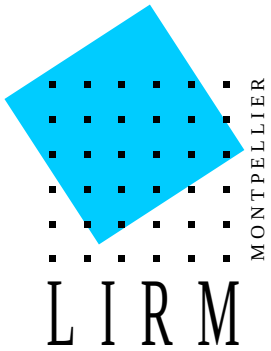

LABORATOIRE D'INFORMATIQUE,
DE ROBOTIQUE ET DE MICROÉLECTRONIQUE
DE MONTPELLIER

Unité Mixte CNRS – Université Montpellier II
C 55060

RAPPORT DE RECHERCHE

La coloration : une technique pour la compilation des langages à objets à typage statique. I. La coloration de classes

Roland Ducournau

7 décembre 2001, révisé le 11 juillet 2002 R.R.LIRMM 2001–225

161, Rue Ada – 34392 Montpellier Cedex 5 – France
Tél: (33) 04 67 41 85 85 – Fax: (33) 04 67 41 85 00

Résumé

L'implémentation des langages à objets n'a toujours pas vu de solution simple et efficace en cas d'héritage multiple et de compilation séparée. En particulier, les implémentations existantes, celle de C++ par exemple, ont un surcoût important même en cas d'héritage simple. Parmi les nombreuses techniques alternatives qui ont été proposées, la coloration de sélecteurs a été écartée, car les premières expérimentations ont été décevantes et que l'approche paraissait implicitement incompatible avec une compilation séparée. Nous proposons, dans cet article, une utilisation du principe de la coloration qui offre une efficacité supérieure, sur tous les points, à celle de l'implémentation standard de l'héritage multiple et qui, en cas d'héritage simple, donne exactement l'implémentation standard. La technique est compatible avec une compilation séparée, au prix d'une phase de pré-édition de liens, qui calcule la coloration proprement dite.

Abstract Implementing object-oriented programming languages remains difficult in case of multiple inheritance and separate compilation. Existing implementations, e.g. C++, have a considerable overhead, even in case of single inheritance. Among the various approaches proposed as alternatives, selector coloring has been rejected due to negative experiments and supposed incompatibility with separate compilation. In this paper, we propose to use coloring to solve all the problems of the standard multiple inheritance implementation, in the context of static typing and separate compilation. We show that, in case of single inheritance, coloring gives the same implementation as standard single inheritance implementation, and that coloring is always better, on all points, than standard multiple inheritance implementation. Moreover, coloring can been done at link-time.

Mots-clés : langage à objets, implémentation, compilation séparée, édition de liens, envoi de message, coloration de sélecteurs, vérification de type.

Table des matières

Table des figures

Chapitre 1

Introduction

Dans les langages objets à typage statique, l'envoi de message, ou liaison tardive, s'implémente en général par des tables, appelées tables de fonctions virtuelles en C++, qui permettent de le réduire à un simple appel de fonction, modulo un nombre limité d'indirections supplémentaires. Autant ces techniques sont simples et efficaces en héritage simple, autant leur surcoût est important en héritage multiple, au moins dans le cadre de la compilation séparée dans lequel nous nous plaçons : [Ellis et Stroustrup, 1990, chapitre 10] en est le meilleur exemple.

De très nombreux schémas d'implémentation ont été proposés : [Ducournau, 1997] en dresse un panorama pour les langages à typage dynamique et [Ducournau, 2001a] pour les langages à typage statique. Parmi ces techniques, la coloration de sélecteurs a été introduite par [Dixon *et al.*, 1989] puis elle a été écartée de la plupart des études à la suite d'une expérimentation aux résultats assez négatifs [André et Royer, 1992] : le temps de calcul de la coloration était prohibitif, ce qui s'explique par le fait que le problème algorithmique sous-jacent est une coloration de graphe, bien connu pour être NP-difficile. Pourtant, nous avions déjà utilisé cette technique, de façon relativement marginale il est vrai, dans le langage YAFOOL [Ducournau, 1991] et une étude ultérieure plus systématique devait montrer que la technique pouvait être utilisée à grande échelle, avec des résultats assez convaincants [Ducournau, 1997]. Alors que notre étude précédente était surtout centrée sur une compilation globale des langages à typage dynamique, nous nous intéressons ici à la compilation séparée des langages à typage statique. L'application de la coloration aux méthodes, aux attributs et aux classes est envisagée et une solution intermédiaire est préférée. La coloration elle-même est améliorée avec la bidirectionnalité, introduite par [Pugh et Weddel, 1990] et adoptée dans d'autres approches [Myers, 1995 ; Eckel et Gil, 2000]. Tous les mécanismes fondamentaux de la programmation par objets, concernés par cette implémentations, sont examinés : envoi de message (ou liaison tardive), accès aux attributs, *casting* ascendant et descendant, vérification de types, redéfinition covariante des types et généricité. A l'issue de cette étude, il apparaît que la coloration fournit une alternative partout meilleure que l'héritage multiple standard, et aussi bonne que l'héritage simple sur la plupart des points. La globalité de la coloration en constitue le seul handicap, mais nous montrons qu'elle peut être repoussée à l'édition de liens.

