	AO
	US
AM ⟶	CB
	AD
	AM
	US
AE ⟶	CB
	AD
	AT
	AC
	AE
	US
AI ⟶	CB
	AD
	AT
	AI
	US
AS ⟶	CB
	AD
	AT
	AC
	AS

Figure 12: Class coloring of the core of the IDL hierarchy. From top to bottom and left to right: core and conflict graph, bidirectional and unidirectional color tables, and the specific memory area gathering all bidirectional color tables.

4.4 Irregular coloring

4.4.1 Perfect irregular coloring

In dynamic typing, when properties are no longer class-based, one would like to formulate conditions on perfect coloring in terms of property conflict graphs—e.g. there is a *perfect k-directional property coloring* iff the property conflict graph can be colored in k colors. However, only the sufficient condition holds. [Pugh et Weddell, 1990] prove that 2-colorability is a sufficient condition for 2-directional perfect coloring. In their 1993 paper, they generalize to all k but they also prove that there are no necessary conditions:

Proposition 4.25. [Pugh et Weddell, 1993, Theorem 1] *There is a perfect k-directional property coloring if the property conflict graph is k-colorable.*

Proposition 4.26. [Pugh et Weddell, 1993, Theorem 2] *For any positive integer n, there is a hierarchy with perfect unidirectional coloring, but where the conflict graph is not n-colorable.*

4.4.2 Attribute coloring and abstract classes

In the case of attributes, the optimization criterion must be the total size but it should also take the number of instances of each class into account. Hence an optimal weighted coloring is needed. Of course, determining w is an issue that requires profiling or some other approach.

An apparently simpler way involves only *abstract classes*, i.e. classes without direct instances—some languages allow the programmer to declare that a class is abstract. This amounts to considering that w takes its values in $\{0, 1\}$. It would be advantageous to remove abstract classes from the conflict graph—in the same way as we did for classes introducing no properties. However, this removal would make us lose the advantage of static typing, i.e. *class-based* properties. When taking abstract classes into account, attribute coloring is no longer regular. Indeed, after the removal of some abstract class, attributes introduced by this class would be implicitly introduced by all of its direct subclasses.

Hence, Proposition 4.26 shows that it is no longer possible to correlate a perfect coloring with abstract classes to the k-colorability of the conflict graph. Assume that all classes in the core are abstract. Irrespective of the conflict graph, given an optimal coloring of the core, it is always possible to define non-abstract classes in the crown in such a way that perfect coloring of the crown is obtained by simply filling the holes in the layout of the abstract classes.

Conversely, when a property is introduced by several classes, it is always possible to add classes to factorize the property. This can be done by minimizing the number of added classes, following the approach of Galois lattices (aka *formal concept analysis*) [Wille, 1992 ; Godin *et al.*, 1998]. This is roughly what is done when one goes from the SMALLTALK type system to the JAVA type system. However, these new classes, called interfaces in JAVA, are abstract—this is a blind alley, as abstract classes and non-class-based attributes appear to be equivalent.

4.5 Problem complexity

In all variants, coloring is an instance of the graph coloring problem. When the minimization criterion is color number, this is the well known *minimum graph coloring problem.* Any graph may be the conflict graph of some class hierarchy—one only needs to add one common

	classes	methods	attributes	
typing			with abstract cl.	without a. c.
static	$k \leq 2$	$k \leq 2$	—	$k \leq 2$
dynamic	$k \leq 2$	—	—	—

Table 2: Conditions for polynomiality when there is perfect coloring

subclass to each connected pair. Therefore, with this minimization criterion, optimal coloring is NP-hard in the general case.

Now, when the minimization criterion is the total size of tables or, equivalently, the hole number, intuition tells us that this is not easier but a proof is required. [Pugh et Weddell, 1990 ; Pugh et Weddell, 1993] give proofs in the specific case of attribute coloring, but the proof is complete only for irregular colorings. We reformulate their results in our notations as follows.

Proposition 4.27. [Pugh et Weddell, 1993, Theorem 7] *Optimal irregular k-directional coloring is NP-hard, for all k.*

The proof proceeds by a reduction to the minimum graph coloring problem, where the colored graph is the conflict graph. As the proof relies on the construction of abstract classes (called *virtual classes* by Pugh and Weddell), it applies only to irregular colorings.

Regarding regular colorings, the situation slightly depends on k.

Proposition 4.28. [Pugh et Weddell, 1993, Theorem 6] *Optimal regular k-directional coloring is NP-hard, for all $k > 2$.*

Proof. It follows from Propositions 4.22 and 4.24 that the existence of perfect k-directional regular coloring polynomially reduces to k-colorability, which is NP-complete when $k > 2$ and polynomial otherwise. As the existence of perfect coloring easily follows from the computation of any optimal coloring, k-directional optimal coloring is NP-hard when $k > 2$. □

Regular k-directional coloring with $k \leq 2$ was the missing link in the Pugh and Weddell results, since their proof of Proposition 4.27 implies abstract classes, hence irregular coloring. When $k \leq 2$, the existence of a perfect regular coloring is polynomial. Therefore it has to be proven that computing an optimal regular coloring is NP-hard, when there is no perfect such coloring. Proving it for class coloring is sufficient, since class coloring is a special case of regular property coloring.

Proposition 4.29. [Takhedmit, 2003] *Optimal k-directional class coloring is NP-hard, for all $k \leq 2$, unless there is perfect k-coloring.*

The proof proceeds by a reduction to the *maximum 2-satisfiability problem.*

Overall, optimal k-directional coloring is always NP-hard, except when it is regular, $k \leq 2$ and there is perfect k-directional coloring (Table 2).

5 Heuristics and experiments

It follows from the complexity results that exact algorithms are intractable, hence heuristics are required. In the case where perfect coloring is tractable, these heuristics should provide perfect colorings when they exist. Several heuristics have already been proposed [Pugh et

```
Algorithm: SI-COLORING
Data: a hierarchy $(X, \prec_d)$
foreach class $x \in X$ in top-down topological ordering do
    if $x$ is a root (no superclass) then
        $\chi(x) \leftarrow 0$
    else
        $y \leftarrow$ direct superclass of $x$;
        $\chi^{\uparrow}(x) \leftarrow \chi^{*}(y)$;
        $\chi(x) \leftarrow \chi(y) + 1$
```

Figure 13: Single inheritance class coloring

Weddell, 1990 ; Pugh et Weddell, 1993 ; Ducournau, 1997 ; Ducournau, 2002b ; Takhedmit, 2003]. Their efficiencies were proven on large-scale class hierarchies. In this section, we describe two heuristic families for regular coloring and experimentation of one of them on large benchmarks. We also briefly consider heuristics for irregular coloring and incremental class coloring.

5.1 Heuristics

The first heuristics are rather naive—they are based on an adaptation of the single inheritance algorithm to multiple inheritance. They were first described in [Ducournau, 2001 ; Ducournau, 2003] (in French). The second ones are based on graph-theoretical results [Takhedmit, 2003]. We first describe class coloring—hence irrespective of static or dynamic typing—before generalizing to property coloring.

5.1.1 Naive heuristics

In single inheritance, the coloring algorithm for Cohen's test is quite simple—the color of a class is its depth in the hierarchy tree. As the intuitive recursive algorithm is not easily generalizable, we just transform it by ordering classes in some topological ordering from root(s) to leaves—in other words, superclasses are colored before subclasses (Figure 13). If there is more than one root, a virtual single root is added.

Schema of heuristics Such a simplicity cannot be kept with multiple inheritance, but it still gives a good starting point for generalizing. First, the structure of the hierarchy—i.e. core, crown, border, conflict graph—must be computed, before coloring the core which is the crux of the heuristics MI-COLORING (Figure 14). In all generality, coloring the core involves computing successive *partial colorings*, by repeatedly choosing a class and a color for this class. A class must be colored after all its superclasses. Therefore a set $Y_{\max}$ of $\preceq$-maximal uncolored classes must be maintained. At each step of the CORE-COLORING algorithm, any element in this set $Y_{\max}$ can be chosen to be colored. In single inheritance, the color choice amounts to taking the next color, i.e. adding 1 to the direct superclass color. With multiple inheritance, classes must maintain a set of *free colors*, i.e. colors which are not already used for superclasses ($\prec$) or conflicting classes ($\leftrightarrow$). Once a class x and a color k have been chosen, the choice must be propagated. First, x is removed from the superclasses of its subclasses, and the set of maximal classes is updated. Then, the color k is '*frozen*' in yet uncolored conflicting classes, because they share common subclasses with x. Note that propagation is done through two relationships, $\leftrightarrow$ and $\prec_r$, which are restricted to $X_{co} \backslash X_{bo}$ and updated in order to remove

```
Algorithm: MI-COLORING
Data: $(X, \prec_d)$ a hierarchy
begin
    $(X_{co}, X_{bo}, X_{cr}) \leftarrow$ computation of core, border and crown ;      /* initialization */
    $\leftrightarrow \leftarrow$ computation of the conflict set;
    core-coloring($X_{co} \backslash X_{bo}$) ;                                      /* core coloring */
    crown-coloring($X_{bo}$) ;                                                       /* border coloring */
    crown-coloring($X_{cr}$) ;                                                       /* crown coloring */
end
```

```
Algorithm: CORE-COLORING
Data: a conflict graph $(Y = X_{co} \backslash X_{bo}, \leftrightarrow)$;
the associated specialization $\prec_r = \prec_d / Y$;
$Y_{\max} \leftarrow \max_{\prec_r}(Y)$ ;                                        /* uncolored maximal classes */
while $Y_{\max} \neq \emptyset$ do
    $x \leftarrow$ choose-max-class($Y_{\max}$) ;                                 /* class choice */
    $\chi^{\uparrow}(x) \leftarrow$ inherits-colors($x$, $\prec_r$) ;             /* initializes color table */
    $\chi(x) \leftarrow$ choose-free-color($x$) ;                                 /* color choice */
    propagate-subclasses($x$, $\prec_r$) ;                                        /* propagates to subclasses */
    $Y_{\max} \leftarrow$ update($Y_{\max}$);                                     /* updates $Y_{\max}$ */
    propagate-conflicts($x$, $\leftrightarrow$) ;                                 /* propagates to conflicting classes */
```

```
Algorithm: CROWN-COLORING
Data: a poset $(Y, \prec_d)$
foreach class $x \in Y$ in topological ordering do
    if $x$ is a root (no superclass) then
        $\chi(x) \leftarrow 0$
    else
        $y \leftarrow$ direct superclass of $x$;
        $\chi^{\uparrow}(x) \leftarrow \chi^{*}(y)$ ;                             /* color table copy */
        if $\chi^{\uparrow}(x)$ has some holes then
            $\chi(x) \leftarrow$ any hole
        else
            $\chi(x) \leftarrow \max(\chi^{\uparrow}(x)) + 1$
```

