Coloring, a Versatile Technique for Implementing Object-Oriented Languages

ROLAND DUCOURNAU
LIRMM – CNRS and Université Montpellier II, France

Object-oriented programming languages represent an original implementation issue due to the so-called *late binding* mechanism aka *message sending*. 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 statically typed languages, the receiver's dynamic type is only known to be a subtype of its static type. A similar issue arises with attributes, since their position in the object layout may depend on the object's dynamic type. Furthermore, subtyping introduces another original feature, namely dynamic—i.e. run-time—subtype checks.

All three mechanisms need specific implementations, data structures and algorithms. In statically typed languages, late binding is generally implemented with tables, called *virtual function tables* in C++ jargon. These tables reduce method calls to function calls, through a small fixed number 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, two major invariants—of reference and position—hold and make the implementation as efficient as possible. Otherwise, dynamic typing or multiple inheritance make it harder to retain these two invariants.

Coloring can be defined as the optimization technique which allows retention of these two invariants at minimal spatial cost. It has been introduced and applied, more or less independently, to methods—under the name of *selector coloring*—to attributes and classes—under the name of *pack encoding*.

This article aims at presenting coloring as the versatile generalization of these three techniques, which covers all specific needs of object-oriented implementation. The paper presents both theoretical analysis, heuristics for overcoming the high complexity of coloring and a general scheme for using coloring in a framework of separate compilation, despite its non-incrementality. Finally, experiments on large benchmarks prove the tractability of the approach.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; EIFFEL; SMALLTALK; CLOS; D.3.3 [**Programming languages**]: Language constructs and features—*classes and objects*; *inheritance*; D.3.4 [**Programming languages**]: Processors—*compilers*; *linkers*; *loaders*

General Terms: Algorithms, Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: coloring, compilers, downcast, late binding, linkers, message sending, multiple inheritance, object layout, object-oriented languages, pack encoding, selector coloring, single inheritance, subtype test, virtual function tables

1. INTRODUCTION

Object-oriented programming languages represent an original implementation issue due to the so-called *late binding* mechanism, 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

Author's address: R. Ducournau, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France

typed languages, the receiver's dynamic type is completely unknown and message sending is not safe from 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 type is bounded 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 are the basis for so-called *downcast* operators.

All three mechanisms 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 of reference and position hold. They 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—therefore its implementation is hampered by both significant overhead and ill-specified features [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2002a].

So, implementation is not a problem with single-subtyping languages. However, there is almost no such languages. There are a few examples such as Oberon [Mössenböck 1993], Modula-3 [Harbinson 1992], or Ada 95 [Barnes 1995], but they result from the evolution of non-object-oriented languages and object orientation is not their main characteristic. This is a strong argument for the importance of multiple inheritance. In static typing, commonly used pure object-oriented languages, such as C++ or Eiffel [Meyer 1992; 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. classes or *interfaces*, are in *multiple subtyping*. So there is a real need for efficient object implementation in the framework of multiple inheritance and static typing. The requirement for multiple inheritance is less urgent in the framework of dynamic typing—an explanation is that the canonical static type system corresponding to a language like Smalltalk [Goldberg and Robson 1983] is that of Java, i.e. multiple subtyping. Anyway, dynamic typing yields 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 which allows retention of these two invariants at minimal spatial cost. It has been introduced and applied, more or less independently, to methods—under the name of *selector coloring* [Dixon et al. 1989]—to attributes [Pugh and Weddell 1990; Ducournau 1991] and classes—under the name of *pack encoding* [Vitek et al. 1997].

This article aims at presenting coloring as the versatile generalization of these three techniques, which covers all specific needs of object-oriented implementation. In the framework of multiple inheritance hierarchies, it has the same efficiency as single inheritance implementations—hence it should be envisaged for implementing

languages. The paper presents a theoretical analysis, following the first results by [Pugh and Weddell 1990; 1993]. As it turns out that coloring is almost always NP-hard, tractable heuristics are presented, together with a general scheme for using coloring in a framework of separate compilation, despite 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. Moreover, coloring is used in an experimental language, Prm [Privat and Ducournau 2005; Privat 2006].

Structure of the paper. Section 2 presents the usual object implementation in the static typing and single inheritance framework. 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). Finally, a generalization to bidirectional and n-dimensional coloring is proposed. The practical use of all these variants is examined, together with the application of coloring in a separate compilation framework. In the following Section, a model of multiple inheritance is sketched and 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*, thus likely NP-hard [Garey and Johnson 1979; Toft 1995; Jensen and Toft 1995]. As expected, it is NP-hard but in few cases. Section 5 describes tractable heuristics and presents the results of experiments on large benchmarks. 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. SINGLE INHERITANCE AND STATIC TYPING

A survey of the most common implementation techniques for object-oriented languages is presented in [Ducournau 2002a]. This section reviews the main points.

2.1 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 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). 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

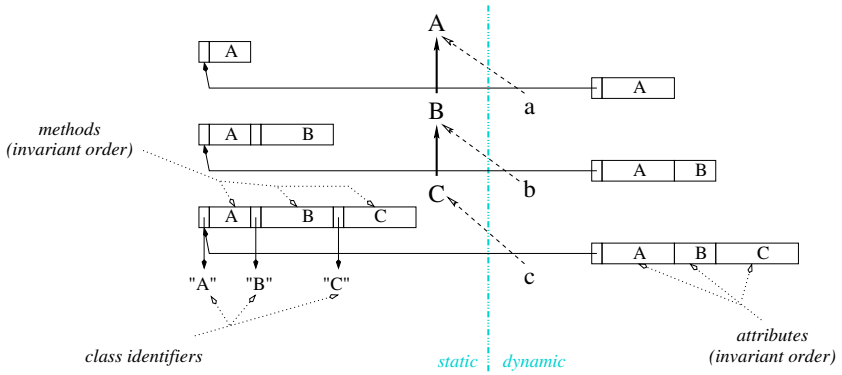

Fig. 1. Object layout (dynamic memory, right) and method tables (static memory, left) in single subtyping: 3 classes A, B and C with their respective instances, a, b and c.

straightforward. Note that 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 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.

2.2 Subtype tests

Though it has long been claimed that class specialization and subtyping are different notions [Cook et al. 1990], the distinction is not required here and we shall consider that a class is or has a type, and that class specialization entails subtyping between the corresponding types. In the following, the class specialization, hence subtyping, 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 framework, 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]. It has been widely reused by different authors (e.g. [Queinnec 1998; Alpern et al. 001b]).

It consists of assigning an offset to each class in the method tables of its subclasses—the corresponding table entry must contain the class identifier. Given an object which is a *direct* instance of a class D[1], called its dynamic type, this object