Le plan de ce rapport est le suivant.

La première partie introduit la technique de coloration et la compare aux implémentations standard (article à paraître à LMO'02). La section 2 présente les implémentations standards, en héritage simple ou multiple. La section suivante est consacrée à la coloration : la technique est présentée, ainsi que son application aux méthodes, aux attributs et aux classes. L'implémentation définitive ne repose pas sur la coloration d'attributs mais sur des décalages basés sur la coloration de classes. La possibilité de mise en œuvre lors de l'édition de liens est justifiée. La section 4 fait un rappel des résultats récents sur l'implémentation des langages à objets et la compilation de l'envoi de message, pour montrer que, malgré les évolutions rapides de l'architecture des processeurs, les techniques de tables ne sont pas obsolètes et que, dans ce contexte, le gain d'efficacité apporté par la coloration justifie certainement l'investissement dans une chaîne de traitement des programmes un peu plus compliquée. En conclusion de la partie, nous montrons différentes perspectives de l'implémentation des langages à objets.

La seconde partie, étudie en profondeur la coloration de classes, en proposant diverses heuristiques et

en décrivant et en analysant les expérimentations qui en ont été faites.

La troisième partie fait la même chose pour la coloration de méthodes ou d'attributs (partie repoussée dans un rapport de recherches ultérieur).

Des annexes, détaillents diverses statistiques.

Première partie

La coloration

Chapitre 2

Les implémentations standards

Si l'implémentation en héritage simple correspond assez à l'intuition du programmeur, l'héritage multiple entraîne, en compilation séparée, des conséquences qui ne sont pas toujours bien perçues, d'autant que la principale source d'information en la matière vient de C++ [Ellis et Stroustrup, 1990, chapitre 10]. Or les spécifications de C++ comportent de nombreuses scories destinées à éviter, autant que possible, le surcoût de l'héritage multiple et de la liaison tardive. Par rapport à C++, dans cet article, nous considérerons que le mot-clé `virtual` est partout utilisé, pour les méthodes comme pour l'héritage (voir [Ducournau, 2001a], par exemple, pour l'héritage « non virtuel »).

2.1 En héritage simple

En héritage simple, un objet est implémenté comme une table de ses attributs, auxquels s'ajoute un pointeur sur une table de méthodes, commune à toutes les instances d'une même classe, qui contient les adresses des méthodes. L'implémentation se caractérise par une totale invariance vis-à-vis du type statique :

Invariant 2.1 *Une référence — paramètre, variable, attribut ou valeur de retour d'un appel fonctionnel — sur un objet est invariante relativement à son type statique.*

Invariant 2.2 *Chaque attribut ou méthode a un indice non ambigu invariant par héritage, donc indépendant du type statique du receveur.*

L'accès à un attribut se compile par un accès direct dans l'objet, à un indice constant, et l'envoi de message nécessite juste une indirection supplémentaire. L'invariance des références rend l'implémentation du *casting* ascendant — lorsque l'on affecte la valeur d'une entité d'un type statique X à une entité d'un type statique Y, super-type de X — aussi transparente que sa syntaxe. De même, les cas de redéfinition covariante qui sont sûrs (type de retour de méthode ou attribut en lecture) ne nécessitent aucun traitement particulier. Le *casting* descendant (ou *downcast*) est plus problématique puisqu'il nécessite une vérification dynamique de type : il revient à faire l'hypothèse qu'une entité de type statique X est en fait d'un sous-type Y. Il peut être effectué au travers d'une affectation ou d'un passage de paramètre, par une construction comme `typecase` en THETA [Liskov *et al.*, 1995] ou par les tentatives d'affectation d'EIFFEL [Meyer, 1997]. En héritage simple, la vérification de type s'implémente par une double numérotation des classes, notée n_1 et n_2, obtenue par un parcours en profondeur : n_1 est affecté à la descente, après incrément du compteur, et n_2 à la remontée, sans incrément. Alors, $D \prec C \Leftrightarrow n_1(C) < n_1(D) \leq n_2(C)$. Lorsque la vérification ne met en jeu que des types cibles statiques, seul n_1 nécessite d'être stocké.