Figure 14: Unidirectional class coloring—general algorithm in multiple inheritance

classes once they are colored. Therefore, the complex part of the algorithm is only a function of the core, not of the whole hierarchy. Finally, once the set of maximal classes is empty, the border and crown can be colored almost as in single inheritance (CROWN-COLORING).

Overall, the heuristics have two degrees of freedom, the choice of a maximal class among $Y_{\max}$ (`choose-max-class`) and the color choice among the free colors (`choose-free-color`). When a maximal class x is being colored, one distinguishes in its color table inherited colors ($\chi^{\uparrow}(x)$), holes, colors frozen by conflicting colored classes and the next free color:

$$\mathit{froz}(x) = \bigcup_{x \leftrightarrow y} \chi^{+}(y) \qquad \text{whereby } y \text{ is restricted to already colored classes}$$

$$\mathit{holes}(x) = [0, \max(\chi^{\uparrow}(x))] \setminus (\chi^{\uparrow}(x) \uplus \mathit{froz}(x)) \qquad \text{unidirectional}$$

$$\mathit{next}(x) = \min\Big( ]\max(\chi^{\uparrow}(x)), \infty[\setminus \mathit{froz}(x) \Big) \qquad \text{unidirectional}$$

So, in the unidirectional case, the free color is selected in $\mathit{holes}(x)$, if it is not empty, or is the single $\mathit{next}(x)$ color. There is a choice only when there are several holes—indeed, it is useless to choose the next color when there are holes, since another class ordering would produce the same coloring. This easily generalizes to k-directional coloring, by considering $\chi_i^{\uparrow}$, froz_i,

$holes_i$ and $next_i$ in each direction i. However, when there are no holes, there is yet a choice between the k directions.

In some sense, all possible colorings can be obtained by the heuristics which only exclude colorings with unnecessary holes. For instance, a random variant would involve taking the class at random among $Y_{\max}$, and taking a color at random among the free colors of the class. Of course, this does not ensure any formal statistical distribution, but it allows us to check that a particular behaviour does not happen only by chance [Ducournau, 2001 ; Ducournau, 2003].

Perfect coloring The heuristics compute a perfect coloring in the unidirectional case—i.e. when the hierarchy is in single inheritance. In the bidirectional case, an extra condition is required. A perfect coloring is computed if (i) `choose-max-class` chooses a maximal uncolored class x among classes conflicting with some previously colored class, if any, and if (ii) `choose-free-color` chooses the next color in the opposite direction to previously colored conflicting classes. This condition ensures that the bipartition will be exact if any exists. However, a preliminary bipartition gives a better result when the coloring is not perfect (see Section 5.1.2).

Weighted heuristics Weight naturally appears in the class choice and $Y_{\max}$ can be ordered by decreasing weight. The weight may be the subclass number, or the expected instance number for attribute coloring. Note that, as for all weighted heuristics, it is quite difficult to tune the weight in an optimal way. Consider simply, for instance, that the number of subclasses should not have the same weight according to their relative depth. More precisely, if the current class has q holes, any subclass of a crown subtree, at a relative depth d will fill $\max(q, d)$ holes. Therefore, q holes in a core class will entail exactly $\sum_{k=1..q-1}(q-k)n_k$ holes in a crown subtree rooted in the considered class with n_k classes at depth k.

Complexity analysis Initialization is mostly linear in the size of the graph, thus in $\mathcal{O}(|X|+|\preceq|)$, i.e. $\mathcal{O}(|X_{co}|+|\preceq_{co}|+|X_{cr}|)$. However, computing the conflict graph is in $\mathcal{O}((|X_{co}|+|\preceq_{co}|)|X_{co}|)$, hence $\mathcal{O}(|X_{co}|^3)$ in the worst case. Coloring the core is in $\mathcal{O}(|X_{co}|^2)$ if one assumes that `choose-max-class`, `choose-free-color` and the computation of free colors are time-constant, for instance in case of arbitrary choice (first found). Indeed, all operations—i.e. propagation, color inheritance—are in $\mathcal{O}(|X_{co}|)$. However, better heuristics may consider the uncolored classes conflicting with the current one, which might yield $\mathcal{O}(|X_{co}|^3)$. Anyway, this is not more than for conflict graph. The second part of the heuristics is obviously in $\mathcal{O}(|X_{bo}|+|X_{cr}|)$ and the overall complexity is $\mathcal{O}(|X_{co}|^3+|X_{cr}|)$.

5.1.2 Graph theoretical heuristics

More sophisticated heuristics are based on various other graph optimization problems, which unsurprisingly are all NP-hard.

Unidirectional coloring and maximum independent sets [Takhedmit, 2003] proposes weighted heuristics for unidirectional coloring, based on *independent sets*. Indeed, an alternative view of coloring is to consider it as a partition in *maximal independent sets* (Definition 4.15).

```
Algorithm: CORE COLORING IS
Data: Class hierarchy $(X, \prec_d)$;
a subset $Y \subset X$ and conflict relationship $\leftrightarrow$
Attribute to each $x \in Y$ a weight $w(x)$ ;                      /* weights */
$S \leftarrow \emptyset$ ;                        /* ordered sequence of independent sets */
$l \leftarrow 0$ ;                                         /* length of the sequence */
foreach $x \in Y$ sorted by decreasing weight do
    $m \leftarrow 0$;
    for $k$=1 to $l$ while $m$=0 do
        if $x$ is not $\prec$- or $\leftrightarrow$-related to $S[k]$ then
            $S[k] \leftarrow S[k] \uplus \{x\}$;                      /* independent set */
            $m \leftarrow k$;
    if $m$=0 then
        $m, l \leftarrow l + 1$;
        $S[m] \leftarrow \{x\}$
    reorder(S, S[m]);                           /* reorder by decreasing weights */
for $k$=1 to $l$-1 do
    permute(S[k],S[k+1]);               /* try permutations to make $S[k]$ heavier */
for $k$=1 to $l$ do
    foreach $x \in S[k]$ do $\chi(x) \leftarrow k$
```

Figure 15: Unidirectional class coloring with independent sets. Note that $\diamond$ is not explicitly computed and independent sets are incrementally checked—$S[k]$ is an independent set and only possible relations between x and elements in $S[k]$ need to be checked.

Maximum independent set is another NP-hard problem—hence greedy heuristics are used. An optimization involves permutation of classes between two independent sets, when this increases the weight of the heavier one. The heuristics CORE-COLORING-IS presented in Figure 15 give sound results if the weight w is somewhat 'additively monotonous', i.e. $w(x) > \sum_{y \prec_d x} w(y)$ when x is a tree root, hence Y is ordered by a top-down topological sorting of $\prec$. This is the case when $w(x)$ is the number of subclasses of x, or its number of instances. As these heuristics involve only coloring the core, perfect coloring is yielded when it exists, i.e. in case of single inheritance.

The complexity of the heuristics for computing independent sets is $\mathcal{O}(|X_{co}|^2)$, i.e. less than that of conflict graph computation. So, the overall complexity is the same as that of the naive heuristics. The heuristics proposed by [Vitek *et al.*, 1997] are akin to CORE-COLORING-IS—independent sets are called *buckets*[9] and the notions of core, crown and border are used, under the respective terms of 'spine', 'plain' and 'join'.