[1]By *direct instance*, we mean that the object has been produced by instantiating this class, e.g. by a `new` D statement. On the contrary, an *indirect instance* is a direct instance of the class, or of one of its subclasses. In a static typing framework, the static type of the reference to the object, e.g. C, is a supertype of the instantiation class: $D \preceq C$.

```
load [object + #attColor], attVal

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

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

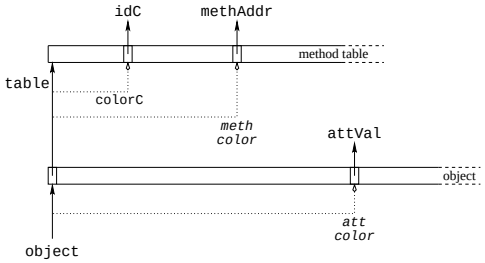

Fig. 2. Single inheritance implementation—code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. Pointers and pointed values are in roman characters, with solid lines, and offsets are italicized, with dotted lines.

is an *indirect* instance of a class C, the *target type*, iff the object's method table, denoted tab_D, contains, at offset $\chi(C)$ (called the *color* of C), the identifier id_C:

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

Originally, Cohen's test was implemented in specific tables and $\chi(C)$ was the depth of C in the hierarchy tree. However, in a statically typed object-oriented framework, it is more efficient to merge these specific tables within the method tables (Figure 1). Class offsets are ruled by the same position invariant as methods and attributes. Actually, this resembles a situation whereby each class C introduces a method for checking whether an object is an instance of C, i.e. such that its instances can check they actually are. The test fails when this pseudo-method is not found, i.e. when something else is found at its expected position.

Two points must be carefully examined. First, inlining the Cohen display in the method table requires that a class identifier cannot be confused with a method address. As addresses are even numbers (due to word alignment), coding class identifiers with odd numbers avoid any confusion between the two types. Secondly, in theory, $tab_D[\chi(C)]$ is sound only if $\chi(C)$ does not run out of the bounds of 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, 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 identifiers—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[2]. If method tables contain more data than addresses and identifiers, 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 identifiers are gathered within specific tables, distinct from method tables and allocated in

[2]In the dynamic loading framework, this maximum size is not known—an upper bound must be used, which is a parameter of the runtime platform. Moreover, there could be more than one method table area—when the current one is full, a new one may be allocated, and the maximum size may also be adjusted, without exceeding the padded size in the previous areas.

Table I. Instruction set of the abstract assembly language [Driesen 2001, p. 193]

`R1`	a register (any argument without `#`)
`#immediate`	an immediate value (prefix `#`)
`load [R1+#imm], R2`	load the word in memory location `R1+#imm` to register `R2`
`add R1, R2, R3`	add register `R1` to `R2`. Result is put in `R3`
`call R1`	jump to address in `R1` (can also be immediate), save return address
`comp R1, R2`	compare value in register `R1` with `R2` (`R2` can be immediate)
`bne #imm`	if last compare is not equal, jump to `#imm` (`beq`, `blt`, `bgt` are analogues)

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. In this way, Cohen's test preserves linear-space tables.

A more common and frequently proposed way to save on this extra bound check is to use fixed-size tab_D. Click and Rose [2002] attribute the technique to [Pfister and Templ 1991]. However, statistics on a set of benchmarks commonly used in the object-oriented language implementation community (see Table V) show that, on these benchmarks, the maximum superclass number may be 5-fold greater than its average. Hence, fixed size entails a large space overhead. Such an arbitrary upper bound does not entail the same overhead and the same limitation, according to whether it is used for each class, as with fixed size tables, or only for one or a few specific areas, as in note 2.

Synthesis. Figure 2 presents the diagram of object layout and method table in this setting, together with the corresponding code sequence in an intuitive pseudo-code. This pseudo-code is borrowed from [Driesen 2001] and its instruction set is described in Table I. 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.

2.3 Multiple inheritance or dynamic typing

2.3.1 *Multiple inheritance.* 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 extends 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 in a static typing and separate compilation framework—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 the attributes introduced by the corresponding class, together with a pointer to a method table which contains the methods known—i.e. defined or inherited—by the class. A reference of a given static type points at the subobject corresponding to this type. This is the C++ implementation, when the keyword `virtual` annotates each superclass [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2002a]. It is time-constant and compatible with dynamic loading, but method tables are no longer space-linear. The number of method tables is

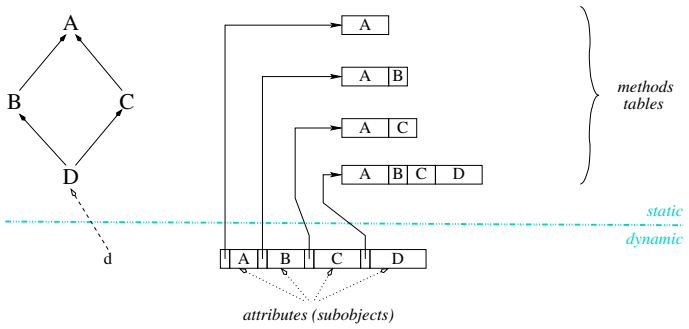

Fig. 3. Object layout and method tables with multiple inheritance: an instance d of the class D is depicted

exactly the size of the specialization relationship, thus quadratic in the number of classes. The total table size of a class is itself quadratic in the number of superclasses. So, in the worst case, 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[3]. These pointer adjustments 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. Sweeney and 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 here ALL implementation, which is closest to most actual implementations, but our conclusions hold for the whole implementation family.

Figure 4 displays the code sequence for basic SMI mechanisms, together with the corresponding diagram. At method invocation, `self`-adjustment 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 Figure displays three different pointers at the same object: `object` is the original reference to the object in the caller, with a certain static time C, `self` is the reference to the subobject of some type D, unrelated with C, where the callee method has been defined, 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 effective overhead for method invocation. 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

[3]These pointer adjustments are safe—i.e. the target type is always a supertype of the source type—and are implemented more efficiently than subtyping tests.

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

load [object + #attOffset], attVal

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

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

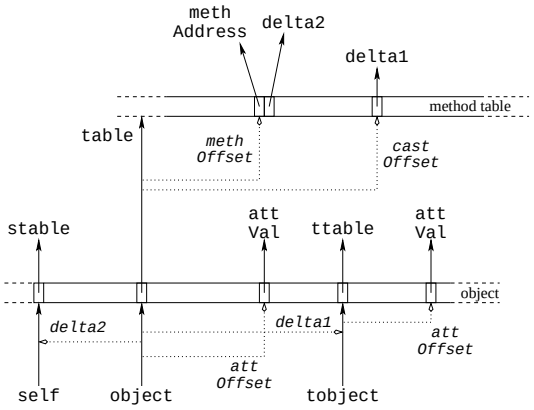

Fig. 4. Standard multiple inheritance implementation—code sequences for all the basic mechanisms, but subtype testing, and the corresponding diagrams of object layout and method table. From top to bottom, pointer adjustment, access to an attribute introduced by the current static type, access to other attributes and method invocation.

`virtual` keyword. It 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.

2.3.2 *Dynamic typing.* Replacing multiple inheritance by dynamic typing leads to similar problems regarding method invocation and access to attributes—however, subtype test does not depend on static or dynamic typing. The same property `foo`—i.e. properties with the same name `foo`—can be defined 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 and Robson 1983], finds an efficient solution in a strict *encapsulation* for attributes—the language reserves to `self`[4] all accesses to attributes. As it turns out that each occurrence of `self` is statically typed by the class defining the method which includes the considered occurrence, it follows that, in SMALLTALK, reference invariant holds together with position invariant, but for attributes only. Regarding method invocation, it mainly involves a combination of non-constant time techniques, e.g. hashing and caching [Conroy and Pelegri-Llopart 1983; Deutsch and Schiffman 1984; Hölzle et al. 1991], with the alternative being global techniques like coloring or row displacement [Driesen and Hölzle 1995].

3. PRINCIPLE AND HISTORY OF COLORING

The general idea underlying coloring is to keep the two invariants of single inheritance—i.e. reference and position. Obviously there is a solution, e.g. an injective numbering of attributes, methods and classes verifies the invariant. So this is

[4] `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 seems closer to the original metaphor. `Self` can be considered as a reserved formal parameter of the method, and its static type—even in dynamically typed languages!—is the class within which the method is defined.

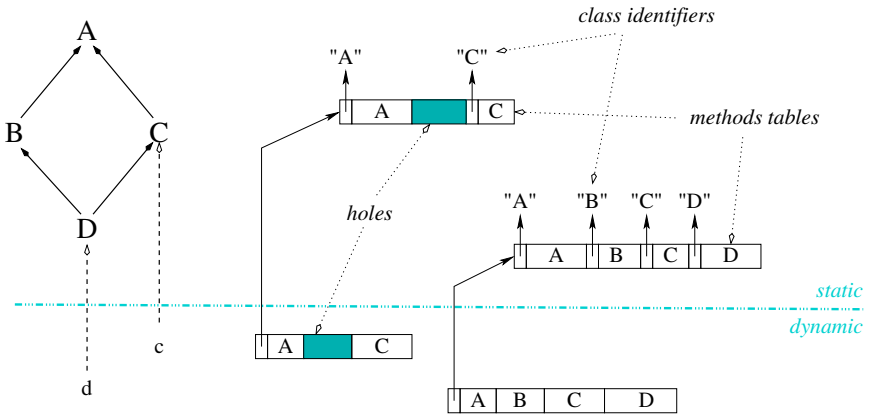

Fig. 5. Unidirectional coloring applied to classes, methods and attributes. A and B classes are presumed to have the same implementation as in Figure 2 and there are holes in the C's tables since some space must be reserved for B in the tables of D.

clearly a matter of optimization, to minimize the total size of all tables. However, this cannot be done separately for each class, but it requires complete knowledge of the whole hierarchy. Of course, the technique takes its name from graph coloring [Toft 1995; Jensen and Toft 1995].

3.1 Short history

Coloring, as a general implementation technique, has been independently discovered in the specific case of each basic mechanism.

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*[5] *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 graph, that here we call *selector coexistence graph*, whose vertices are selectors, whereby there is an edge between two selectors when they are known—i.e. defined or inherited—by some class. Regarding the optimization criterion, the authors minimize the color number. From the complexity standpoint, as an instance of the classic *minimum graph coloring problem*, this is a NP-hard problem [Garey and Johnson 1979; Toft 1995; Jensen and 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 and 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 in-

[5]Selector is the term used in SMALLTALK for method names, which here we call *generic property*, i.e. not the code itself, but the part of the property which is invariant by inheritance and specialization—name, signature, etc. (see Section 4.1).

tractable technique[6]. After André and Royer, selector coloring has been integrated in a general framework for method dispatch by Holst and Szafron [1997].

3.1.2 *Attribute coloring and minimization of table size.* At about the same time as selector coloring, Pugh and Weddell [1990] propose 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. Obviously, the point is no longer to minimize the number of colors, i.e. the matrix size, but the total size of all tables associated to the different classes. Equivalently, this amounts to minimizing the number of *holes*, i.e. empty entries of tables. A coloring without hole is said *perfect*.

From an historical standpoint, Pugh and 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 existence of bidirectional perfect coloring, or in order to prove the NP-hardness of bidirectional coloring (Section 4.5). However, in a technical report, Cheung and Grogono [1992] make a similar but rather underdeveloped proposition, which is applied in the Dee system. Both [Dixon et al. 1989] and [Pugh and 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 survey paper of all techniques for message sending in dynamically typed languages [Ducournau 1997].

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 and Weddell [1993] and rediscovered by Zibin and Gil [2003]. Besides coloring, bidirectionality of method tables or object layout has been widely reused [Myers 1995; Krall and Grafl 1997; Gil and Sweeney 1999; Gagnon and Hendren 2001]. Bidirectional coloring must not be confused with *two-way coloring* [Huang and Chen 1992], which is a generalization of selector coloring that merges both rows and columns of the dispatch matrix.