La redéfinition covariante du type des paramètres n'est pas sûre du point de vue des types — règle dite de contravariance — mais elle est justifiée par les besoins de la modélisation [Meyer, 1997 ; Ducournau, 2001b]. Pour les langages comme EIFFEL qui adoptent cette politique covariante, toute redéfinition d'un attribut ou d'un paramètre impose une vérification de type, qui doit être effectuée par la méthode appelée ou avant l'affectation de l'attribut : pour ce dernier, il faut alors stocker dans la table des méthodes l'identifiant n_1 du type de l'attribut. [Myers, 1995] propose d'affecter un indice différent à chaque signature de

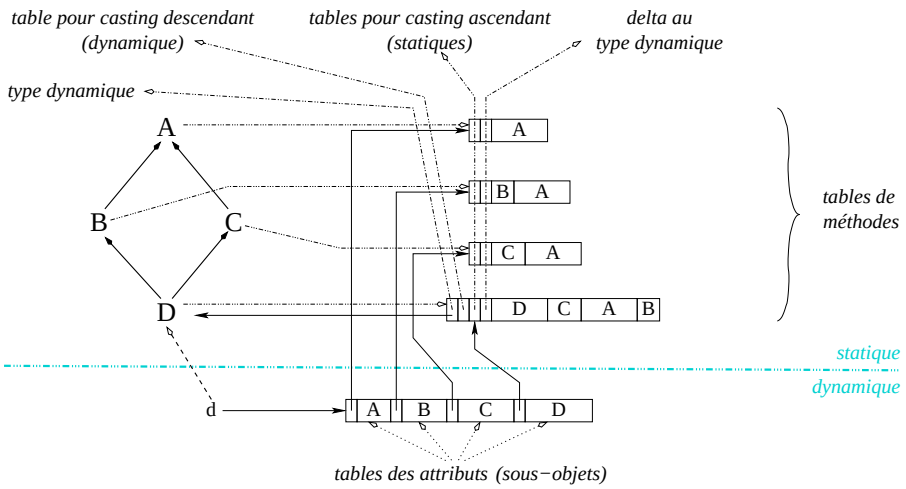

FIG. 2.1 – Tables des attributs et des méthodes en héritage multiple

méthode : le test n'est plus nécessaire qu'en cas de redéfinition entre les méthodes des types statique et dynamique du receveur.

Enfin, les deux invariants rendent l'implémentation *homogène* [Odersky et Wadler, 1997] de la généricité immédiate, à l'unique condition que la généricité soit bornée, c'est-à-dire que les types formels soient contraints à être des sous-types d'un type donné.

L'efficacité temporelle est optimale, puisque tout se fait par une simple indirection dans une table obtenue par une simple indirection dans l'objet. La transparence complète du *casting* ascendant et le faible coût du *casting* descendant, réduit à un simple test numérique, enlève tout surcoût à la redéfinition covariante. D'un point de vue spatial, l'occupation mémoire, tant dynamique que statique, est optimale : la taille totale des tables de méthodes est égale au nombre de couples classe-sélecteur valides, ce qui est l'optimum de compactage des tables [Ducournau, 1997], plus deux entiers par classe pour la vérification dynamique de type. Cette implémentation est la référence qui permet d'étalonner le « surcoût » de l'héritage multiple.

2.2 En héritage multiple

En héritage multiple, l'indice d'une méthode ou d'un attribut ne peut plus être invariant par héritage (invariant 2.2), en tout cas tant que ces indices sont calculés de façon séparée, en cherchant à les minimiser : étant donné deux classes incomparables B et C, qui occupent les mêmes indices, il est toujours possible d'en définir une sous-classe commune D. On se trouve donc en situation de conflit puisque deux attributs ou deux méthodes sont en compétition pour le même indice.