Multi-directional coloring and maximum k-cut k-directional coloring amounts to unidirectional coloring of k sub-hierarchies. Therefore, the point is to partition $X_{co} \backslash X_{bo}$ into k blocks which minimize the number of conflicts inside the blocks. The *maximum k-cut problem* offers a formalization of this problem. It involves partitioning an undirected graph in k blocks maximizing the number of crossing edges. This is yet another NP-hard problem. [Takhedmit, 2003] proposes heuristics based on the family of meta-heuristics GRASP (*Greedy Randomized Adaptive Search Procedure*) [Festa *et al.*, 2002].

When $k = 2$, an exact bipartition algorithm may be tried before applying the heuristics—this will give, in case of success, a perfect bidirectional coloring. However, *maximum 2-cut*, aka *maximum bisection*, is also NP-hard in the general case—hence, if the conflict graph is

[9] In the literature on coloring, independent sets are often called 'buckets', 'slices' or 'layers'.

```
Algorithm: IRREGULAR-COLORING
Data: a class hierarchy $(X, \prec_r)$ with conflict relationship $\leftrightarrow$;
a set $P_x$ of non-class-based properties associated with each class $x$;
a number $n_x$ of class-based properties associated with each class $x$;
$Y_{\max} \leftarrow \max_{\prec_r}(X)$ ;                          /* uncolored maximal classes */
while $Y_{\max} \neq \emptyset$ do
    $x \leftarrow$ choose-max-class($Y_{\max}$) ;                          /* class choice */
    $\chi^{\uparrow}(x) \leftarrow$ inherits-colors($x$, $\prec_r$) ;           /* initializes color table */
    $\chi^{+}(x) \leftarrow$ reserved-colors($x$)
    while $P_x \neq \emptyset$ do
        $p \leftarrow$ choose-property($P_x$) ;                          /* property choice */
        $P_x \leftarrow P_x \backslash \{p\}$
        $k \leftarrow$ choose-one-free-color($x$, $c(x)\backslash\{x\}$) ;             /* color choice */
        propagate-introduction($k$, $c(x)\backslash\{x\}$);     /* propagates to introduction classes */
        $\chi^{+}(x) \leftarrow \chi^{+}(x) \uplus \{k\}$
    $K \leftarrow$ choose-n-free-colors($x$,$n_x$) ;                             /* color choice */
    $\chi^{+}(x) \leftarrow \chi^{+}(x) \uplus K$
    propagate-subclasses($x$, $\prec_r$) ;                        /* propagates to subclasses */
    $Y_{\max} \leftarrow$ update($Y_{\max}$);                                  /* updates $Y_{\max}$ */
    propagate-conflicts($x$, $\leftrightarrow$) ;              /* propagates to conflicting classes */
```

Figure 16: Irregular coloring—general algorithm in multiple inheritance

not bipartite, the heuristics is applied.

5.1.3 Property coloring

Property coloring involves the same kind of techniques as class coloring. However, the property coexistence graph is an order of magnitude larger than the class coexistence graph. This is certainly the reason for the relative failure of the tests of [André et Royer, 1992], though Pugh and Weddell [1990 ; 1993] were more successful. Therefore, a sensible solution is to rely on class coloring heuristics—i.e. on class hierarchy and conflict graph. This is rather straightforward for regular coloring, but it is also possible for non-class-based properties as the SMALLTALK hierarchies colored in [André et Royer, 1992 ; Ducournau, 1997].

Regular coloring The naive heuristics of Figure 14 extend easily to property coloring—for each class x, the algorithm allocates as many colors as there are properties introduced by x, instead of a single one. Regarding n-directional coloring, the properties introduced by a class can be colored in different directions, but only when there are some holes to fill. Anyway, the preliminary partition is only based on the conflict graph, possibly restricted to classes introducing properties.

Irregular coloring When a property can be introduced by more than one class, relying only on the class hierarchy and conflict graph is not enough. However, there are likely not many non-class-based properties—only 27% in the SMALLTALK hierarchy tested by [Ducournau, 1997]—so the main work can rely on class hierarchy. The heuristics principle remains, but its complexity is markedly increased (Figure 16). First, one must distinguish between *class-based* and non-class-based properties—introducing classes have to be associated with each property of the second kind. Coloring such a property involves selecting a color that is free not only in all introduction classes but also in their superclasses (`choose-one-free-color`). Propagation to yet uncolored classes is also more complex: the selected color is 'reserved' in all other introduction classes, and 'frozen' in all their super-classes and conflicting classes

```
Algorithm: INCR-CLASS-COLORING
Data: a class hierarchy $(Y, \prec_d)$;
a loaded class $x \in Y$, minimal in $(Y, \prec_d)$;
a partial class coloring $\chi$ on the filter $Y \backslash x$
begin
    $super_d \leftarrow \{y \in Y \mid x \prec_d y\}$;                     /* direct superclasses */
    $k \leftarrow \max(\chi(Y \backslash x))$;
    if $super_d = \emptyset$ then
        $\chi(x) \leftarrow 0$;                                  /* $x$ is a root (no superclass) */
    else
        if $super_d = \{y\}$ then
            $\chi^{\uparrow}(x) \leftarrow \chi^*(y)$;                           /* like crown coloring */
        else
            $C[0..k]$ an array of empty sets ;                    /* multiple inheritance */
            foreach class $z \in Y$ such that $x \prec z$;               /* all superclasses */
            do
                $C[\chi(z)] \leftarrow C[\chi(z)] \uplus \{z\}$
            $cs \leftarrow \{c \mid |C[c]| > 1\}$ ;                                /* conflict set */
            if $cs \neq \emptyset$ then
                $Z \leftarrow \emptyset$;
                foreach color $c \in cs$ ;                           /* conflict resolution */
                do
                    $u \leftarrow$choose-unchanged($C[c]$) ;                   /* left unchanged */
                    $Z \leftarrow Z \uplus C[c] \backslash \{u\}$;
                    $C[c] \leftarrow \{u\}$
                $C \leftarrow$ recompute-colors($Z$)
            $\chi^{\uparrow}(x) \leftarrow \{c \mid C[c] \neq \emptyset\}$;
        if $\chi^{\uparrow}(x)$ has some holes then
            $\chi(x) \leftarrow$ any hole
        else
            $\chi(x) \leftarrow \max(\chi^{\uparrow}(x)) + 1$;
            if $\chi(x) > k$ then
                reallocate all color tables;                 /* only with fixed-size tables */
end
```

Figure 17: Load-time unidirectional class coloring—general algorithm (from Palacz and Vitek [2003]). The algorithm is straightforward unless there are conflicts. Of course, the choice of the unchanged class and the computation of new colors is a matter of global optimization.

(propagate-introduction). Of course, reserved colors are not free and must be included in the color table once a class is colored. In this way, the algorithm limits propagation to the detriment of free color testing, which must look up in superclasses. An alternative would be to propagate down allocated colors to subclasses. Finally, it is useless to design a specific algorithm for the crown, since non-class-based property can be introduced anywhere.

5.1.4 Incremental class coloring

In their proposition, [Palacz et Vitek, 2003] consider a special case of class coloring where color tables have fixed size. As aforementioned, this is a common way of saving on bound checks in Cohen's test, to the detriment of space. However, this space overhead can be partially counter-balanced by encoding class identifiers on single bytes, instead of half-words, so that the identifiers are unique only among classes with the same color. Moreover, since incremental coloring is restricted to classes, space concerns are less urgent.

The heuristics are sketched in Figure 17. Anyway, whether the size is fixed or variable, coloring heuristics present different degrees of efficiency. Firstly, the new loaded class may have a single direct superclass. Otherwise, the first step involves inheriting the colors of all its superclasses—this is a *partial coloring.* A conflict may occur when two superclasses share the same color. When there is no conflict, extending the partial coloring is straightforward. The only point is to select a free color—i.e. a free position in the fixed-size table. So this is very efficient when there is no conflict and when the table is not full—an $\mathcal{O}(k+n)$ operation, where n is the number of superclasses of the considered class and k is the color table size. When the table is full, the fixed size must be incremented and all color tables must be reallocated—this is an $\mathcal{O}(kN)$ operation, where N is the number of classes. With a variable-size color table, this is even simpler since there is no need for propagating the new color to already loaded classes—hence this remains an $\mathcal{O}(k+n)$ operation.

When there are conflicts, extending the partial coloring is no longer possible. The color of one of the two conflicting classes must be changed, in a way which must be compatible with all subclasses of the changed class. Therefore, color recomputation (`recompute-colors` in Figure 17) does not extend a partial coloring—i.e. by examining only superclasses—but must take into account all subclasses and all conflicting classes of those that must be recolored. The optimum is of course NP-hard but simple heuristics are possible. Their exact worst-case cost cannot be estimated since it depends on the possible propagation that is done when a color is assigned to a class. However this worst-case cost at load-time must be at least $\mathcal{O}(nN)$. Indeed, an incremental computation of the conflict graph is first required, with $\mathcal{O}(n^2)$ complexity since all superclasses might be conflicting, and all other classes might already conflict with the superclasses.