Whereas [Pugh and Weddell 1990; 1993] have been a deciding work on coloring—all considerations were already there apart from the application to method invocation and subtype testing—these papers have been curiously unrecognized. In the 90s, it seems that the first paper is only cited by [Myers 1995; Gil and Sweeney 1999][7].

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 Co-

[6]This criticism is not directed at the authors, but at computer scientists, collectively—a negative experiment is not a counter-example of the tested approach, but only of the applied protocol.
[7]In 1999, [Pugh and Weddell 1990] appears in the reference list of Driesen's PhD thesis [2001], but we did not find in the text any comment on this reference.

Table II. Coloring bibliography

paper		optimization criterion #colors	unidir.	bidir.	colored entities method	attribute	class	typing static	dynamic
Dixon et al.	[1989]	×			×				
Pugh and Weddell	[1990]		×	×		×			×
Ducournau	[1991]		×			×			×
André and Royer	[1992]	×			×				×
Ducournau	[1997]		×		×				×
Vitek et al	[1997]	×					×	×	×
Ducournau	[2002b]		×	×	×	×	×	×	
Zibin and Gil	[2003]			×		×		×	
Palacz and Vitek	[2003]	×					×	×	×

hen'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. As the authors use fixed-size tables, the technique amounts to minimizing the color number. However, the authors 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] or [Pugh and Weddell 1990].

Finally, though coloring is inherently non-incremental, Palacz and 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 II):

—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 subtyping test together with to provide 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 might or even should be weighted by the number of instances;

—typing may be static or dynamic;

—coloring may be unidirectional, bidirectional, or even n-dimensional.

—and, finally, coloring is preferably used in a global setting rather than in a dynamic loading framework, but incremental versions have also been proposed for Smalltalk [André and Royer 1992] or Java [Palacz and Vitek 2003].

Obviously, 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 [Garey and Johnson 1979]. 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 in practice

3.3.1 *Application to static typing.* In a static typing framework, uni- or bidirectional coloring is a direct extension of single inheritance implementation, and

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

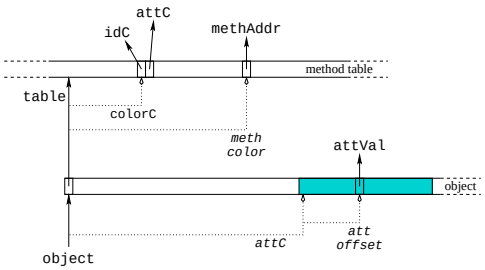

Fig. 6. Coloring with accessor simulation—sequence code for access to attributes and diagram of object layout and method table. The grey part is the group of attributes introduced by class C.

all three mechanisms use the exact same code and the same implementation as that used for single inheritance (Figure 2). The only difference is that colors must be globally computed, whereas single inheritance allows an incremental computation. This point will be examined Section 3.4.

However, another point must be discussed. The pros of coloring are that it allows the same code and the same time efficiency as with single inheritance. However, you get what you pay for—the cons of coloring are that it generates tables with *holes*, i.e. empty entries. This is not too highly payed for method and class coloring since the tables are *static*—i.e. they are shared by all instances—and the alternative, i.e. subobject-based implementation, is quite space-consuming.

3.3.2 *Attribute coloring vs. accessor simulation.* On the contrary, attribute coloring deserves some examination. A few holes in the object layout of a class with very few attributes and many instances might entail a considerable relative overhead. Therefore, the optimization criterion should account for the number of instances of each class, which would require profiling of the programs. Overall, an alternative to attribute coloring might be preferred—namely *accessors*. This approach, called *field dispatching* by Zibin and Gil [2003], involves encapsulating all accesses to attributes in special accessor methods, which are redefined in a class when the attribute position differs from its position in superclasses. Of course, true accessors would report attribute implementation upon method coloring, but at the expense of a method call for each access—this would be quite inefficient. A midway solution is to *simulate accessors*. A first improvement consists of replacing, in the method table, the accessor method address by the offset of the attribute in the object layout. Finally, the attributes introduced by a class can be grouped together in the object layout. Then one can substitute to 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.

The *double compilation* scheme proposed by Myers [1995] provides a final optimization, when accesses 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.

Note that accessor simulation is a generic approach for access to attributes, which works with any method invocation technique, at least in static typing or when attributes are always encapsulated. In dynamic typing, non-encapsulated attributes require separate storage of the offset of each attribute, since attributes are no longer 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 that is to add an extra parameter to each method—i.e. the selector itself—and to check it in the method prologue[8]. Furthermore, empty entries are filled with the address of a function which signals an exception—`doesntunderstand` 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—i.e. encapsulation which makes attributes reserved to `self`. Accesses to attributes of other entities are mediated by *accessors*, which may be generated by the compiler or defined by the programmer—the latter only in SMALLTALK. Strict encapsulation makes accesses to attributes statically typed, since an occurrence of `self` has always the static type of the class which contains this occurrence. Therefore, attribute coloring and accessor simulation apply as in static typing. Note that, despite this relative inadequacy, attribute coloring was only proposed in dynamic typing frameworks—either FLAVORS [Pugh and Weddell 1990] or YAFOOL [Ducournau 1991]. Finally, as noted above, subtype check does not depend on static types, so class coloring can be applied without change in the dynamic typing framework.

3.3.4 *Application of n-dimensional coloring.* Let us first specify the terminology: n-directional coloring consists of coloring with n tables of positive offsets. Bidirectional coloring interprets both tables as a single 1-dimensional array, with negative and positive offsets. One generalizes with n-dimensional coloring, which involves n bidirectional arrays, i.e. $2n$-directional coloring. Conversely, n-directional coloring involves $\lceil n/2 \rceil$ dimensions[9].

Unidirectional. Unidirectional coloring directly applies to all three cases of attributes, classes and methods—more likely, of classes and methods jointly. However,

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

[9] This terminology is slightly different from that of [Zibin and 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 n) and the offset in the bidirectional table—hence, the actual color is the pair index-offset.

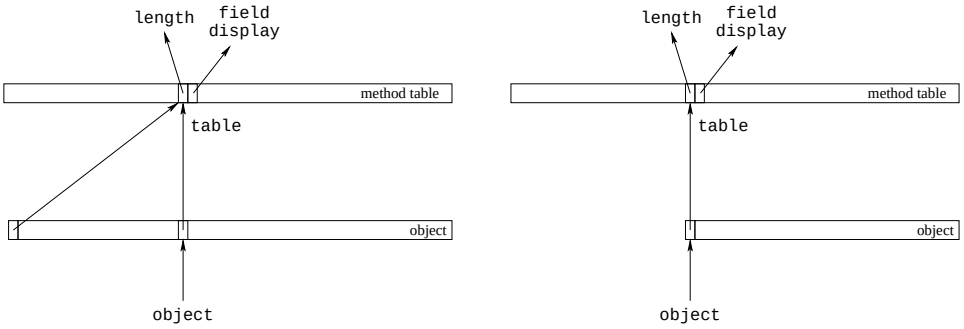

Fig. 7. Bidirectional object layout for garbage collection. When an object layout uses negative colors, an extra pointer to the method table is added. The length of the object-layout is included in the method table, together with the required information to decide which fields are pointers that must be followed during the garbage processing.

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 and Grafl 1997; Gil and Sweeney 1999; Gagnon and Hendren 2001]. However, bidirectional object layout might make garbage collection difficult, as it requires some information regarding memory allocation at the beginning—or at least at a fixed offset from 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 7). The memory gain w.r.t. unidirectional coloring will be reduced.

Gagnon and 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 obviously decrease when n increases[10]. 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). Of course, in a static typing framework, 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. This incurs no overhead in object

[10]This is a mathematical evidence for optimal algorithms but heuristics cannot ensure it.

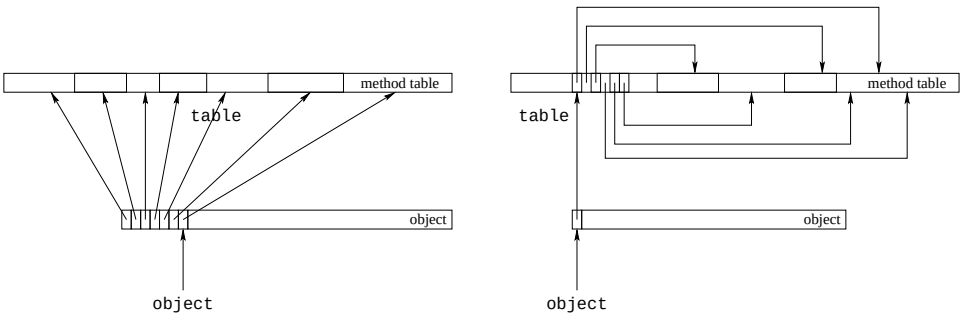

Fig. 8. n-dimensional method tables, with an n-pointer header in the object layout (left) or with indirections in the method table (right).

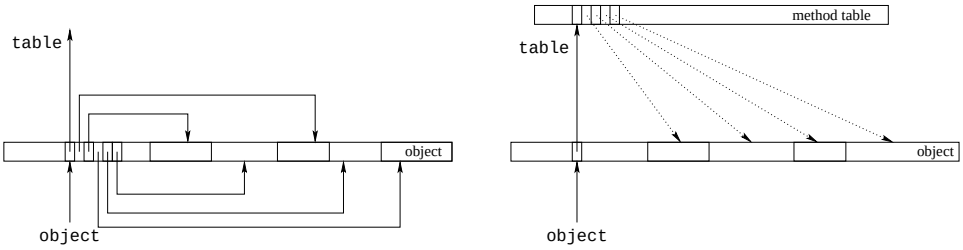

Fig. 9. n-dimensional object layout, with an n-pointer header in the object layout (left) or with indirections in the method table (right).

layout, but a constant overhead in method invocation for all methods which are not in the first dimension. In a global compilation framework, the overhead will likely concern only a small subset of method calls. However, in a separate compilation and global linking framework, it might concern all method calls.

Regarding class coloring, the same arguments apply—however, the left truncation would require an extra bound check. Now, this is not mandatory to improve 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, this is likely useless to consider n-dimensional coloring for classes and methods.

The real point is attribute coloring which presents the same alternative as method coloring (Figure 9). 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 impede the intention of saving on space—this would generate the same drawbacks as VBPTRs (Section 2.3.1). Therefore, the alternative proposed by [Zibin and Gil 2003] would 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. 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 attributes introduced in superclasses. 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.

3.4 From compilation to class loading

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, leaving the modularity brought by separate compilation may be considered too highly payed for program optimization. An alternative was already noted by Pugh and Weddell [1990]. Coloring does not require knowledge of the code itself, but only of the "model" (aka "schema") of the classes, all of which is already needed by separate compilation of object-oriented programs, namely the model of the superclasses of the currently compiled class, together with other classes which are used for typing the code of the current class. This class model is included in specific header files (in C++) or automatically extracted from source or compiled files (in Java). 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 the classes and color all the entities, before substituting values to the different symbols, as a linker commonly does.

Other powerful optimizations can be added to this compilation framework, e.g. type analysis and dead code elimination [Privat and Ducournau 2005].

3.4.2 *Load-time coloring.* Finally, a definitive defect will remain—coloring is not incremental, hence apparently not suitable for dynamic loading. However, as aforementioned, some authors attempted to use coloring at load-time. Actually, any technique can be used in a dynamic loading framework, at the expense of dynamic data structures, hence extra indirections, and of a possible complete recomputation. When the considered technique is inherently non-incremental, using it in a dynamic framework might make it lose all its desired qualities. So, the point with using coloring at load-time is to examine the overhead entailed at load-time—what is the recomputation cost?—and at run-time—extra indirections. From the load-time standpoint, specific incremental heuristics must be designed since coloring is NP-hard and near-optimal[11] global heuristics are roughly cubic (see Section 5). These incremental heuristics will likely favour time to the detriment of 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

[11] 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].