2.2.1 Principe d'implémentation

Si l'on veut que l'envoi de message s'effectue par une indirection dans une table, un pointeur sur un objet n'est plus invariant suivant son type statique. On est donc conduit à relâcher l'invariance de l'indice des attributs et méthodes comme suit :

Invariant 2.3 *Chaque attribut (resp. méthode) a un indice non ambigu et invariant dans le contexte du type qui introduit l'attribut (resp. du type statique qui connaît la méthode), donc indépendamment du type dynamique.*

Mais ces indices ne sont plus invariants par héritage, c'est-à-dire entre deux types statiques liés par une relation de spécialisation, pas plus qu'une référence à l'objet lui-même : la valeur de `self` et de tout pointeur sur un objet dépend de son type statique. Tout se passe comme si les tables des attributs (l'objet) et des méthodes étaient constituées de sous-tables (ou sous-objets), une par super-classe (figure 2.1). L'invariant principal, doublé d'un anti-invariant, est alors le suivant :

Invariant 2.4 *Toute entité de type statique T est liée au sous-objet correspondant à T, muni de sa propre table de méthodes, et deux sous-objets de types statiques différents sont distincts.*

Un sous-objet ne contient que les attributs *introduits* par son type statique et sa table de méthodes contient toutes les méthodes connues par ce type, avec des valeurs (adresses) correspondant aux méthodes héritées par le type dynamique. Pour un type statique, l'ordre des méthodes est *a priori* quelconque, mais il est raisonnable de les regrouper par classe, comme dans la figure. L'invariant 2.4 impose de recalculer la valeur de `self` à chaque envoi de message : lorsque le receveur est une entité de type statique[1] τ_s, et que la méthode sélectionnée a été définie dans la classe v, il faut savoir de combien incrémenter ou décrémenter le receveur pour obtenir, à partir du sous-objet de type τ_s un sous-objet de type v, ce que l'on notera $\Delta_{\tau_s,v}$. La table des méthodes est donc double : elle contient, pour chaque méthode, l'adresse et le décalage. Au total, l'envoi de message se compile donc par la séquence[2] :

```
load [object + #tableOffset], table
load [table + #deltaOffset], delta
load [table + #selectorOffset], method
add object, delta, object
call method
```

Une technique alternative consiste à définir une petite fonction intermédiaire, un *thunk*, qui fait le décalage. Pour chaque méthode, il y a un *thunk* par doublet de classes (τ_s, τ_d). De plus, le *thunk* de décalage 0 (quand la méthode est définie dans τ_s) s'identifie à la méthode elle-même. La séquence d'instructions est la même qu'en héritage simple, mais elle provoque un branchement au code suivant :

```
add object, #delta, object
jump #method
```

2.2.2 Le *casting*

Si le *casting* est inexistant en héritage simple, il est omniprésent dans cette implémentation par sous-objets. Il s'implémente par deux tables supplémentaires, plus un décalage vers le type dynamique (noté $\Delta^{\Downarrow}_{\tau_s} = \Delta_{\tau_s,\tau_d}$).

Casting ascendant et accès aux attributs.

Pour passer du type statique τ_s à un super-type (statique) t, il faut connaître $\Delta_{\tau_s,t}$, qui peut varier suivant le type dynamique τ_d : une table, notée $\Delta^{\uparrow}_{\tau_s}$, est nécessaire dans chaque table de méthodes. L'indice de t dans $\Delta^{\uparrow}_{\tau_s}$ est invariant relativement à τ_d et est donc connu statiquement. Une implémentation de cette table directement dans les tables de méthodes est plus efficace : le *casting* ascendant peut être traité comme les méthodes, comme si chaque classe définissait la méthode de *casting* ascendant vers elle-même. Mais la description par une table supplémentaire est plus claire. La table $\Delta^{\uparrow}_{\tau_s}$ sert aussi à l'accès aux attributs qui sont introduits dans une super-classe de τ_s. Si $\delta(p,u)$ représente la position d'un attribut p par rapport au sous-objet de type u et que δ_p est l'indice de l'attribut p dans le type t_p qui l'introduit (invariant 2.3), on obtient la position de l'attribut par : $\delta(p,\tau_s) = \Delta^{\uparrow}_{\tau_s}(t_p) + \delta_p$. Lorsque $t_p \neq \tau_s$, l'accès sera donc plus compliqué qu'en héritage simple :

```
load [object + #tableOffset], table
load [table + #castOffset], delta
add object, delta, object
load [object + #attributeOffset], attribute
```

Casting descendant.