5.2 Benchmarks

We tested coloring on several large benchmarks commonly used in the object-oriented implementation community[10], e.g. by [Vitek *et al.*, 1997 ; Gil et Zibin, 2005]. The benchmarks are abstract schemata of large class libraries, from various languages: Java (from IBM-SF to HotJava in Table 3), Cecil (Cecil, Vortex3), Dylan, Clos (Harlequin), Self, Eiffel (SmartEiffel), Eiffel-like (Lov and Geode) and Prm (PRMcl). All benchmarks are classical, except PRMcl which is the Prm compiler-linker [Privat et Ducournau, 2005 ; Privat, 2006]. Here, all benchmarks are multiple inheritance hierarchies and, in Java benchmarks, there is no distinction between classes and interfaces. According to our conclusions on n-dimensional coloring, we restricted our tests to uni- and bi-directional coloring.

5.2.1 Hierarchies and conflict graphs

Table 3 presents statistics on class hierarchies and conflict graphs: (i) the total number of classes and the subset in the core, (ii) the number of conflicting classes and the subsets which *introduce* methods (wm) or attributes (wa), (iii) the average, maximum and total number of conflicting edges, and (iv) statistics on connected components. A first observation is that the core, i.e. the set of classes that are in multiple inheritance, is quite a bit smaller than the whole hierarchy, while being of the same magnitude in many benchmarks. More precisely, the crown is larger than the core in all hierarchies, except Lov-obj-ed, Geode and IBM-SF, where multiple inheritance is the most intensively used. So many benchmarks are representative of

[10] Many people contributed to these benchmarks, including Karel Driesen and Jan Vitek: a recent repository was Yoav Zibin's website, `http://www.cs.technion.ac.il/~zyoav/`. They are also available on the author's website `http://www.lirmm.fr/~ducour/`.

	class		conflict graphs						connected components				
	number		vertices			edges			bipartite		non-bipartite		
	total	core	tot	wm	wa	avg	max	total	#	size	#	size	max
IBM-SF	8793	4770	2566	1852		13.3	2057	17099	3	8	2	2558	2550
JDK1.3.1	7401	1512	762	572		6.6	205	2529	21	61	8	701	596
Orbix	2716	271	177	106		4.7	110	416	7	23	1	154	154
Corba	1699	383	213	92		7.6	144	810	0	0	1	213	213
Orbacus	1379	502	315	184		6.5	166	1025	3	9	1	306	306
HotJava	736	217	108	91		10.8	54	583	2	8	2	100	95
Cecil	932	306	167	100		6.1	46	511	6	20	2	147	130
Dylan	925	65	35	17		3.1	8	54	5	20	1	15	15
Harlequin	666	278	169	23		15.8	71	1331	9	37	5	132	95
Java1.6	5075	1422	645	506	155	6.6	283	2124	15	45	4	600	574
Geode	1318	989	500	373	206	22.5	258	5613	1	2	3	498	482
Unidraw	614	25	14	8	9	2.1	5	15	1	9	1	5	5
Lov-obj-ed	436	271	159	144	73	15.6	81	1241	1	2	1	157	157
SmartEiffel	397	67	26	21	7	2.8	8	36	2	5	1	21	21
PRMcl	479	133	81	56	34	3.4	12	139	6	26	3	55	29

Table 3: Class hierarchies and conflict graphs. Columns 2-3 display the number of classes in the whole hierarchy and in the core. Columns 4-6 present the number of non-isolated vertices in the conflict graph and distinguish classes which introduce methods ('wm') or attributes ('wa'). Columns 6-9 present the number of edges—the average and maximum per class and the total. Columns 10-14 give statistics on connected components in the conflict graph—number, total and maximum size—according to whether they are bipartite or not.

an heavy use of multiple inheritance and heuristics will benefit from being restricted to the core.

The conflict graph is even smaller, and the restriction to classes introducing properties is effective—columns 'wm' and 'wa'. Regarding the conflict graph, the Table presents statistics on the number and size of its connected components, according to whether they are bipartite or not. It appears that all conflict graphs consist of a single large non-bipartite connected component, plus some other small components—some of which are bipartite, but they are quite small. A deeper analysis [Ducournau, 2001], not reported here, shows that the same phenomenon arises if one considers 2-connected components[11]. This proves that, on all the considered benchmarks, bidirectional class coloring will be far from perfect.

5.2.2 Class coloring

Table 4 presents the class coloring results, which should be compared with the size of the hierarchy ($|X|$, $|\preceq|$ and their ratio). The first four columns give the size of the specialization graph together with the average and maximum number of indirect superclasses per class. The three following columns give the hole numbers, according to the minimization criterion—i.e. color number, unidirectional and bidirectional colorings. Let us examine the PRMcl example: when minimizing color#, the color tables have 2203 ($|\preceq|$) occupied slots, plus 3545 holes, hence a total of 5748=12*479 ($\max *|X|$). Minimizing the color number is not as good as uni- or bi-directional coloring from a spatial standpoint. This implies that, when applied to classes

[11] Each connected component can be colored regardless of the other components. This is no longer true for 2-connected components, but two of them share at most one class. Hence, the complexity is mostly in the size of the 2-connected components.

	hierarchy				total hole number			bidirectional		
			$\| \preceq \|/\|X\|$		minimization criterion			hole	size	
	$\|X\|$	$\| \preceq \|$	avg	max	color#	uni-	bi-	rate	avg	max
IBM-SF	8793	80860	9.2	30	182930	19746	4507	.06	9.7	30
JDK1.3.1	7401	32480	4.4	24	145144	4331	1307	.04	4.6	24
Orbix	2716	7560	2.8	13	27748	347	18	.00	2.8	13
Corba	1699	6551	3.9	18	24031	533	115	.02	3.9	18
Orbacus	1379	6244	4.5	19	19957	935	181	.03	4.7	19
HotJava	736	3768	5.1	23	13160	1210	248	.07	5.5	23
Cecil	932	6032	6.5	23	15404	573	141	.02	6.6	23
Dylan	925	5097	5.5	13	6928	87	4	.00	5.5	13
Harlequin	666	4493	6.7	31	16153	1599	545	.12	7.6	31
Java1.6	5075	27309	5.4	23	89416	3314	1155	.04	5.6	23
Geode	1318	18442	14.0	50	47458	10005	4477	.24	17.4	50
Unidraw	614	2468	4.0	10	3672	15	1	.00	4.0	10
Lov-obj-ed	436	3707	8.5	24	6757	1838	1283	.35	11.4	24
SmartEiffel	397	3428	8.6	14	2130	36	4	.00	8.6	14
PRMcl	479	2203	4.6	12	3545	131	31	.01	4.7	12

Table 4: Class coloring. Columns 2-5 display statistics on the class hierarchy. Columns 6-8 present the total hole number according to the minimization criterion. The last part presents, in the bidirectional case, the global hole rate and the average and maximum color table size per class—they must be compared with the average and maximum superclass number, columns 4-5.

only, as in [Vitek *et al.*, 1997 ; Palacz et Vitek, 2003], variable length tables are cheaper than fixed-size tables, even with byte-encoding of class identifiers. Unidirectional coloring is better and, unsurprisingly, bidirectional coloring is the best. Finally, the hole rate appears to be quite small, less than 10% in most cases and 35% in the worst case. The last two columns give the average and maximum size of the color tables, which must be compared to the superclass number (columns 4 and 5).

From a graph-theoretic standpoint, one must observes that, in all cases, the heuristics computes the exact optimal color number (the so-called *chromatic number*)—the two columns 'max' are equal. This only proves that all these benchmarks are easy instances of the minimum graph coloring problem.

5.2.3 Property coloring

Here, we must recall the terminology proposed in Section 2.2.1: methods and attributes are signatures *introduced* in some class, and only *virtual* functions are considered. We only investigated *regular coloring*—i.e. each set of properties is interpreted as if it were in static typing so that each property is introduced by a single class. Table 5 presents statistics of methods (and attributes), introduced or known per class, together with method (and attribute) coloring. All data are 2-fold, namely average and maximum per class, and represent cardinal numbers (#) or rates (/) relative to the total useful size of method tables (or object layouts). Rates are ratios, not percentages. The difference between columns 'avg# color number' and 'avg# known' is the average hole number.