```
load [object + #tableOffset], table
load [table + #ctableOffset], ctable
load [classColor], colorC
add ctable, colorC, entry
load [entry + #1], attC
add object, attC, attgr
load [attgr + #attOffset], attVal

// lines 1-4
load [entry + #2], methC
add table, methC, methgr
load [methgr + #methOffset], methAddr
call methAddr

// lines 1-4
load [entry], idC
comp idC, #targetId
bne #fail
// succeed
```

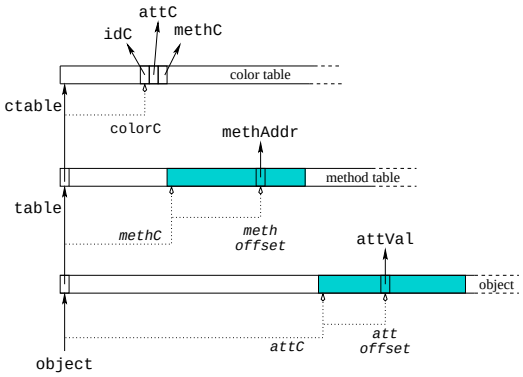

Fig. 10. Load-time coloring—object layout and method table

the method table of existing instances would have to be updated—this is easy to do, but at the expense of an extra indirection. Moreover, 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 one accepts such an expensive feature as the CLOS `change-class` generic function [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 framework, as follows (Figure 10):

—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 `idC`;

—methods are implemented in an analogous way, i.e. all the 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 `idC`.

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 to make object-layout and method tables invariant and to reduce 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 `load`s, 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

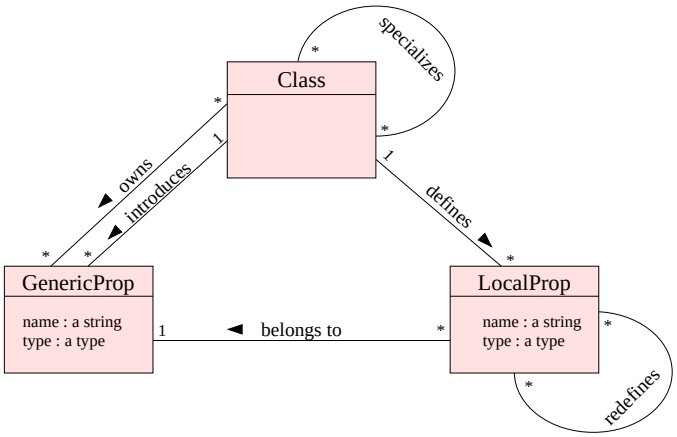

Fig. 11. Class and property meta-model

proposition of Palacz and Vitek [2003]—but also to method invocation when the receiver is typed by an interface.

Dynamic typing would make things even more difficult since, in this framework, attributes and methods are no longer introduced by a single class, so they cannot be grouped according to their introduction class, and their offset is meaningless. So, dynamic typing requires load-time method coloring but this would be quite expensive.

4. FORMAL DEFINITIONS AND THEORETICAL ISSUES

4.1 Semantics of multiple inheritance

Coloring is mainly a matter for multiple inheritance. Therefore, in order to make things clear, we first propose an analysis of multiple inheritance based on a simple meta-model of classes and properties, i.e. attributes and methods (Figure 11). This meta-model is based on the distinction of two kinds of properties and it ensures that each occurrence of a property identifier denotes a single instance of the meta-model. *Local properties* are properties as defined in the code of a class, and *generic properties*[12] are abstractions of properties through specialization and inheritance. These two kinds of properties are specialized in attributes and methods, but we do not need to develop this here. A class definition is made of a triplet: its name, the name of its superclasses, and a set of definition of local properties. The class inherits all *generic properties* of its superclasses and it can *redefine* (aka *override*) some of them by *defining* the corresponding *local properties*. Furthermore, the class introduces some new generic properties, by defining local properties which do not correspond to any inherited generic property. The correspondence between generic and local properties is made by their identifier—i.e. their name, possibly associated with parameter types, in the context of *static overloading*, as in C++ or Java. A

[12]The term *generic property* is coined on the model of Clos *generic functions* [Steele 1990] and Clos *methods* are the corresponding local properties.

```
class A              [1]
def foo              [2]

class B inherits A   [3]
def foo              [4]