Pour passer du type statique τ_s à un sous-type (statique) t, il faut à la fois une vérification de type et $\Delta_{\tau_s,t}$. En compilation séparée, un accès direct est impossible : on construit, pour chaque classe τ_d, une

[1] On notera τ_s (resp. τ_d) le type statique (resp. dynamique) d'une entité.

[2] En italiques, le code spécifique à l'héritage multiple, son surcoût. Le pseudo-code qui illustre les différentes techniques est emprunté à [Driesen, 1999].

table, notée $\Delta^{\Uparrow}$, qui associe $\Delta_{\tau_s,t}$ à chaque super-classe t de τ_d. Pour faire un *casting* descendant de τ_s à t, dans un objet de type dynamique τ_d, il faut alors retourner au type dynamique par $\Delta^{\Downarrow}_{\tau_s} = \Delta_{\tau_s,\tau_d}$ et rechercher t dans la table d'association : si on ne le trouve pas, il y a une erreur de type, et sinon on retourne $\Delta_{\tau_d,t}$. On obtient alors le résultat par $\Delta_{\tau_s,t} = \Delta^{\Downarrow}_{\tau_s} + \Delta^{\Uparrow}(t)$. Cette implémentation de la vérification de type est peu efficace. Un codage d'ordre, notablement plus difficile en héritage multiple, ne convient pas puisque le problème n'est plus booléen : les décalages sont nécessaires. La solution naïve serait une matrice d'incidence de la fermeture transitive : elle coûterait N^2 octets, pour N classes.

2.2.3 Redéfinition de types

Si, en héritage simple, la redéfinition de types s'implémentait de façon à peu près transparente, ce n'est plus le cas en héritage multiple. Pour les méthodes, parmi les techniques possibles, nous ne décrirons que celle des *thunks*. La redéfinition du type de retour nécessite un *casting* ascendant : le *thunk* effectuera le décalage sur le receveur avant l'appel et sur la valeur de retour après. L'appel de la méthode par le *thunk* a l'inconvénient de ne plus être terminal (`call` au lieu de `jump`). Pour la redéfinition du type du paramètre, le *thunk* doit effectuer un *casting* descendant entre les types statiques du paramètre dans les méthodes respectives des types statique et dynamique du receveur. Les *thunks* rendent inutile la technique de [Myers, 1995], qui en est une variante.

La redéfinition covariante des attributs est une synthèse des redéfinitions covariantes des types de retour et des paramètres : on choisira une technique qui ne fait pas appel aux *thunks*, pour éviter un appel de méthode supplémentaire, et qui repose sur le stockage du type de l'attribut dans la table des méthodes. En contrepartie, les vérifications doivent être systématiques. L'invariant 2.3, qui s'applique aussi aux attributs, ne peut plus être vérifié et il faut le réinterpréter comme suit :

Invariant 2.5 *L'attribut pointe sur le sous-objet correspondant au type de l'attribut dans le type dynamique de l'objet qui contient l'attribut.*

Tout accès en lecture est suivi d'un *casting* ascendant sur le type statique de l'attribut, de même que tout accès en écriture est précédé d'un *casting* descendant sur le type de l'attribut dans le type dynamique de l'objet, pour assurer la première règle.

2.2.4 La généricité

L'absence d'invariance complique l'implémentation homogène de la généricité, même quand elle est bornée. En effet, dans le cas d'un type paramétré $A\langle T < B\rangle$, instancié par $A\langle C\rangle$, où T est le type formel, B la borne et C un sous-type de B, il est impossible de déduire les indices des méthodes et attributs de C de ceux de B. On rajoute donc aux tables de méthodes de A des tables de conversion entre les indices utilisés pour les accès à T et ceux de C [Day *et al.*, 1995] : l'accès au type formel nécessite alors juste une indirection supplémentaire. Pour les attributs, il faut une indirection supplémentaire, puisque leurs indices sont relatifs à la classe qui les introduit. L'implémentation se complique encore si l'on accepte la covariance pour les types paramétrés : des *castings* ascendants et descendants sont nécessaires en permanence.