Only bidirectional method coloring is displayed, since it is better and does not raise any problem. Note that in most benchmarks the maximum color number is exactly the maximum number of known methods—this means that the heuristics give the optimal solution of the

	methods				method coloring				SMI	
	introduced		known		color number		hole rate		method table	
	avg#	max#	avg#	max#	avg#	max#	avg/	max/	avg#	max#
IBM-SF	2.8	257	44.9	346	62.9	397	.40	15.9	231.3	2063
JDK1.3.1	1.3	149	19.2	243	19.8	243	.03	15.2	72.4	1391
Orbix	0.4	64	8.3	109	8.4	109	.00	2.1	23.0	534
Corba	0.4	43	8.0	67	9.9	81	.24	5.0	26.9	427
Orbacus	1.2	74	18.0	137	18.3	137	.01	2.3	68.3	761
HotJava	1.8	80	34.2	189	36.0	189	.05	17.8	134.8	817
Cecil	2.9	61	78.7	156	80.2	156	.02	0.6	441.5	2058
Dylan	0.9	64	77.1	139	77.2	139	.00	0.2	335.8	1073
Harlequin	0.6	62	34.8	129	35.0	129	.01	0.3	219.3	977
Java1.6	4.4	286	37.5	670	38.2	670	.02	4.5	136.5	3873
Geode	6.1	193	231.8	880	291.0	892	.26	16.1	1445.6	10717
Unidraw	2.9	103	24.1	124	24.1	124	.00	0.0	68.9	318
Lov-obj-ed	8.3	117	85.9	289	113.5	289	.32	4.7	422.1	1590
SmartEiffel	12.2	222	135.3	324	135.4	324	.00	0.2	743.5	1576
PRMcl	4.9	115	78.3	208	83.1	209	.06	2.7	316.5	1918

	attribute number				color		holes						SMI		
	introduced		known		number		bidirect.			unidirect.			subobject#		
	avg#	max#	avg#	max#	avg#	max#	avg/	max/	max#	avg/	max/	max#	avg/	max/	max#
Java1.6	1.6	55	5.4	137	6.4	138	.00	0.0	0	.00	0.0	0	.81	7.3	22
Geode	2.2	182	10.9	217	12.7	218	.13	7.0	21	.18	19.7	59	1.19	10.0	49
Unidraw	2.6	36	8.3	47	9.3	48	.05	1.0	1	.00	0.5	2	.36	2.0	9
Lov-obj-ed	2.9	74	8.2	105	9.3	106	.07	3.0	3	.21	6.0	29	.92	10.5	23
SmartEiffel	2.5	39	4.9	44	5.9	45	.00	0.0	0	.00	0.0	0	1.55	5.0	13
PRMcl	1.2	28	5.0	29	6.1	30	.06	1.3	4	.03	2.0	8	.72	3.0	11

Table 5: Property coloring. Each table first displays the number of properties (methods or attributes) introduced or known per class, then the size of bidirectional tables—which must be compared with the number of known propertiess—with the associated hole rate. In the attribute case, unidirectional results are also displayed. The last part presents the method table size or the subobject number in 'standard' multiple inheritance (SMI), i.e. C++.

underlying minimum graph coloring problem. The hole rate is somewhat larger than for class coloring—this is likely because the heuristics are less optimal as there are more choices. The last columns give the corresponding table sizes in the 'standard' multiple inheritance implementation (SMI), based on subobjects[12]. While 'SMI avg#' is between 2- and 7-fold larger than 'color avg#', 'known methods max#' is often larger than 'SMI avg#'—once again, this is against minimizing the color number.

Attribute coloring has been tested on the few benchmarks which include attribute definition data. However, the heuristics do not take the expected number of instances into account, since such data were not available. The statistics are analogous to that on methods, with a few differences—the unidirectional case is added and, in the SMI numbers, the subobject number replaces the table size, and the maximum number of holes or subobjects is added ('max#'). The numbers from the last 3 groups of columns are respectively comparable—they represent the number of useless fields in the object layout or the ratio w.r.t. the useful fields. Moreover, when negative colors are used, the extra pointer required for garbage collection (Figure 8) is considered as a hole, and the 'color number' column includes all of the pointers at the method table. One observes that the solution to the underlying minimum graph coloring problem is still optimal—known attribute and maximum color numbers differ by exactly one in all benchmarks. Moreover, both Java1.6 and SmartEiffel have a perfect coloring, because Proposition 4.24 applies. The comparison between the average rates ('avg/') of all 'useless' fields in the object layout is markedly in favor of coloring. The number of SMI subobjects has actually the same order of magnitude as the number of known attributes. Hence, SMI—in the ALL implementation (see Section 2.2.2)—roughly doubles the object size and coloring markedly reduces the dynamic space overhead—and of course it markedly improves all other aspects. Other SMI implementations, with extra compiler-generated fields allocated in the object layout—like VBPTRs or with the ARM implementation—would be markedly more space-consuming. Anyway, attribute coloring is a significant improvement w.r.t. SMI implementations, mostly because holes do not concern a large part of each hierarchy.

The comparison between uni- and bi-directional coloring shows that the gain from bidirectionality is mostly counterbalanced, on average, by the extra method table pointer. However, the attribute case is more demanding and one must also take care of the maximum hole rate and number, which represent the worst case in a hierarchy—both are markedly in favor of bidirectionality. Let us consider the bidirectional data for Geode. The average hole rate is .13—i.e. assuming that all classes have exactly the same instance number, the total hole rate is 13%. This would be an acceptable overhead. However, the maximum hole rate is up to 7—this means that there is some class in the Geode hierarchy which has x useful and $7x$ useless fields. The maximum hole number confirms the comparison—it is 21 in bidirectional coloring, versus 59 in unidirectional and 49 extra subobjects in SMI. Such a worst case might be unacceptable if the considered class is intensively instantiated—imagine that the Lisp cells (instances of the `cons` class) occupy 24 words—i.e. 3 useful fields and 21 useless ones—instead of 3! Therefore, attribute coloring should be coupled with profiling or *accessor simulation* might be preferred.

[12] Remember that SMI is the C++ implementation when the keyword `virtual` is used for all inheritance relationships. This is the only way to define fully reusable classes but not the common C++ programming style, so SMI numbers do not reflect actual C++ programs.

	initialization		coloring			total
	read	init	conflict	bipartite	color	avg/class
IBM-SF	1281	647	280	547	5091	0.9
JDK1.3.1	438	364	48	104	1129	0.3
Orbix	113	88	8	16	238	0.2
Corba	110	97	24	26	237	0.3
Orbacus	77	56	15	37	183	0.3
HotJava	43	74	8	22	88	0.3
Cecil	65	42	11	18	149	0.3
Dylan	37	34	2	2	82	0.2
Harlequin	52	33	14	62	116	0.4
Java1.6	415	796	58	227	1031	0.5
Geode	139	148	73	259	534	0.9
Unidraw	65	22	9	1	76	0.3
Lov-obj-ed	82	35	14	50	140	0.7
SmartEiffel	56	17	2	2	79	0.4
PRMcl	72	19	3	7	67	0.3

Table 6: Performance of bidirectional class and method coloring (in ms, on an Intel® Core^{TM} 2 CPU T7200 at 2.0 GHz). The first five columns display the duration time of successive phases and the last column is the total duration per class.

5.2.4 Synthesis

Bidirectional coloring is clearly the best approach. When considering classes and methods, the gain w.r.t. unidirectional coloring is not essential, but uni- and bi-directional coloring are markedly better than fixed-size tables. In contrast, bidirectionality is likely essential for attribute coloring as it markedly reduces the worst-case overhead.

5.2.5 Performance

Table 6 describes the performance of the heuristics on a Intel® Core^{TM} 2 CPU T7200 at 2.0 GHz. Coloring is computed on a platform written in Common Lisp and Clos, dedicated to the simulation of various implementation techniques. This platform was used for other simulations [Ducournau, 2002a ; Ducournau, 2008]. Many other statistics are computed and coloring was not specially optimized—on the contrary, the general algorithm (Figure 14) is designed for testing various heuristics. Therefore, an operational implementation would be substantially more efficient. The bidirectional coloring heuristics use a preliminary bipartition and methods and classes are colored together. The table displays the duration time of five main steps. The first steps are not specific to coloring: 'read' is the input phase, which reads the class schemata; 'init' manages inheritance and computes the core, border and crown. The last steps are specific: 'conflict' computes the conflict graph, 'bipartite' makes a bipartition of the conflict graph and 'color' computes the bidirectional coloring. Overall, the coloring time is not much greater than the read and initialize phases, and the overall time is less than 1 ms per class.

6 Related works

There is quite substantial literature on the implementation of object-oriented languages. Problems arise when dynamic typing or multiple inheritance are considered. Separate compilation and dynamic loading worsen the situation. We just consider, here, some related works that are either close to coloring or true alternatives.

Multiple selection In many dynamically typed languages, method invocation is specified with *multiple selection* (aka *multi-methods*), which involves the dynamic type of all parameters, not only that of the receiver. Clos, Cecil and Dylan are typical examples of this approach. Methods are now orthogonal to classes—they are called *generic functions* in Clos—and dispatch tables are attached to them, not to classes. As for usual method invocation in dynamic typing, the most common technique is based on hashtables and caches [Kiczales et Rodriguez, 1990]. [Amiel *et al.*, 1994 ; Dujardin *et al.*, 1998] propose a constant-time implementation based on a generalization of the coloring principle, with several indirections. The dispatch table of a generic function of arity n is an n-dimensional array, indexed by classes. This array can be compressed by grouping classes which determine identical hyperplanes in the array. This is a matter of coloring.

C++ implementations The specification and implementation of multiple inheritance in C++ has been the source of a lot of papers. [Sweeney et Burke, 2003] examines the best ways to distribute compiler-generated fields between the object layout and the method tables. *Devirtualization* is a global optimization which aims at removing all unnecessary `virtual` keywords—in both meanings—from the class definition of a given program [Gil et Sweeney, 1999 ; Eckel et Gil, 2000]. This global optimization could be applied in a global compilation setting but it seems difficult to apply it at link-time without substantially modifying C++ compilers.

Row displacement In a global setting, an alternative to coloring has been proposed, *row displacement* [Driesen, 1993 ; Driesen et Hölzle, 1995 ; Driesen, 2001]. This sparse table compression technique, from [Tarjan et Yao, 1979], involves superposing rows of the large class-selector matrix in such a way that two occupied entries do not coincide. It works well for methods and roughly achieves the same compactness as coloring. Regarding subtype tests, row displacement has not yet been considered but it could obviously work, with the same precautions as for Cohen's test to avoid bound checks. Finally, row displacement does not apply to attributes—hence, it should be coupled with accessor simulation and it is more appropriate for static typing.