x:A
x.foo()              [5]
```

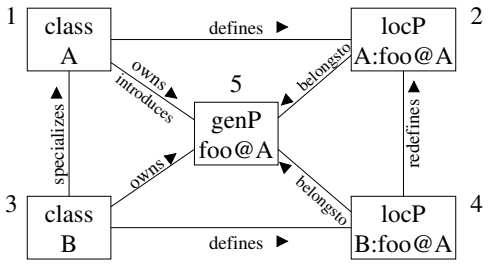

Fig. 12. Code sample and corresponding instance diagram

generic property of name `foo` contains all local properties of name `foo` defined in the subclasses of the class introducing the generic property.

Figure 12 presents a small piece of code with 2 class definitions and a call site, together with the corresponding instance diagram. In a static typing framework, in an access `x.foo()` to a property named `foo`, `foo` denotes the generic property of this name owned by the static type of `x`. The role of late binding is to select, in this generic property, the local property corresponding to the dynamic type of the value bound to `x`. Of course, in a dynamic typing framework, there is no static type to distinguish different generic properties with the same name—this is why, in SMALLTALK, generic properties would be identified to *selectors*, i.e. method names.

Regarding implementation, each class must assign a position, i.e. an offset in a table, to each of its generic properties, method or attribute. In the case of methods, the corresponding entry in the method table contains the address of the corresponding local property. In a position-invariant implementation, the position is a characteristic of the generic property and does not depend on the local properties.

This is the intuitive meta-model of object-oriented programming and the underlying meta-model of all languages, as long as there is no ambiguity—i.e. in a call site, a property identifier must always unambiguously denote a single generic property. Note here a key analogy between property identifiers and colors—both must be unambiguous in the context of a given class. In other words, coloring is a system for unambiguously denoting properties. A first cause of ambiguity is *static overloading*—i.e. methods with the same name but different parameter types in the context of a single class. The meta-model deals with static overloading by integrating parameter types in the property identifier—hence, in languages which allow static overloading (e.g. C++, JAVA, C#), different parameter types correspond to different generic properties. The second cause of ambiguity is *multiple inheritance* and this meta-model is the basis of a quite simple and natural analysis of these ambiguities. When defining a class with several direct superclasses—e.g. D in Figure 5—two problems may occur. First, D may inherit from its superclasses several generic properties with the same name, e.g. `foo`, but introduced in different classes, e.g. B and C—therefore the name `foo` is ambiguous in the context of D. This is a *generic property conflict* which must be solved by renaming, since this is a naming issue. This renaming can be local with respect to the class where the conflict occurs (D and all its subclasses), as in EIFFEL, or global to one of the two conflicting generic properties, i.e. in B or C, and all subclasses. Incidentally, this meta-model is implicit in EIFFEL, as long as one does not misuse renaming [Meyer

1997]. An alternative to renaming is a fully qualified syntax, e.g. `foo@A`, allowing the programmer to unambiguously denote the generic property `foo` introduced in A[13]. As a second problem, D may inherit, for a given non-ambiguous generic property `bar`, several *local properties* in such a way that none of them is more specific than the other—e.g. they are defined in B and C. This is a *local property conflict* which must be solved, e.g. by defining a *local property* in D, or with a linearization, like in CLOS or DYLAN [Ducournau et al. 1994; Barrett et al. 1996].

This analysis of multiple inheritance was proposed in [Ducournau et al. 1995] (in French) and it is very close to that of some other authors, possibly with different arguments, e.g. [Nystrom et al. 2006]. Now, coloring is somewhat concerned. Attribute and method coloring means generic property coloring—the point is to assign a single color to each generic property, it does not depend on local properties. Therefore, in the following, we shall no longer mention local properties and an unqualified use of 'property' will stand for 'generic property'. Moreover, a distinction must be made between static and dynamic typing. In the former, a generic property is introduced by one class—i.e. it is determined by its name, possibly its signature, and the introducing class. Pugh and Weddell [1990] call these properties *class-based*. In the latter case, a generic property may be introduced by several classes and is identified only by its name.

A more formal definition of this meta-model can be found in [Privat and Ducournau 2006].

4.2 Notations and definitions

Definition (HIERARCHY). A *hierarchy* (aka *inheritance graph*) is an acyclic directed 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 $\prec$. Moreover, $(X, \preceq)$ is a *partial order* (aka *poset*), since $\preceq$ is reflexive, transitive and antisymmetric.

P is the set of the considered generic properties, either attributes or methods. Finally, h is a subset of $X \times P$ which formalizes the relationship between classes and generic properties. $h(x, y)$ holds iff the class x has the property y. Alternatively, one defines the set of properties known by a class, with a function $p : X \to 2^P$, and $p(x) = \{y \in P \mid h(x, y)\}$. p and h satisfy the following property, which characterizes inheritance:

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

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

$$c(y) \stackrel{\text{def}}{=} \{x \in X \mid h(x, y) \;\&\; \forall x' \in X, x \prec x' \Rightarrow \neg h(x', y)\} \tag{3}$$

As aforementioned, in static typing, properties are *class-based*, i.e. $c(y)$ is always a singleton.

[13] Note that this essentially differs from `A::foo` in C++, which denotes a local property.

Finally, given two sets E and F, $E \uplus F$ asserts that the sets are disjoint and denotes their union. Moreover, given a function $f : E \to F$ and a subset $E' \subset E$, $f(E')$ denotes the set $\{f(x) \mid x \in E'\}$. 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 (COLORING). A *unidirectional* (resp. *bidirectional, k-directional*) *coloring* is a function $\chi : Y \to Z$, with $Z = \mathbf{N}$ (resp. $\mathbf{Z}$, $\mathbf{N} \times [1, k]$) such that $\forall x, y \in Y$,

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

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

Definition (COLOR TABLE). Given a coloring $\chi : Y \to Z$, the function $\chi^* : X \to 2^Z$ is defined as follows:

$$\chi^*(x) \stackrel{\text{def}}{=} \begin{cases} \{\chi(y) \mid x \preceq y\} & Y = X \\ \{\chi(y) \mid h(x, y)\} = \chi(p(x)) & Y = P \\ \text{union of both} & Y = X \uplus P \end{cases} \tag{5}$$

$\chi^*(x)$ is called the *color table* of x.

When $Z = \mathbf{N} \times [1, k]$, for all $i \in [1, k]$, $\chi_i^* : Y \to \mathbf{N}$ is defined as a projection: $\chi_i^*(x) = \{n \mid (n, i) \in \chi^*(x)\}$.

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

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

$$\chi^*(x) = \left(\bigcup_{x \prec_d y} \chi^*(y) \right) \biguplus \chi^+(x) = \biguplus_{x \preceq y} \chi^+(y) \tag{7}$$

An alternative characterization of coloring is the following:

PROPOSITION 4.2.1. *A function $\chi : Y \to Z$ is a coloring iff χ is injective on $\chi^{-1}(\chi^*(x))$, for all $x \in X$.*

The proof is trivial.

Definition (OPTIMAL COLORING). A coloring $\chi : Y \to Z$ is *optimal* if it minimizes

$$\begin{cases} \max_{y\in Y} \chi(y) & Z = \mathbf{N},\ \text{minimum color number} \\ \sum_{x\in X} \max(\chi^*(x)) & Z = \mathbf{N},\ \text{minimum size} \\ \sum_{x\in X} (\max(\chi^*(x)) - \min(\chi^*(x))) & Z = \mathbf{Z},\ \text{minimum size} \\ \sum_{i=1..k} \sum_{x\in X} \max(\chi_i^*(x)) & Z = \mathbf{N} \times [1,k],\ \text{minimum size} \end{cases} \tag{8}$$

In the bidirectional case, the additional constraint $\min(\chi^*(x)) \leq 0$ is required to keep the isomorphism with 2-directional coloring. Furthermore, the argument used in Section 2.2 for saving the bound check in Cohen's test can be applied to bidirectional coloring, under the only condition that color 0 is inside each table. In practice, this is likely for all rooted hierarchies, as 0 is the natural root color. Of course, this is always true for attribute coloring—color 0 is occupied by the pointer at method table, which may be considered as an extra attribute introduced in the hierarchy root—and for method coloring, as color 0 is used for information, e.g. memory allocation, common to all classes.

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

Definition (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

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

$h_\chi(x)$ is the number of holes in the color table of x, i.e. the cardinal 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 (OPTIMAL WEIGHTED COLORING). Given a *weight* function $w : X \to \mathbf{N}$, a coloring $\chi : Y \to Z$ is *optimal* if it minimizes

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

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

4.3 Structure of the hierarchy

4.3.1 *Regular coloring.* First note that attribute and method coloring is largely reduced 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 operational only when properties are class-based or in a static typing framework, since with dynamic typing a property may be introduced by several incomparable classes. Intuitively, a class-based property can be colored in the same condition as the 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 (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.

4.3.2 *Core and crown.* A second remark is that 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 the subclasses which are only specialized in single inheritance. This leads to distinguish the parts of the hierarchy which are in single or multiple inheritance.

Definition (CORE). The *core* of the inheritance graph is the subset X_{co} of classes in multiple inheritance:

$$X_{co} \stackrel{\text{def}}{=} \{x \in X \mid \exists y, z_1, z_2 \in X : y \preceq x, y \prec_d z_1, y \prec_d z_2, z_1 \neq z_2\} \qquad (11)$$

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

In the poset terminology, the core is a *filter* of $(X, \preceq)$ and the crown is an *ideal*[14]. 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 (BORDER). The *border* of the inheritance graph is the subset X_{bo} of minimal classes in the core: $X_{bo} \stackrel{\text{def}}{=} \{x \in X_{co} \mid y \prec x \Rightarrow y \in X_{cr}\} = \min_{\prec}(X_{co})$.

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.

Definition (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 (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 4.3.1. *Any partial regular coloring of the core can be extended in a coloring of the whole hierarchy.*

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

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

However, the proposition does not mean that any optimal regular coloring of the core can be extended in 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.

4.3.3 *Conflict graph.* Coloring amounts to graph coloring, for some *coexistence graph* of classes, attributes or methods. In the case of classes, it is defined as follows:

Definition (CLASS COEXISTENCE GRAPH). The *class coexistence graph* is the undirected graph $(X, \diamond)$, where $x \diamond y$ iff $x \neq y$ and $\exists z \in X, z \preceq x, z \preceq y$.

Of course, $\diamond$ includes $\prec$.

Definition (PROPERTY COEXISTENCE GRAPH). The *property coexistence graph* is the undirected graph $(P, \diamond)$, where $x \diamond y$ iff $x \neq y$, and $\exists z \in X, h(z, x)$ and $h(z, y)$.

The $\diamond$ relationships account for the informal use of the term 'conflict' in Section 4.3.1. However, the class coexistence graph 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 (CONFLICT GRAPH). The *conflict graph* is the undirected graph $(X_{co}, \leftrightarrow)$, where $x \leftrightarrow y$ iff $x \not\preceq y$, $y \not\preceq x$ and $\exists z \in X_{co}, z \prec x, z \prec y$.

In other words, the conflict graph is the *incomparability graph* of $(X, \preceq)$, restricted to classes with common subclasses. $\prec$ and $\leftrightarrow$ are disjoint, and the class coexistence graph ($\diamond$) is the union of both relationships. All edges in the conflict graph are between two classes in $X_{co} \backslash X_{bo}$. However, Proposition 4.3.1 does not hold if the border is removed from the core.

So, in the following, the term 'conflict' will be related to 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 appears as a simplification of the conflict graph defined by [Pugh and Weddell 1990], on the set P of all properties—attributes or methods:

Definition (PROPERTY CONFLICT GRAPH). 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:

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

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 and Weddell 1990].

PROPOSITION 4.3.2. *Let $y, y' \in P$ two class-based properties, with $c(y) = \{x\}$ and $c(y') = \{x'\}$. Then $y \leftrightarrow y'$ iff $x \leftrightarrow x'$.*

The proof is straightforward. □

As an immediate corollary, class coloring is a special case of regular property coloring.

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

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

Obviously, a perfect coloring is optimal. Proposition 4.3.1 extends to perfect colorings:

PROPOSITION 4.3.3. *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 consists in 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 conflict graph colorability. The following proposition generalizes their results to regular coloring.

PROPOSITION 4.3.4. [Pugh and Weddell 1990, Theorem 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 speak of 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 4.3.5. *There exists 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 4.3.1. *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 for each class is 2-colorable.

Proposition 4.3.5 holds also for class and method joint coloring when methods are class-based. In the case of property regular coloring, the necessary condition requires restricting the conflict graph to classes introducing properties:

PROPOSITION 4.3.6. *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 4.3.5 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. □

4.3.5 *Example.* Figure 13 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 checking the bounds.

4.4 Non-regular coloring

4.4.1 *Perfect non-regular coloring.* In a dynamic typing framework, when properties are no longer class-based, one would like to formulate conditions on perfect coloring in terms of attribute or method 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 and 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 4.4.1. [Pugh and Weddell 1993, Theorem 1] *There is a perfect k-directional property coloring if the property conflict graph is k-colorable.*

PROPOSITION 4.4.2. [Pugh and Weddell 1993, Theorem 2] *For any positive integer n, there is a hierarchy with a 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 which 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

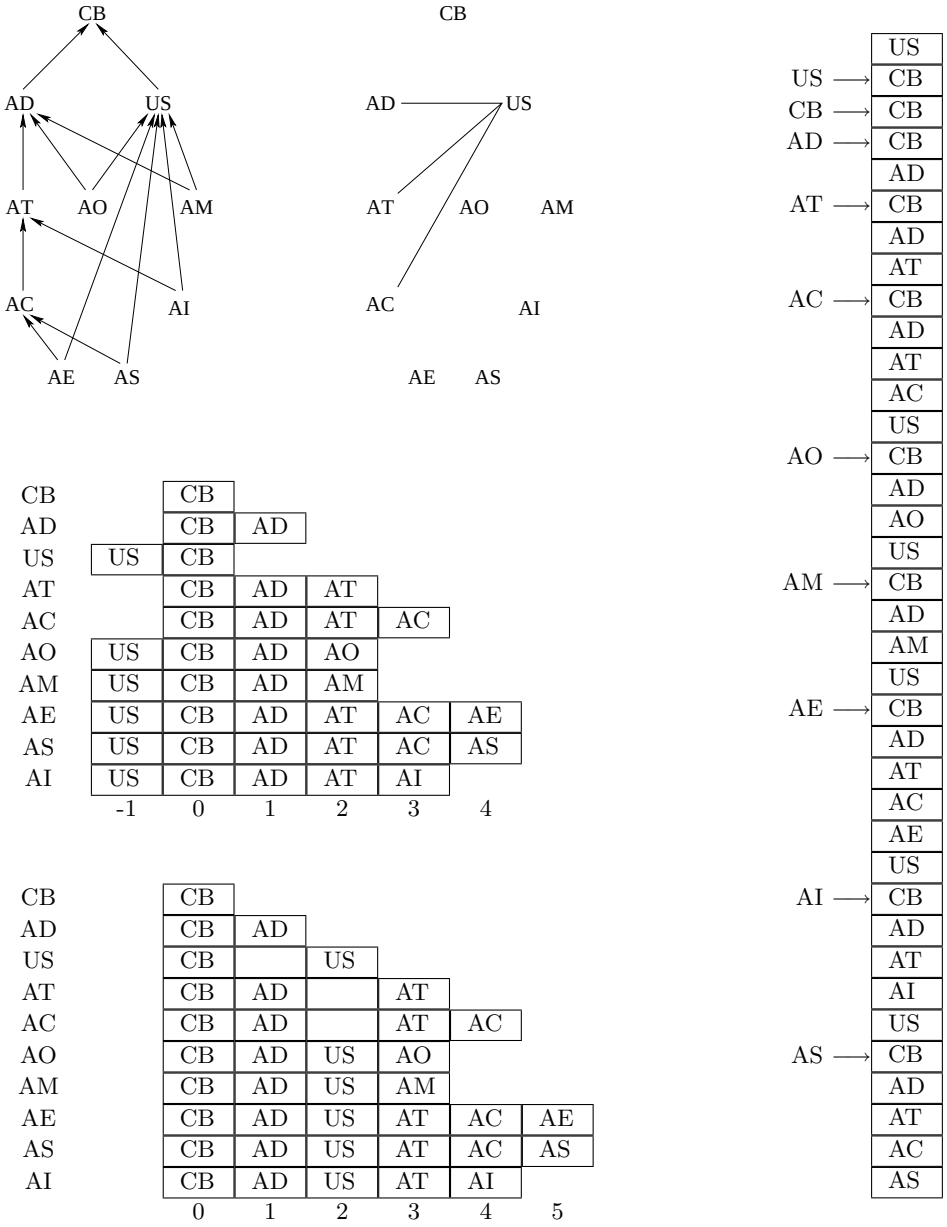

Fig. 13. Class coloring of the core of the IDL hierarchy. From top to bottom and left to right: core and conflict graph, bidirectional and unidirectional color tables, and the specific memory area gathering all bidirectional color tables.

Table III. Conditions for polynomiality when a perfect coloring exists and in a static typing framework

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

abstract classes into account, attribute coloring is no longer regular. Indeed, after the removal of some abstract class, the attributes introduced by this class would be implicitly introduced by all of its direct subclasses.

Hence, Proposition 4.4.2 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 coloring of the crown is perfect, by simply filling the holes in the abstract class layout.

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*) [Godin et al. 1998; Wille 1992]. 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 not 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*, which is NP-hard in the general case [Garey and Johnson 1979]. Any graph may be the conflict graph of some class hierarchy—one only needs to add one common subclass to each connected pair. Therefore, with this minimization criterion, optimal coloring is NP-hard.

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 and Weddell 1990; 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 framework as follows.

PROPOSITION 4.5.1. [Pugh and 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 the authors), it applies only to irregular colorings.

Regarding regular colorings, the situation slightly depends on k.

PROPOSITION 4.5.2. [Pugh and Weddell 1993, Theorem 6] *Optimal regular k-directional coloring is NP-hard, for all $k > 2$.*

PROOF. It follows from Propositions 4.3.5 and 4.3.6 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 $k > 2$. □

When $k \leq 2$, the existence of perfect regular coloring is polynomial. Therefore it has to be proven that, when there is no perfect uni- (resp. bi-) directional coloring, computing an optimal regular coloring is NP-hard. Proving it for class coloring is sufficient, since class coloring is a special case of property coloring, when exactly one property is introduced by each class.

PROPOSITION 4.5.3. [Takhedmit 2003] *Optimal k-directional class coloring is NP-hard, for all $k \leq 2$, unless a perfect k-coloring exists.*

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

Overall, optimal coloring is always NP-hard, except when there is a perfect uni- or bi-directional regular coloring (Table III).

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

5.1 Heuristics

The first heuristics family is rather naive—it is based on an adaptation of the single inheritance algorithm to multiple inheritance. It was first described in [Ducournau 2001; 2003] (in French). The second one is 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 only technical point is to order classes in some topological ordering from root(s) to leaves—in other words, superclasses are colored before subclasses (Figure 14). Regarding roots, if there are more than one root, a virtual single root must be 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 (Figure 15). 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

Algorithm: SI-COLORING
Data: a hierarchy $(X, \prec_d)$
foreach *class* $x \in X$ *in topological ordering* **do**
  if x *is the root (no superclass)* **then**
    $\chi(x) \leftarrow 0$
  else
    x has superclass y;
    $\chi(x) \leftarrow \chi(y) + 1$

Fig. 14. Single inheritance class coloring

to be colored. Regarding colors, single inheritance involves only choosing 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 coloring superclasses ($\prec$) or already colored conflicting classes ($\leftrightarrow$). Finally, once a class and a color have been chosen, the choice must be propagated to subclasses—the chosen color is inherited and the set of maximal classes must be updated—and to yet uncolored conflicting classes—the chosen color must be "frozen" in the conflicting classes because they share common subclasses with the currently colored class. Finally, once the set of maximal classes is empty, border and crown can be colored almost as in single inheritance. 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 classes as soon as they are colored. Therefore, the complex part of the algorithm is only a function of the core, not of the whole hierarchy.

On the whole, 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 class x is maximal, its color table contains inherited colors $\mathit{inh}(x) = \bigcup_{x \prec_d y} \chi^*(y) = \chi^*(x) \backslash \chi^+(x)$ and colors frozen by conflicting colored classes, $\mathit{froz}(x) = \bigcup_{x \leftrightarrow y} \chi^+(y)$, whereby y is restricted to already colored classes. In the unidirectional case, the free colors are the non-frozen holes, i.e. $\mathit{holes}(x) = [0, \max(\mathit{inh}(x))] \backslash (\mathit{inh}(x) \uplus \mathit{froz}(x))$, if non-empty, or the *next color*, i.e. the first non-frozen color greater than $\max(\mathit{inh}(x))$, i.e. $\min([\max(\mathit{inh}(x)), \infty[\backslash \mathit{froz}(x))$. There is a choice only when there are several holes—indeed, it is useless to choose the first color not frozen when there are holes, since another class ordering would produce the same coloring. This easily generalizes to k-directional coloring, by considering free_i, froz_i, holes_i and inh_i in each direction i. However, when there are no holes, there remains a choice between k next colors.

In some sense, all possible colorings can be obtained by the heuristics which only exclude colorings with unnecessary holes. For instance, a random variant will consist of 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 to check that a particular behaviour does not happen only by chance [Ducournau 2001; 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 the classes conflicting with some previ-

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;
  $Y \leftarrow X_{co} \backslash X_{bo}$; `/* uncolored classes */`
  `core-coloring`(Y) ; `/* 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 */`
  $\chi^*(x) \leftarrow$ `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
    x has superclass y;
    $\chi^*(x) \leftarrow \chi^*(y)$; `/* color table copy */`
    if $\chi^*(x)$ *has some holes* **then**
      $\chi(x) \leftarrow$ any hole
    else
      $\chi(x) \leftarrow \max(\chi^*(x)) + 1$

Fig. 15. Unidirectional class coloring—general algorithm in multiple inheritance

ously colored class, if any, and if ii) `choose-free-color` chooses a color which does not create a hole, if any, i.e. there is no hole and $\max(inh_+(x)) + 1$ or $\max(inh_-(x)) + 1$ are not frozen. The first condition ensures that the bipartition will be exact if any exists. However, a preliminary bipartition gives a better result (see Section 5.1.2).

Weighted heuristics. Weight naturally appears in the class ordering. 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 $\mathcal{O}((|X_{co}|+|\preceq_{co}|)|X_{co}|)$, i.e. $\mathcal{O}(|X_{co}|^3)$. 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 a 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* [15]. For all k, $\chi^{-1}(k)$ is an independent set of the coexistence graph—i.e. the restriction $\diamond/\chi^{-1}(k) = \emptyset$.

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 give sound results if the weight w is somewhat "additively monotonous", i.e. $w(x) > \sum_{y \prec_d x} w(y)$. This is the case when $w(x)$ is the number of subclasses of x, or its number of instances—in the latter case, x must not be abstract. 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 the previous algorithms—independent sets are called *buckets* and the notions of core, crown and border are used.

Multi-directional coloring and maximum k-cut. A 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. Note however that *maximum 2-cut*, aka *maximum bisection*, is also NP-hard in the gen-

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