2.2.5 Evaluation

Le surcoût temporel de l'héritage multiple est évident :

- une affectation ou un passage de paramètres qui n'est pas à type statique constant exige un décalage ;
- l'efficacité de l'accès aux attributs est significativement réduite puisqu'il faut passer par un décalage stocké dans la table de méthodes ;
- l'appel de méthode nécessite un décalage sur le receveur, sans parler des décalages nécessaires sur les paramètres ;
- le *casting* descendant ne se fait en temps constant qu'au prix de très grandes tables de fermeture transitive ;
- enfin, cette implémentation augmente le coût de la gestion mémoire : le *garbage collector* devra gérer des pointeurs au milieu des objets.

L'analyse et les expérimentations de [Driesen et Hölzle, 1996 ; Driesen, 1999] compliquent grandement l'analyse de l'efficacité : en effet, le surcoût de l'héritage multiple dans l'appel de méthode serait annulé par l'architecture du processeur (*pipe-line*), le décalage se faisant en parallèle ou pendant les latences de chargement. Mais cette analyse est contredite par les expérimentations, qui donnent l'avantage à la technique des *thunks* d'un point de vue temporel, à cause du grand nombre de *thunks* avec un décalage nul. Ces décalages nuls s'expliquent par le fait qu'il s'agit de programmes C++ standard, majoritairement sans le mot-clé `virtual` (cette implémentation n'est pas décrite ici à cause du manque de place et de sa mauvaise sémantique vis-à-vis de l'héritage multiple). Il faut donc déduire de ces expérimentations que les décalages peuvent se paraléliser, mais au détriment d'une autre séquence d'instructions, par exemple sur les paramètres. Le surcoût de l'héritage multiple reste donc significatif.

D'un point de vue spatial, le surcoût dynamique dans chaque objet est égal au nombre de super-classes de la classe de l'objet. Il peut même être exagérément plus grand, dans certaines implémentations qui remplacent les tables de décalages $\Delta^{\uparrow}$ par des pointeurs (`VBPTR`) dans l'objet [Eckel et Gil, 2000]. Pour ce qui est de la mémoire statique, le nombre de tables est quadratique : pour le nombre d'entrées, [Driesen et Hölzle, 1996 ; Driesen, 1999] donne un facteur supérieur à 3, mais avec des programmes avec héritage « non virtuel ». De plus, les décalages doublent quasiment la taille des tables et du code.

Cette implémentation présente un inconvénient majeur. L'héritage simple n'est plus un cas particulier de l'héritage multiple et le surcoût de ce dernier se fait sentir même sur une hiérarchie de classes en héritage simple.

Chapitre 3

La coloration

La coloration de sélecteurs a été proposée dans [Dixon *et al.*, 1989] et expérimentée par [André et Royer, 1992], à la suite de quoi elle a été écartée à cause de son apparente inefficacité. L'heuristique proposée par [Ducournau, 1997] démontre des performances qui justifient pleinement sa prise en considération.

3.1 Principe de la coloration

On peut voir la coloration comme le problème du maintien des deux invariants de l'héritage simple, en cas d'héritage multiple, alors que l'implémentation standard consiste à relâcher ces invariants. Le deuxième invariant peut se préciser ainsi :

Invariant 3.1 *Un attribut (resp. méthode) a une couleur invariante par héritage. Deux attributs (resp. méthodes) de même couleur n'appartiennent pas à la même classe.*

Une numérotation injective suffit à satisfaire l'invariant : on obtient la table gigantesque mais presque vide, qui a fait l'objet de nombreux travaux de compactage [Ducournau, 1997]. Un critère de minimisation est donc nécessaire à la bonne formulation du problème : dans l'approche originelle de [Dixon *et al.*, 1989] et [André et Royer, 1992], ce critère était le nombre maximal de couleurs, ce qui en faisait un problème de coloration de graphes, proprement incalculable de façon directe, même par des techniques approchées. Par ailleurs, ce critère donnait un compactage assez médiocre. Dans [Ducournau, 1997], nous avons apporté deux améliorations à la technique. D'abord, le critère de minimisation est la somme, sur toutes les classes, de la couleur maximale des méthodes (resp. attributs) des classes : on minimise donc la taille totale des tables générées, ce qui donne un compactage très satisfaisant, le taux de trous étant d'à peine quelques dizaines de %.