Rows may be class-based or selector-based. According to Driesen, selector-based row displacement achieves better compression than class-based row displacement, which is better than selector coloring in its original variant, i.e. unidirectional and minimizing the color number. As compared to our results, it appears that selector-based (resp. class-based) row displacement and bi-directional (resp. uni-directional) coloring are quite comparable. Overall, row displacement might offer an alternative to class and method coloring, with exactly the same generated code and similar table sizes. Selector-based displacement achieves a better compression rate but is less adapted to dynamic class loading. Hence, it should be globally computed at link-time, like coloring. On the contrary, class-based rows might be considered in a dynamic loading setting, though the cost of inserting a new row may not be negligible, and the compression rate is not very good.

Binary tree dispatch (BTD) Method tables represent the most intuitive and common technique for implementing late binding, but it is not the only one. Cache techniques used in dynamically typed languages have been extended to *polymorphic caches* when several types are expected [Hölzle *et al.*, 1991]. Usually, cache-based techniques must provide a solution for cache misses—e.g. hashtables—when the actual type is not among the expected types. However, in a global setting, a static type analysis—coupled with dead code elimination—allows computing of the *concrete type*, i.e. the set of possible receiver types in all executions of the considered program. In this context, all types expected at a given call site can be exhaustively tested. An efficient organization of the tests is a balanced binary tree. The technique also applies to attributes and subtype tests. This is the basis of the GNU Eiffel compiler, Smart Eiffel (formerly Small Eiffel) [Zendra *et al.*, 1997 ; Collin *et al.*, 1997]. Of course, binary tree dispatch is not constant-time, but its average behavior is considered as very good. Two arguments are in favor of it: (i) modern processors are equipped with prediction capabilities for conditional branching which outmatch the corresponding capabilities for indirect branching; (ii) most method call sites are *monomorphic* or weakly polymorphic (aka *oligomorphic*). Against it, when call sites are highly polymorphic (aka *megamorphic*), the technique can be quite inefficient. The efficiency of BTD depends on the class IDs. In single inheritance, with a preorder class numbering, BTD can be thought of as the inlined form of *interval containment*, the technique proposed by [Muthukrishnan et Muller, 1996]—this ordering likely ensures a quasi-optimal test number. However, the extension to multiple inheritance proposed by [Gil et Zibin, 2007] does not seem to apply to BTD, i.e. it does not seem to compile well.

In all compilation settings, an intraprocedural type analysis can detect a small part of monomorphic call sites that can be compiled into a static call. For the other call sites, when an interprocedural type analysis is possible—i.e. with global compilation, as in Smart Eiffel, or global linking, as in Prm—the best method invocation implementation involves binary tree dispatch for oligomorphic sites and coloring for megamorphic sites. The only point is to finely tune the threshold between oligo- and mega-morphism.

Two-way coloring and mixed techniques [Huang et Chen, 1992] propose a generalization of selector coloring where both methods and classes are colored. Two classes C and D can share the same color if their method tables are compatible, i.e. for each entry i, either $tab_C[i] = tab_D[i]$ or one of them is empty—therefore, both method tables can be merged. It works well for message sending in either static or dynamic typing but it does not apply to object layout nor subtype testing. [Vitek et Horspool, 1994] generalize this idea by accepting merging even when both method tables have a few conflicting entries—each entry of the resulting table contains the address of a method or of a binary tree dispatch. This technique aims at reducing the total size of method tables but the size of the trees mostly counterbalances the gain in method tables [Vitek et Horspool, 1996 ; Ducournau, 1997]. However, this approach provides an incremental variant of coloring, which can be interesting in a dynamic typing setting since message sending now requires an additional parameter, i.e. the selector, which must be tested to detect the ‘message not understood’ error. This extra parameter may also be used in dispatch trees when there are several possible selectors for one table entry. However, in static typing, this would add uniform and significant overhead. [Alpern *et al.*, 2001] takes a similar approach for method invocation on interface-typed receivers in Java—the so-called `invoke-interface` operator—with hashtables where collisions are compiled into tree dispatches. [Queinnec, 1998] proposes a dual approach, where dispatch is organized as a binary tree some nodes of which contain an array. In all cases, this does not work for subtype tests—on the contray, subtype testing may be required to select the right branch in the dispatch tree.

In their 1996 paper, Vitek and Horspool try to correct the drawbacks of their approach by proposing to partition the property set in order to allow better sharing of the same method tables between several classes. This is akin to multi-dimensional coloring, with the same drawbacks—i.e. one extra memory access. It might be considered as an optimization of multi-dimensional coloring—i.e. the optimization criterion would be maximizing sharing. There are, however, other approaches for higher compactness, but they markedly degrade the time efficiency. For instance, [Muthukrishnan et Muller, 1996] propose an implementation with linear-size method tables, in $\mathcal{O}(N+M)$, but $\mathcal{O}(\log\log N)$ method invocation time, where N is the number of classes and M is the number of method definitions.

Subtype tests Class coloring is likely not the most efficient subtype test. Its benefits are that it is efficient, simple, inlinable and generalizable to method invocation and attribute access. In another setting, e.g. knowledge representation instead of object-oriented programming, another technique might be preferred.

A classic alternative to Cohen's test is *relative numbering* [Schubert *et al.*, 1983], which involves a very simple double numbering of classes. Of course, the technique applies only in single inheritance and, contrary to Cohen's test, it is not incremental. Cohen's display and Schubert's numbering are the best techniques from the time standpoint and they differ by the encoding direction. A class encodes its superclasses in the former, its subclasses in the latter. A bottom-up encoding is in favour of incrementality. Both techniques have been generalized to multiple inheritance, the former with coloring and the latter with *range compression* [Agrawal *et al.*, 1989], where several intervals can be associated with each class. The test is no longer time-constant. Several authors propose to combine both approaches in such a way that each class is associated with: (i) a *color* (equivalently, the set of classes is partitioned in *slices*, see Note 9), (ii) one or more identifiers, (iii) one or more intervals. With *PQ-encoding* [Gil et Zibin, 2005], each class is associated with one identifier in each slice and a single interval for encoding its subtypes. It gives one of the most compact constant-time techniques available for multiple inheritance. In contrast, [Alavi *et al.*, 2008] propose a similar though inverted combination for encoding the supertypes. Each class is now associated with a single identifier and one interval in each slice. However, with a single interval in each slice, the technique is not incremental and the authors avoid global recomputations by allowing several intervals per slice. This closely resembles *range compression*, apart from the encoding direction. Hence, the test is no longer time-constant.

On the other hand, a generic approach to subtype testing involves bit-vector encoding of hierarchies [Caseau, 1993 ; Habib et Nourine, 1994 ; Habib *et al.*, 1995 ; Habib *et al.*, 1997 ; Krall *et al.*, 1997]. The technique involves coloring a conflict graph whose definition is slightly different from both our conflict and coexistence graphs. The subtype testing problem can also be generalized to the computation of lower and upper bounds in a lattice, e.g. for computing the most general common subclass of two classes [Aït-Kaci *et al.*, 1989]—this is however far beyond the scope of this paper. Anyway, all of these approaches to subtype testing do not efficiently generalize to method invocation or attribute access.

Perfect hashing Hashtables provide a general alternative to implementations based on the position invariant. They are usually not strictly time-constant, but only on average. However, there is a constant-time, collision-free variant called *perfect hashing* [Sprugnoli, 1977 ; Mehlhorn et Tsakalidis, 1990 ; Czech *et al.*, 1997]. Perfect hashing applies only to static hashtables, without addition or deletion—this is the case for class hierarchies as long as one does not consider dynamic unloading and reloading, or incremental definitions of classes. [Ducournau, 2008] proposes to use perfect hashing for subtype tests and method invocation—the latter in the special case of interface-typed receiver in JAVA. In all generality, perfect hashing applies to both subtype tests and method invocation. Hence, it would also apply to attributes, through *accessor simulation.* Perfect hashing is a true generalization of class coloring, where the coloring function χ is replaced by a family of functions h_C which depend on class C. A key difference is that h_C is an explicit function—e.g. $h_C(x) = hash(x, H_C)$, where *hash* is some hash function and H_C is a parameter depending on C—whereas χ has a purely extensional definition.

Overall, perfect hashing is a complete implementation technique. It has a major advantage—it is incremental and compatible with dynamic loading. It has, however, a major drawback. While being time-constant and space-linear, time constant is about 2-fold that of coloring for subtype test and method invocation, and substantially more for access to attributes. Therefore, besides subtype testing, it may only be suited to JAVA-like languages, where it can be reserved to interface-typed operations, as they remain a small part of all object-oriented operations and do not concern attributes.