```
Algorithm: CORE COLORING IS
Data: Class hierarchy (X, ≺_d);
a subset Y ⊂ X and conflict relationship ↔
Attribute to each x ∈ Y a weight w(x) ;                          /* weights */
Order Y by decreasing weight: x_1, ..., x_n ;                   /* ordering */
S ← ∅ ;                          /* ordered sequence of independent sets */
l ← 0 ;                                  /* length of the sequence */
for i=1 to n do
    m ← 0;
    for k=1 to l do
        if S[k] ⊎ {x_i} is an independent set of ≺ and ↔ then
            S[k] ← S[k] ⊎ {x_i};
            m ← k;
            exit for k
    if m=0 then
        m, l ← l + 1;
        S[m] ← {x_i}
    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] heavy */
for k=1 to l do
    foreach x ∈ S[k] do χ(x) ← k
```

Fig. 16. 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_i and elements in $S[k]$ need to be checked.

eral case—hence, if the conflict graph is not bipartite, then the heuristics must be 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 and Royer 1992], but experiments by [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 and Royer 1992; Ducournau 1997].

Static typing framework. The naive heuristics of Figure 15 extend easily to property coloring—for each class x, instead of allocating a single color, the algorithm has to allocate as many colors as there are properties introduced by x. Regarding n-directional coloring, the properties introduced by a class may be colored in different directions, but only when there are holes to fill. Anyway, the preliminary partition is only based on the conflict graph, possibly restricted to classes introducing properties.

Dynamic typing framework. When the properties may be introduced by more than one class—$c(p)$ is not a singleton—relying only on the class hierarchy and conflict graph is not enough. However, there are likely not many properties intro-