7 Conclusion and Perspectives

Coloring is a versatile implementation technique whose different variants have been discovered and rediscovered by several people [Dixon *et al.*, 1989 ; Pugh et Weddell, 1990 ; Ducournau, 1991 ; Vitek *et al.*, 1997 ; Zibin et Gil, 2003]. The main goal of this paper was to generalize these different works and highlight their unity—i.e. what we have called *coloring.* A first by-product of this synthesis is the obvious possibility of applying an optimization provided in some variant to the technique proposed in another one—e.g. bi-directionality [Pugh et Weddell, 1990] improves *selector coloring* [Dixon *et al.*, 1989] and *pack encoding* [Vitek *et al.*, 1997]. Another contribution of this article is an analysis of theoretical issues—though the work was mostly undertaken by [Pugh et Weddell, 1990 ; Pugh et Weddell, 1993]. As expected, the coloring problem is NP-hard, except in a few cases where coloring is regular and there is perfect coloring. A more decisive contribution is a proposition of efficient heuristics and their systematic experimentation, on a wide set of large-scale benchmarks. This was the main drawback of all early studies—apart from [Vitek *et al.*, 1997]. Overall, coloring appears to be an efficient implementation technique, likely one of the most efficient in multiple inheritance. Contrary to C++ subobject-based implementation, which is detrimental to efficiency even when and where one does not use multiple inheritance, coloring provides exactly the same implementation as with single inheritance for single inheritance hierarchies. Moreover, with multiple inheritance, its overhead w.r.t. single inheritance is low and concerns only the memory space occupied by objects and classes. Coloring requires, however, a global computation which can be postponed at link-time.

Coloring in a real object-oriented language A main perspective of this work is to use coloring for implementing real object-oriented languages. This is complicated, however, since implementation of basic features is usually hard-wired in most compilers. Moreover, there are not many object-oriented languages, especially with both static typing and multiple inheritance—C++, EIFFEL—and they present inheritance-related features that would not easily fit with coloring [Ducournau et Privat, 2008]. In C++, the specification of inheritance, either virtual or non-virtual, seems to impose a subobject-based implementation [Lippman, 1996]. In EIFFEL, an unrestricted use of renaming makes the semantics unclear. In both cases, besides the difficulty of modifying the compiler, applying the modified compiler to existing programs would be a mess. So the solution has been to specify and implement a new object-oriented language, i.e. PRM. We are currently achieving this [Privat et Ducournau, 2005 ; Privat, 2006]. The bootstrap of the PRM compiler provides a test-bed dedicated to the run-time assessment of different implementation techniques, including coloring, binary tree dispatch, perfect hashing, etc. The specifications of the PRM language include a powerful notion of module and class refinement [Ducournau *et al.*, 2007] which is the basis of the PRM compiler modular architecture and makes it easy to replace, e.g. the coloring module, by the BTD module, and test it. Systematic tests are currently undertaken.

Truly incremental implementation The major drawback of coloring is its non-incrementality, hence its relative incompatibility with dynamic loading. Therefore, the main issue concerning coloring is to find an efficient incremental variant. We have briefly proposed a solution, inspired by [Palacz et Vitek, 2003], available only in a static typing setting (Figure 7). This solution works but would be likely inefficient at run-time, especially for attributes. So this can only be envisaged for JAVA-like languages. Nevertheless, when class loading introduces conflicts between previously non-conflicting superclasses, the load-time recomputation cost must be precisely evaluated. Alternatives to this scheme must also be examined. For instance, method tables themselves could also be recomputed but could this save on one indirection? Besides the algorithm itself, its run-time cost and its total space-cost, the issue would be method table management. How could the method tables of current instances be modified without adding an extra indirection?

Among the various alternatives to coloring that we have briefly surveyed (Section 6), three techniques can be understood as possible propositions for an incremental implementation: (i) the mixed technique proposed by [Vitek et Horspool, 1994] could be used at load-time for solving color conflicts between already loaded superclasses of the currently loaded class; (ii) perfect hashing is an exact incremental generalization of coloring; (iii) class-based row displacement represents a last alternative. All three techniques have numerous drawbacks. The former could penalize all usages of multiple inheritance, as long as classes are loaded one at a time. Therefore, the technique would gain from a notion of 'module'—long advocated in the object-oriented community [Szyperski, 1992 ; Ducournau *et al.*, 2007]—allowing to load and color a set of classes as a whole. Moreover, an extra parameter is required for all method invocations and subtype checks. Access to attributes, though possible, would be rather inefficient, as compared to link-time coloring. On the other hand, perfect hashing is truly incremental, without penalization for any specific situation apart from uniform overhead. It is not too inefficient for method invocation and subtype test, but access to attributes would really not be better than with the aforementioned incremental version of coloring. So, like incremental coloring, this can likely be only envisaged for multiple subtyping languages.

Class-based row displacement may be promising but this requires further research.

Currently, there is no convincing incremental version of coloring, and the search for such a variant or for an alternative is the main perspective of this work. Anyway, Prm and its modular compiler will represent a powerful testbed for assessing the feasability and the efficiency of these various hypothetic techniques.

References

[Agrawal *et al.*, 1989] R. Agrawal, A. Borgida, et H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. SIGMOD'89*, ACM SIGMOD Record, 18(2), pages 253–262, 1989.

[Aït-Kaci *et al.*, 1989] H. Aït-Kaci, R. Boyer, P. Lincoln, et R. Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, 1989.

[Alavi *et al.*, 2008] H. S. Alavi, S. Gilbert, et R. Guerraoui. Extensible encoding of type hierarchies. In *Proc. POPL'08*, pages 349–358. ACM, 2008.

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

[Amiel *et al.*, 1994] E. Amiel, O. Gruber, et E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In *Proc. OOPSLA'94*, SIGPLAN Notices, 29(10), pages 244–258. ACM Press, 1994.

[André et Royer, 1992] P. André et J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*, SIGPLAN Notices, 27(10), pages 110–126. ACM Press, 1992.

[Bellare *et al.*, 1998] M. Bellare, O. Goldreich, et M. Sudan. Free bits, PCPs and non-approximability—towards tight results. *SIAM J. Comp.*, 27:804–915, 1998.

[Caseau, 1993] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. OOPSLA'93*, SIGPLAN Notices, 28(10), pages 271–287. ACM Press, 1993.

[Cheung et Grogono, 1992] B. Cheung et P. Grogono. Compact record layouts for multiple inheritance. Rapport Technique OOP-92-01, Department of Computer Science, Concordia University, 1992.

[Click et Rose, 2002] C. Click et J. Rose. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE Conf. on Java Grande (JGI'02)*, pages 96–107, 2002.

[Cohen, 1991] N. H. Cohen. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, 1991.

[Collin *et al.*, 1997] S. Collin, D. Colnet, et O. Zendra. Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conference*, LNCS 1204, pages 67–81. Springer, 1997.

[Conroy et Pelegri-Llopart, 1983] T. J. Conroy et E. Pelegri-Llopart. An assessment of method-lookup caches for Smalltalk-80. In *Smalltalk-80 Bits of History, Words of Advice*, éditeur Krasner, pages 238–247. 1983.

[Cormen *et al.*, 2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, et C. Stein. *Introduction to Algorithms*. MIT Press, second édition, 2001.

[Czech *et al.*, 1997] Z. J. Czech, G. Havas, et B. S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2):1–143, 1997.

[Desnos, 2004] N. Desnos. Un garbage collector pour la coloration bi-directionnelle. Master's thesis, Université Montpellier 2, 2004.

[Deutsch et Schiffman, 1984] L. P. Deutsch et A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. ACM Symp. on Principles of Prog. Lang. (POPL'84)*, pages 297–302, 1984.

[Dijkstra, 1960] E. W. Dijkstra. Recursive programming. *Numer. Math.*, 2:312–318, 1960.

[Dixon *et al.*, 1989] R. Dixon, T. McKee, P. Schweitzer, et M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*, pages 211–214. ACM Press, 1989.

[Driesen et Hölzle, 1995] K. Driesen et U. Hölzle. Minimizing row displacement dispatch tables. In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 141–155. ACM Press, 1995.

[Driesen, 1993] K. Driesen. Selector table indexing and sparse arrays. In *Proc. OOPSLA'93*, SIGPLAN Notices, 28(10), pages 259–270. ACM Press, 1993.

[Driesen, 2001] K. Driesen. *Efficient Polymorphic Calls*. Kluwer Academic Publisher, 2001.

[Ducournau *et al.*, 2007] R. Ducournau, F. Morandat, et J. Privat. Modules and class refinement: a metamodeling approach to object-oriented languages. Rapport Technique LIRMM-07021, Université Montpellier 2, 2007.

[Ducournau et Privat, 2008] R. Ducournau et J. Privat. Metamodeling semantics of multiple inheritance. Rapport Technique LIRMM-08017, Université Montpellier 2, 2008. (to appear in *Science of Computer Progamming*).

[Ducournau, 1991] R. Ducournau. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France, 1991.

[Ducournau, 1997] R. Ducournau. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3):241–276, 1997.

[Ducournau, 2001] R. Ducournau. La coloration, une technique pour l'implémentation des langages à objets à typage statique. I. Coloration de classes. Rapport Technique LIRMM-01225, Université Montpellier 2, 2001.

[Ducournau, 2002a] R. Ducournau. Implementing statically typed object-oriented programming languages. Rapport Technique 02-174, LIRMM, Université Montpellier 2, 2002. (to appear in ACM *Comp. Surv.* 43(4), 2011).

[Ducournau, 2002b] R. Ducournau. La coloration pour l'implémentation des langages à objets à typage statique. In *Actes LMO'2002* in *L'Objet vol. 8*, éditeurs M. Dao et M. Huchard, pages 79–98. Lavoisier, 2002.

[Ducournau, 2003] R. Ducournau. La coloration, une technique pour l'implémentation des langages à objets à typage statique. II. Coloration de méthodes et d'attributs. Rapport Technique LIRMM-03036, Université Montpellier 2, 2003.

[Ducournau, 2008] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):1–56, 2008.

[Dujardin *et al.*, 1998] E. Dujardin, E. Amiel, et E. Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. Program. Lang. Syst.*, 20(1):116–165, 1998.

[Eckel et Gil, 2000] N. Eckel et J. Gil. Empirical study of object-layout and optimization techniques. In *Proc. ECOOP'2000*, éditeur E. Bertino, LNCS 1850, pages 394–421. Springer, 2000.

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

[Fall, 1995] A. Fall. Heterogeneous encoding. In *Proc. KRUSE'95*, éditeurs G. Ellis, R. A. Levinson, A. Fall, et V. Dalh, pages 162–167, 1995.

[Festa *et al.*, 2002] P. Festa, P. M. Pardalos, M. G. C. Resende, et C. C. Ribeiro. Randomized heuristics for the MAX-CUT problem. *Optimization Methods and Software*, 7:1033–1058, 2002.

[Gagnon et Hendren, 2001] E. M. Gagnon et L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proc. USENIX JVM'01*, pages 27–40, 2001.

[Garey et Johnson, 1979] M. R. Garey et D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco (CA), USA, 1979.

[Gil *et al.*, 2008] J. Gil, W. Pugh, G. E. Weddell, et Y. Zibin. Two-dimensional bi-directional object layout. *ACM Trans. Program. Lang. Syst.*, 30(5):28:1–38, 2008.

[Gil et Sweeney, 1999] J. Gil et P. Sweeney. Space and time-efficient memory layout for multiple inheritance. In *Proc. OOPSLA'99*, SIGPLAN Notices, 34(10), pages 256–275. ACM Press, 1999.

[Gil et Zibin, 2005] J. Gil et Y. Zibin. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst.*, 27(5):819–856, 2005.

[Gil et Zibin, 2007] J. Gil et Y. Zibin. Efficient dynamic dispatching with type slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):5:1–53, 2007.

[Godin *et al.*, 1998] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, et T. T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Practice of Object Systems*, 4(2):117–133, 1998.

[Goldberg et Robson, 1983] A. Goldberg et D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading (MA), USA, 1983.

[Habib *et al.*, 1995] M. Habib, M. Huchard, et L. Nourine. Embedding partially ordered sets into chain-products. In *Proc. KRUSE'95*, éditeurs G. Ellis, R. A. Levinson, A. Fall, et V. Dalh, pages 147–161, 1995.

[Habib *et al.*, 1997] M. Habib, L. Nourine, et O. Raynaud. A new lattice-based heuristic for taxonomy encoding. In *Proc. KRUSE'97*, pages 60–71, 1997.

[Habib et Nourine, 1994] M. Habib et L. Nourine. Bit-vector encoding for partially ordered sets. In *ORDAL*, éditeurs V. Bouchitté et M. Morvan, LNCS 831, pages 1–12. Springer, 1994.

[Harbinson, 1992] S. P. Harbinson. *Modula-3.* Prentice Hall, 1992.

[Holst et Szafron, 1997] W. Holst et D. Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In *Proc. ECOOP'97*, éditeurs M. Aksit et S. Matsuoka, LNCS 1241, pages 271–301. Springer, 1997.

[Hölzle *et al.*, 1991] U. Hölzle, C. Chambers, et D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91*, éditeur P. America, LNCS 512, pages 21–38. Springer, 1991.

[Huang et Chen, 1992] S.-K. Huang et D.-J. Chen. Two-way coloring approaches for method dispatching in object-oriented programming systems. In *Proc. COMPSAC'92*, pages 39–44, 1992.

[Jensen et Toft, 1995] T. R. Jensen et B. Toft. *Graph Coloring Problems.* John Wiley, 1995.

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

[Krall *et al.*, 1997] A. Krall, J. Vitek, et R. N. Horspool. Near optimal hierarchical encoding of types. In *Proc. ECOOP'97*, éditeurs M. Aksit et S. Matsuoka, LNCS 1241, pages 128–145. Springer, 1997.

[Krall et Grafl, 1997] A. Krall et R. Grafl. CACAO - a 64 bits JavaVM just-in-time compiler. *Concurrency: Practice ans Experience*, 9(11):1017–1030, 1997.

[Lippman, 1996] S. B. Lippman. *Inside the C++ Object Model.* New York, 1996.

[Mehlhorn et Tsakalidis, 1990] K. Mehlhorn et A. Tsakalidis. Data structures. In *Algorithms and Complexity*, éditeur J. Van Leeuwen, volume 1 de *Handbook of Theoretical Computer Science*, chapitre 6, pages 301–341. Elsevier, Amsterdam, 1990.

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

[Meyer, 1997] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second édition, 1997.

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

[Muthukrishnan et Muller, 1996] S. Muthukrishnan et M. Muller. Time and space efficient method lookup for object-oriented languages. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 42–51. ACM/SIAM, 1996.

[Myers, 1995] A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 124–139. ACM Press, 1995.

[Palacz et Vitek, 2003] K. Palacz et J. Vitek. Java subtype tests in real-time. In *Proc. ECOOP'2003*, éditeur L. Cardelli, LNCS 2743, pages 378–404. Springer, 2003.

[Pfister et Templ, 1991] B. H. C. Pfister et J. Templ. Oberon technical notes. Rapport Technique 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik, 1991.

[Privat et Ducournau, 2005] J. Privat et R. Ducournau. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*, pages 20–27, 2005.

[Privat et Morandat, 2008] J. Privat et F. Morandat. Coloring for shared object-oriented libraries. In *Workshop ICOOOLPS at ECOOP'08*, 2008.

[Privat, 2006] J. Privat. PRM, the language. version 0.2. Rapport Technique 06-029, LIRMM, Université Montpellier 2, 2006.

[Pugh et Weddell, 1990] W. Pugh et G. Weddell. Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*, ACM SIGPLAN Notices, 25(6), pages 85–91, 1990.

[Pugh et Weddell, 1993] W. Pugh et G. E. Weddell. On object layout for multiple inheritance. Rapport Technique CS-93-22, University of Waterloo, 1993.

[Queinnec, 1998] Ch. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 64(6):315–321, 1998.

[Schubert *et al.*, 1983] L.K. Schubert, M.A. Papalaskaris, et J. Taugher. Determining type, part, color and time relationship. *Computer*, 16:53–60, 1983.

[Sprugnoli, 1977] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Comm. ACM*, 20(11):841–850, 1977.

[Steele, 1990] G. L. Steele. *Common Lisp, the Language.* Digital Press, second édition, 1990.

[Sweeney et Burke, 2003] P. F. Sweeney et M. G. Burke. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Softw., Pract. Exper.*, 33(7):595–636, 2003.

[Szyperski, 1992] C. Szyperski. Import is not inheritance. Why we need both: Modules and classes. In *Proc. ECOOP'92*, éditeur O. L. Madsen, LNCS 615, pages 19–32. Springer, 1992.

[Taft *et al.*, 2006] éditeurs S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, et P. Leroy. *Ada 2005 Reference Manual: Language and Standard Libraries.* LNCS 4348. Springer, 2006.

[Takhedmit, 2003] P. Takhedmit. Coloration de classes et de propriétés : étude algorithmique et heuristique. Master's thesis, Université Montpellier 2, 2003.

[Tarjan et Yao, 1979] R. E. Tarjan et A. C. C. Yao. Storing a sparse table. *Comm. ACM*, 22(11):606–611, 1979.

[Toft, 1995] B. Toft. Colouring, stable sets and perfect graphs. In *Handbook of Combinatorics*, éditeurs R. L. Graham, M. Grötschel, et L. Lovász, volume 1, chapitre 4, pages 233–288. Elsevier, MIT Press, 1995.

[Vitek *et al.*, 1997] J. Vitek, R. N. Horspool, et A. Krall. Efficient type inclusion tests. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997.

[Vitek et Horspool, 1994] J. Vitek et R. N. Horspool. Taming message passing: efficient method look-up for dynamically typed languages. In *Proc. ECOOP'94*, éditeurs M. Tokoro et R. Pareschi, LNCS 821, pages 432–449, 1994.

[Vitek et Horspool, 1996] J. Vitek et R. N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Proc. CC'96*, LNCS 1060, pages 309–325. Springer, April 1996.

[Wille, 1992] R. Wille. Concept lattices and conceptual knowledge systems. *Computers Math. Applic.*, 23(6-9):493–515, 1992.

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

[Zibin et Gil, 2003] Y. Zibin et J. Gil. Two-dimensional bi-directional object layout. In *Proc. ECOOP'2003*, éditeur L. Cardelli, LNCS 2743, pages 329–350. Springer, 2003.