```
Algorithm: INCR-CLASS-COLORING
Data: a class hierarchy (Y, ≺_d);
a loaded class x ∈ Y, minimal in (Y, ≺_d);
a partial class coloring χ on the filter Y\x
begin
    super_d ← {y ∈ Y | x ≺_d y};                          /* direct superclasses */
    k ← max(χ(Y\x));
    if super_d = ∅ ;                                  /* x is a root (no superclass) */
    then
        χ(x) ← 0
    else
        if super_d = {y} ;                                /* like crown coloring */
        then
            χ*(x) ← χ*(y) ;                               /* color table copy */
        else
            C[0..k] an array of empty sets ;             /* multiple inheritance */
            super ← {y ∈ Y | x ≺ y} ;                     /* all superclasses */
            foreach class z ∈ Y such that x ≺ z do
                C[χ(z)] ← C[χ(z)] ⊎ {z}
            cs ← {c | |C[c]| > 1} ;                          /* conflict set */
            if cs ≠ ∅ then
                Z ← ∅;
                foreach color c ∈ cs ;                    /* conflict resolution */
                do
                    u ←choose-unchanged(C[c]) ;              /* left unchanged */
                    Z ← Z ⊎ C[c]\{u};
                    C[c] ← {u}
                recompute-colors(Z)
            χ*(x) ← {c | C[c] ≠ ∅};
        if χ*(x) has some holes then
            χ(x) ← any hole
        else
            χ(x) ← max(χ*(x)) + 1;
            if χ(x) > k ;                          /* only with fixed-size tables */
            then
                foreach class z ∈ Y do
                    allocate and copy larger color table
end
```

Fig. 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.

duced by more than one class—only 27% in the SMALLTALK hierarchy tested by [Ducournau 1997]—so the main work can rely on class hierarchy. First, one must distinguish *class-based* properties, introduced by only one class, from other properties. Then, introducing classes have to be associated with each property of the second kind. Coloring such a property will involve "freezing" the selected color in all classes which have the property. This is roughly the only point to add to the heuristics of Figure 15.

5.1.4 *Incremental class coloring.* In their proposition, [Palacz and 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 counterbalanced 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.

Anyway, with fixed or variable size color tables, coloring heuristics have a very efficient best case. When loading a class, 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 possible. 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—a $\mathcal{O}(k+n)$ 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 colors tables must be reallocated—this is an $\mathcal{O}(kN)$ operation, where N is the number of classes. With 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 a $\mathcal{O}(k+n)$ 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 the 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 the ones that must be recolored. The optimum is of course NP-complete 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 conflict graph is first required, with a $\mathcal{O}(n^2)$ complexity since all superclasses might be conflicting, and all other classes might already conflict with the superclasses. The heuristics are sketched in Figure 17.

5.2 Benchmarks

We tested coloring on several large benchmarks commonly used in the object-oriented implementation community[16], e.g. by [Vitek et al. 1997; Zibin and Gil 2001]. The benchmarks are abstract schemes of large class libraries, from various languages: Java (from IBM-SF to IBM-XML in Table IV), Cecil (Cecil, Vortex3), Dylan, Clos (Harlequin), Self, Eiffel (SmartEiffel), Eiffel-like (Lov and Geode) and Prm (PRM-dotc). All benchmarks are classical, except PRM-dotc which is part of the forthcoming Prm compiler [Privat and Ducournau 2005; Privat 2006]. Here, all benchmarks are multiple inheritance hierarchies and, in Java benchmarks, there is no distinction between classes and interfaces.

Note that we only investigated regular coloring—i.e. each set of properties is interpreted as if it were in static typing—and we restricted our tests to uni- and

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

Table IV. Statistics on class hierarchies and conflict graphs. The Table displays first the number of classes, in the whole hierarchy and in the core, then the number of non-isolated vertices and edges in the conflict graph. Regarding vertices, classes which introduce methods ('wm') or attributes ('wa') are distinguished. The Table ends with statistics on connected-components in the conflict graph—number, total size and maximum size—according to whether they are bipartite or not.

	class		conflict graphs						connected components				
	number		vertices			edges			bipartite		non-bipartite		
	total	core	tot	wm	wa	avg	max	total	nb	tot	nb	tot	max
IBM-SF	8793	4770	2566	1852		13.3	2057	17099	3	8	2	2558	2550
MI-jdk1.3.1	7401	1512	762	572		6.6	205	2529	21	61	8	701	596
MI-Orbix	2716	271	177	106		4.7	110	416	7	23	1	154	154
MI-Corba	1699	383	213	92		7.6	144	810	0	0	1	213	213
MI-Orbacus	1379	502	315	184		6.5	166	1025	3	9	1	306	306
MI-HotJava	736	217	108	91		10.8	54	583	2	8	2	100	95
JDK.1.0.2	604	105	61	52		3.9	17	120	2	5	2	56	34
IBM-XML	145	61	42	27		4.8	27	100	2	6	2	36	32
vortex3	1954	696	392	80		7.9	53	1558	13	31	8	361	164
Self	1802	154	104	97		27.4	62	1427	1	3	1	101	101
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
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
PRM-dotc	222	53	37	29	17	3.5	9	64	2	9	2	28	15

bi-directional coloring.

5.2.1 *Hierarchies and conflict graphs.* Table IV 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 is quite a bit smaller than the whole hierarchy. The conflict graph is even smaller, and the restriction to classes introducing properties is effective—columns 'wm' and 'wa'. 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. 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 appears if one considers 2-connected components[17]. This proves that, on all the considered benchmarks, bidirectional class coloring will be far from perfect.

[17]Obviously, 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.

Table V. Statistics on class coloring. The first part of the Table displays statistics on the class hierarchy. The next part presents the total hole number according to the minimization criterion. The last part presents, in the bidirectional case, the global hole rate and the average color table size per class—the latter must be compared with the average superclass number, column 4.

	hierarchy				total hole number minimization criterion			bidirectional hole rate	bidirectional size	
	$\|X\|$	$\|\preceq\|$	$\|\preceq\|/\|X\|$ avg	max	color#	uni-	bi-		avg	max
IBM-SF	8793	80860	9.2	30	182930	19746	4507	.06	9.7	30
MI-jdk1.3.1	7401	32480	4.4	24	145144	4331	1307	.04	4.6	24
MI-Orbix	2716	7560	2.8	13	27748	347	18	.00	2.8	13
MI-Corba	1699	6551	3.9	18	24031	533	115	.02	3.9	18
MI-Orbacus	1379	6244	4.5	19	19957	935	181	.03	4.7	19
MI-HotJava	736	3768	5.1	23	13160	1210	248	.07	5.5	23
JDK.1.0.2	604	2802	4.6	14	5654	108	10	.00	4.7	14
IBM-XML	145	640	4.4	14	1390	82	14	.02	4.5	14
vortex3	1954	14146	7.2	30	44474	1557	521	.04	7.5	30
Self	1802	55639	30.9	41	18243	935	625	.01	31.2	41
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
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
PRM-dotc	222	912	4.1	11	1530	61	11	.01	4.2	11

5.2.2 *Coloring.* Table V presents the results of class coloring, which are to 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. Note that minimizing the color number is not good from a spatial standpoint—though, in all cases, the heuristics computes the exact optimal color number. This implies that, when applied to classes only, as in [Vitek et al. 1997; Palacz and 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 column give the average and maximum size of the color tables, which must be compared to the superclass number (columns 4 and 5).

Table VI presents statistics of methods, introduced or inherited per class, together with method bidirectional coloring. The difference between columns "color table" and "inherited" is the number of holes. One observes that, in most benchmarks, the maximum color number is exactly the maximum number of inherited methods—this means that the heuristics give the optimal solution of the underlying minimum graph coloring problem. 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[18].

[18]Remember that SMI is the C++ implementation when the keyword `virtual` is used for all

Table VI. Statistics on method bidirectional coloring. The Table displays first the number of methods introduced or inherited per class, then the size of bidirectional method tables—which must be compared with the number of inherited methods—with the associated hole rate. The last part presents the method table size in "standard" multiple inheritance (SMI), i.e. C++. All data are 2-fold, namely average and maximum per class, and represent cardinal numbers (#) or rates (%) relative to the total size of method tables.

	methods				method coloring				SMI	
	introduced		inherited		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	.29	15.9	231.3	2063
MI-jdk1.3.1	1.3	149	19.2	243	19.8	243	.03	15.2	72.4	1391
MI-Orbix	0.4	64	8.3	109	8.4	109	.00	2.1	23.0	534
MI-Corba	0.4	43	8.0	67	9.9	81	.19	5.0	26.9	427
MI-Orbacus	1.2	74	18.0	137	18.3	137	.01	2.3	68.3	761
MI-HotJava	1.8	80	34.2	189	36.0	189	.05	17.8	134.8	817
JDK.1.0.2	5.3	75	37.0	158	37.1	158	.00	0.6	127.3	691
IBM-XML	2.5	29	16.1	57	16.3	57	.01	1.2	50.4	284
vortex3	0.5	148	156.5	204	156.9	204	.00	0.1	1117.7	4994
Self	14.6	233	577.4	969	586.4	969	.02	246.5	3706.2	10098
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
Geode	6.1	193	231.8	880	291.0	892	.20	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	.24	4.7	422.1	1590
SmartEiffel	12.2	222	135.3	324	135.4	324	.00	0.2	743.5	1576
PRM-dotc	4.2	103	80.3	145	80.5	145	.00	0.3	299.6	971

Table VII. Statistics on attribute coloring. The Table displays first the number of attributes introduced or inherited per class, then the size of attribute tables—which must be compared with the number of inherited methods—with the associated hole rates in the bidirectional and unidirectional cases. The last part presents the subobject number in "standard" multiple inheritance (SMI), i.e. C++. All data are 2-fold, namely average and maximum per class, and represent cardinal numbers (#) or rates (%) relative to the total size of the object layout.

	attribute number				attribute coloring						SMI	
					color		hole rate				subobject	
	introduced		inherited		number		bidirect.		unidirect.		number	
	avg#	max#	avg#	max#	avg#	max#	avg%	max%	avg%	max%	avg#	max%
Geode	2.2	182	10.9	217	12.7	218	.11	7.0	.14	19.7	14.0	10.0
Unidraw	2.6	36	8.3	47	9.3	48	.04	1.0	.00	0.5	4.0	2.0
Lov-obj-ed	2.9	74	8.2	105	9.3	106	.06	3.0	.16	6.0	8.5	10.5
SmartEiffel	2.5	39	4.9	44	5.9	45	.00	0.0	.00	0.0	8.6	5.0
PRM-dotc	1.2	20	3.6	20	4.6	21	.06	0.5	.03	0.7	4.1	3.0

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. Table VII presents statistics on attributes. They are analogous to that on methods, with a few differences—the unidirectional case is added and, in the comparison with SMI, the Table gives the subobject number instead of the table size. Moreover, the "color number" col-

inheritance relationships. This is the only way to define classes that are fully compatible with further multiple specialization. This is however not the common C++ programming style, so SMI numbers do not reflect actual C++ programs.

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

	initialization		coloring			total
	read	init	conflict	bipartite	color	avg/class
IBM-SF	3598	1041	447	747	7575	1.5
MI-jdk1.3.1	1238	372	69	141	1526	0.5
MI-Orbix	221	123	13	22	345	0.3
MI-Corba	110	95	20	43	362	0.4
MI-Orbacus	208	99	32	87	390	0.6
MI-HotJava	80	64	17	46	176	0.5
JDK.1.0.2	168	34	4	7	98	0.5
IBM-XML	18	9	3	5	29	0.4
vortex3	163	404	44	54	318	0.5
Self	262	186	28	91	526	0.6
Cecil	110	59	17	24	149	0.4
dylan	43	46	4	3	125	0.2
harlequin	46	45	21	46	167	0.5
Geode	204	472	108	277	691	1.3
Unidraw	71	27	1	1	86	0.3
Lov-obj-ed	71	40	20	69	175	0.9
SmartEiffel	66	28	3	3	94	0.5
PRM-dotc	48	11	2	4	44	0.5

umn includes the pointers at method table—the usual one, at color `#tableOffset` (i.e. 0), together with the extra pointer required for garbage collection for all objects where negative colors are used. One observes that the solution to the underlying minimum graph coloring problem is still optimal—inherited attribute and maximum color numbers differ by exactly one in all benchmarks. The comparison between uni- and bi-directional coloring shows that the gain from bidirectionality is mostly counterbalanced by the extra method table pointer (Figure 7). The comparison with SMI shows that the numbers of inherited attributes, colors and subobjects have the same order of magnitude. Hence, SMI—in the ALL implementation [Sweeney and Burke 2003]—roughly doubles object size and coloring markedly reduces the dynamic space overhead—and of course it markedly improves all other aspects. Of course, other SMI implementations, with extra compiler-generated fields allocated in the object layout—like VBPTRs or with the ARM implementation [Sweeney and Burke 2003]—are markedly more space-consuming. Therefore, attribute coloring is a significant improvement w.r.t. SMI implementations. However, in the attribute case, one may be more demanding and one must also take care of the maximum hole rate, which is here up to 7, in Geode (but more than 10 with SMI). This means that, in the layout of the instances of some classes, there would be 7-fold more holes than attributes. This is better than with unidirectional coloring or SMI, but it might be unacceptable if the considered class is intensively instantiated—imagine that the Lisp cells (instances of the `cons` class) occupy 16 words, instead of 3! Therefore, the *accessor simulation* alternative might be considered (Section 3.3.2).

5.2.3 *Performance.* Table VIII describes the performance of the heuristics on a Intel® Core™ 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; 2005]. Many other statistics are computed and coloring was not specially optimized—on the contrary, the general algorithm (Figure 15) 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 two steps are not specific to coloring: 'read' is the input phase, which build from file the data structures; 'init' step makes basic computations and manages inheritance. 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 about 1 ms per class.

6. RELATED WORKS

The literature on the implementation of object-oriented languages is rather large. Problems arise when dynamic typing or multiple inheritance are considered. Separate compilation and dynamic loading worsen the situation.

Dynamic typing. SMALLTALK is an archetype of dynamic typing and single inheritance in a dynamic loading framework. The question of attributes is solved by encapsulation—i.e. accesses are reserved to `self`. Message sending is implemented with various non-constant time techniques, such as hashtables optimized with caches or, alternatively, caches with hashtables for cache misses [Deutsch and Schiffman 1984; Hölzle et al. 1991]. When multiple inheritance is added to dynamic typing, as in CLOS, the question of attributes can no longer rely on the position invariant. Plain accessors—aka *field dispatching* [Zibin and Gil 2003]—is the solution used in many languages: CLOS, CECIL [Chambers 1993] or DYLAN [Shalit 1997]. Of course, these accessors could be simulated by offsets, with one offset per attribute. This yields non-constant time accesses.

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. As for usual method invocation in dynamic typing, the most common technique is based on hashtables and caches [Kiczales and 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. Methods are now orthogonal to classes—they are called *generic functions* in CLOS—and dispatch tables are attached to them, not to classes. 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 and 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 and Sweeney 1999; Eckel and Gil 2000]. This

global optimization could be applied in a global compilation setting but it seems difficult to apply it at link-time without deeply modifying C++ compilers.

Row displacement. In a global setting, an alternative to coloring has been proposed, *row displacement* [Driesen 1993; Driesen and Hölzle 1995; Driesen 2001]. This sparse table compression technique, from [Tarjan and Yao 1979], involves superposing the 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. Rows may be class-based or selector-based. According to Driesen, selector-based row displacement achieves better compression than classed-based row displacement, which is itself 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 the framework of dynamic loading, though the cost of inserting a new row may not be negligible, and the compression rate is not very good. However, row displacement does not apply to attributes—hence, it should be coupled with accessor simulation (Section 3.3.2).

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 to compute the *concrete type*, i.e. the set of possible receiver types in all executions of the considered program. In this framework, 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[19] [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.

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

[19]Previously named Small Eiffel.

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.

Mixed techniques. A combination of table-based and tree-based techniques is also possible. Vitek and Horspool [1994] propose to merge different method tables when they have almost the same content—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 all method tables but it appears that the size of the trees mostly counterbalances the gain in method tables [Vitek and Horspool 1996; Ducournau 1997]. However, this approach provides an incremental variant of coloring, which can be interesting in a dynamic typing framework since message sending now requires an additional parameter, the selector, which must be tested to detect the "message not understood" error. This extra parameter may also be used in the dispatch trees, when there are several possible selectors for one table entry. However, in static typing, as this extra parameter is not required in the general case, this would add a uniform and significant overhead. [Alpern et al. 001a] 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 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 similar 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 time efficiency. For instance, Muthukrishnan and 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. 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. Zibin and Gil [2001] propose a combination of relative numbering and class coloring, called *PQ-encoding*, which achieves high compression rates. On the other hand, a generic approach to subtype testing involves bit-vector encoding of hierarchies [Caseau 1993; Habib and 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.

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 and 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 2005] 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* (Section 3.3.2). 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, its constant of time is about 2-fold that of coloring for subtype test and method invocation, and substantially more for accesses 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 and Weddell 1990; Ducournau 1991; Vitek et al. 1997; Zibin and Gil 2003]. The main goal of this paper was to gather these different works and to 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 and 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 and Weddell 1990; 1993]. As expected, the coloring problem is NP-hard, except in some few cases where there is a perfect regular 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—except for [Vitek et al. 1997]. Overall, coloring appears to be an efficient implementation technique, likely one of the most efficient in a multiple inheritance framework. 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 may be postponed at link-time but is incompatible with dynamic loading.

The main perspective of this work is to use coloring for implementing real object-oriented languages. This is, however, a hard work, since implementation of ba-

sic 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, any others?—and they all present inheritance-related features that would not easily match with coloring. So the solution has been to specify and implement a new object-oriented language, Prm. This was a harder work, that we are currently achieving [Privat and Ducournau 2005; Privat 2006]. The bootstrap of the Prm compiler will provide a test-bed dedicated to the run-time assessment of different implementation techniques, including coloring, binary tree dispatch, perfect hashing, etc. The Prm language specifications include a powerful notion of module and class refinement [Privat and Ducournau 2006]. This is the basis of the Prm compiler modular architecture and it will make it easy to replace, e.g., the coloring module, by the BTD module, and test it.

Incremental coloring. The major drawback of coloring is its non-incrementality, hence its relative incompatibility with dynamic loading. Therefore, the main issue concerning coloring would be to find an efficient incremental variant. We have briefly proposed a solution, inspired from [Palacz and Vitek 2003], available only in a static typing framework (Figure 10). This solution works but would be likely inefficient at run-time, especially for attributes. So this can be only 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 schema must be also examined. For instance, method tables themselves could be also recomputed but could it 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), two techniques can be understood as possible propositions for incremental coloring. First, the mixed technique proposed by [Vitek and Horspool 1994] could be used at load-time for solving color conflicts between already loaded superclasses of the currently loaded class. Secondly, perfect hashing [Ducournau 2005] is an exact incremental generalization of coloring in the particular case of subtype tests. However, both 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'—advocated for long in the object-oriented community [Szyperski 1992]—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. It is not too inefficient for method invocation and subtype test, but access to attributes would not be truly better than with the proposition of Figure 10. So, like incremental coloring, this can be only envisaged for multiple subtyping languages. A last alternative should also be considered, namely class-based row displacement.

Currently, there is no convincing incremental version of coloring, and the search for such a variant or for an alternative is certainly the main perspective of this

work.

REFERENCES

Aksit, M. and Matsuoka, S., Eds. 1997. *Proceedings of the Eleventh European Conference on Object-Oriented Programming, ECOOP'97*. LNCS 1241. Springer.

Alpern, B., Cocchi, A., Fink, S., and Grove, D. 2001a. Efficient implementation of Java interfaces: Invokeinterface considered harmless. See OOPSLA [2001], 108–124.

Alpern, B., Cocchi, A., and Grove, D. 2001b. Dynamic type checking in Jalapeño. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*.

Amiel, E., Gruber, O., and Simon, E. 1994. Optimizing multi-methods dispatch using compressed dispatch tables. See OOPSLA [1994], 244–258.

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

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

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., Shalit, A. L. M., and Withington, P. T. 1996. A monotonic superclass linearization for Dylan. In *Proc. OOPSLA'96*. SIGPLAN Notices, 31(10). ACM Press, 69–82.

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

Cardelli, L., Ed. 2003. *Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP'2003*. LNCS 2743. Springer.

Caseau, Y. 1993. Efficient handling of multiple inheritance hierarchies. See OOPSLA [1993], 271–287.

Chambers, C. 1993. The cecil language, specification and rationale. Technical Report 93-03-05, University of Washington.

Cheung, B. and Grogono, P. 1992. Compact record layouts for multiple inheritance. Tech. Rep. OOP-92-01, Department of Computer Science, Concordia University. January.

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

Cohen, N. 1991. Type-extension type tests can be performed in constant time. *Programming languages and systems 13,* 4, 626–629.

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

Conroy, T. J. and Pelegri-Llopart, E. 1983. An assessment of method-lookup caches for smalltalk-80. In *Smalltalk-80 Bits of History, Words of Advice*, Krasner, Ed. 238–247.

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

Czech, Z. J., Havas, G., and Majewski, B. S. 1997. Perfect hashing. *Theor. Comput. Sci. 182,* 1-2, 1–143.

Desnos, N. 2004. Un garbage collector pour la coloration bi-directionnelle. M.S. thesis, Université Montpellier 2.

Deutsch, L. and Schiffman, A. 1984. Efficient implementation of the smalltalk-80 system. In *Proc. of ACM Symp. on Principles of Prog. Lang. (POPL'84)*. 297–302.

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

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

Driesen, K. 1993. Selector table indexing and sparse arrays. See OOPSLA [1993], 259–270.

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

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. See OOPSLA [1995], 141–155.

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

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

Ducournau, R. 2001. La coloration : une technique pour l'implémentation des langages à objets à typage statique. I. La coloration de classes. Tech. Rep. 01-225, LIRMM, Université Montpellier 2.

Ducournau, R. 2002a. Implementing statically typed object-oriented programming languages. Tech. Rep. 02-174, LIRMM, Université Montpellier 2. (submitted to *ACM Computing Surveys*; revised July 2005).

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

Ducournau, R. 2003. La coloration : une technique pour l'implémentation des langages à objets à typage statique. II. La coloration de méthodes et d'attributs. Tech. Rep. 03-036, LIRMM, Université Montpellier 2.

Ducournau, R. 2005. Perfect hashing as an almost perfect subtype test. Tech. Rep. 05-058, LIRMM, Université Montpellier 2. (submitted to *ACM TOPLAS*, revised December 2006).

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1994. Proposal for a monotonic multiple inheritance linearization. See OOPSLA [1994], 164–175.

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

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

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

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

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

Gagnon, E. M. and Hendren, L. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*. 27–40.

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

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

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

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

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

Habib, M. and Nourine, L. 1994. Bit-vector encoding for partially ordered sets. In *ORDAL*, V. Bouchitté and M. Morvan, Eds. LNCS 831. Springer, 1–12.

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

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

Holst, W. and Szafron, D. 1997. A general framework for inheritance management and method dispatch in object-oriented languages. See Aksit and Matsuoka [1997], 271–301.

Huang, S.-K. and Chen, D.-J. 1992. Two-way coloring approaches for method dispatching in object-oriented programming systems. In *Proc. of Computer Software and Applications Conference. COMPSAC '92.* 39–44.

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

Jensen, T. R. and Toft, B. 1995. *Graph Coloring Problems*. John Wiley.

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

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

Krall, A., Vitek, J., and Horspool, R. 1997. Near optimal hierarchical encoding of types. See Aksit and Matsuoka [1997], 128–145.

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

Mehlhorn, K. and Tsakalidis, A. 1990. Data structures. In *Algorithms and Complexity*, J. Van Leeuwen, Ed. Handbook of Theoretical Computer Science, vol. 1. Elsevier, Amsterdam, Chapter 6, 301–341.

Meyer, B. 1992. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA.

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

Myers, A. 1995. Bidirectional object layout for separate compilation. See OOPSLA [1995], 124–139.

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

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

OOPSLA 1993. *Proceedings of the Eighth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'93*. SIGPLAN Notices, 28(10). ACM Press.

OOPSLA 1994. *Proceedings of the Ninth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'94*. SIGPLAN Notices, 29(10). ACM Press.

OOPSLA 1995. *Proceedings of the Tenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'95*. SIGPLAN Notices, 30(10). ACM Press.

OOPSLA 1997. *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*. SIGPLAN Notices, 32(10). ACM Press.

OOPSLA 2001. *Proceedings of the Sixteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press.

Palacz, K. and Vitek, J. 2003. Java subtype tests in real-time. See Cardelli [2003], 378–404.

Pfister, B. H. C. and Templ, J. 1991. Oberon technical notes. Tech. Rep. 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik.

Privat, J. 2006. PRM, the language. version 0.2. Tech. Rep. 06-029, LIRMM, Université Montpellier 2.

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

Privat, J. and Ducournau, R. 2006. Multiple inheritance, class refinement and modules at the light of meta-modeling. Tech. Rep. 06-015, LIRMM, Université Montpellier 2.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)*. ACM SIGPLAN Notices, 25(6). 85–91.

Pugh, W. and Weddell, G. 1993. On object layout for multiple inheritance. Tech. Rep. CS-93-22, University of Waterloo.

Queinnec, C. 1998. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters 64,* 6, 315–321.

Schubert, L., Papalaskaris, M., and Taugher, J. 1983. Determining type, part, color and time relationship. *Computer 16*, 53–60.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley.

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

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

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

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

Takhedmit, P. 2003. Coloration de classes et de propriétés : étude algorithmique et heuristique. M.S. thesis, Université Montpellier 2.

Tarjan, R. E. and Yao, A. C. C. 1979. Storing a sparse table. *Comm. ACM 22,* 11, 606–611.

Toft, B. 1995. Colouring, stable sets and perfect graphs. In *Handbook of Combinatorics*, R. L. Graham, M. Grötschel, and L. Lovász, Eds. Vol. 1. Elsevier, MIT Press, Chapter 4, 233–288.

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

Vitek, J., Horspool, R., and Krall, A. 1997. Efficient type inclusion tests. See OOPSLA [1997], 142–157.

Vitek, J. and Horspool, R. N. 1996. Compact dispatch tables for dynamically typed object oriented languages. In *Int. Conf. on Compiler Construction (CC'96)*. LNCS 1060. Springer, 309–325.

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

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. See OOPSLA [1997], 125–141.

Zibin, Y. and Gil, J. 2001. Efficient subtyping tests with PQ-encoding. See OOPSLA [2001], 96–107.

Zibin, Y. and Gil, J. 2003. Two-dimensional bi-directional object layout. See Cardelli [2003], 329–350.

Received November 2006