Ensuite, le schéma d'algorithme est une adaptation du schéma d'algorithme de l'héritage simple au cas de l'héritage multiple : on se sert donc de la hiérarchie d'héritage, ce qui évite d'être confronté directement à une coloration de graphe . Nous apportons ici une nouvelle amélioration de la compacité des tables en reprenant la bidirectionnalité déjà adoptée dans d'autres approches [Myers, 1995 ; Eckel et Gil, 2000], et originellement employée par [Pugh et Weddel, 1990] pour la coloration d'attributs.

De plus, comme le montre la formulation de l'invariant, la coloration peut s'appliquer aussi bien aux méthodes qu'aux attributs : c'était d'ailleurs le but originel de cette heuristique dans le langage YAFOOL [Ducournau, 1991], ainsi que dans [Pugh et Weddel, 1990].

3.2 Coloration de classes

La coloration a encore une troisième cible : les classes, pour l'implémentation de la vérification de types. Comme en héritage simple, le *casting* ascendant n'a plus de raisons d'être et le *casting* descendant se ramène à un test booléen puisqu'il n'y a pas de décalage à retourner, `self` étant invariant. Une technique de codage d'ordre, notablement plus difficile en héritage multiple, serait donc envisageable [Caseau, 1993 ; Queinnec, 1997 ; Habib *et al.*, 1997 ; Vitek *et al.*, 1997 ; Raynaud et Thierry, 2001 ; Thierry, 2001]. Mais la

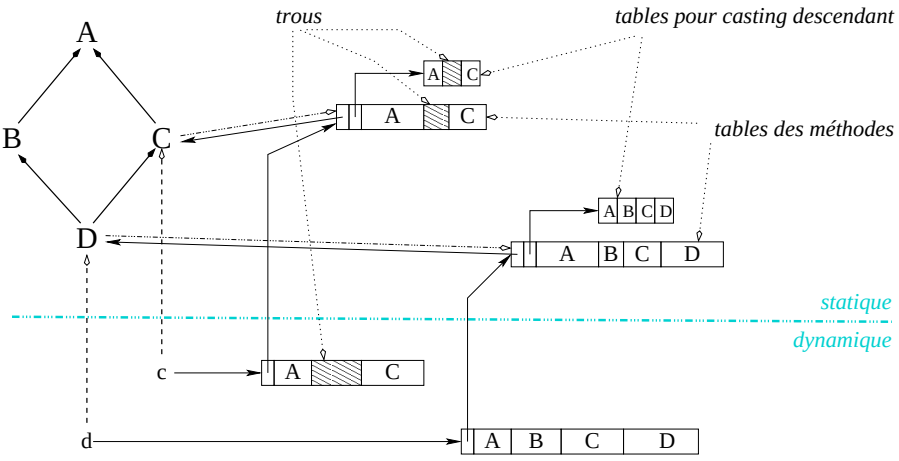

FIG. 3.1 – Exemple de la figure 2.1, avec l'heuristique de coloration monodirectionnelle, appliquée aux classes, aux méthodes et aux attributs

coloration peut aussi faire l'affaire : on colore les classes de façon à ce que deux classes de même indice n'aient pas de sous-classe commune, exactement comme si les classes étaient des méthodes. Pour vérifier que l'objet considéré est une instance de la classe C, il suffit de vérifier, dans la table T_{τ_d} de la classe de l'objet, que $T_{\tau_d}[k(C)] = n(C)$, où k est la couleur des classes et n leur identifiant. Le coût de ce test est du même ordre qu'en héritage simple et, dans les deux cas, inférieur au coût d'un appel de méthode.

Comme pour les tables $\Delta^{\uparrow}$, la table de coloration des classes peut être intégrée dans la table de méthode, ce qui évite une indirection : la seule condition est qu'un identifiant de classe $n(C)$ ne puisse pas être confondu avec une adresse de méthode. Au prix d'un pointeur dynamique de plus, on peut aussi référencer cette table depuis l'objet, ce qui évite un test sur la longueur de la table.

La coloration de classes a été plusieurs fois proposée, sans que le lien avec la coloration de sélecteurs soit établi : [Cohen, 1991] et [Queinnec, 1997] la décrivent pour l'héritage simple où elle est beaucoup moins efficace que la double numérotation présentée plus haut : la couleur s'identifie alors à la profondeur, ou niveau ou rang, de la classe. [Vitek *et al.*, 1997] la généralise à l'héritage multiple, en conservant le critère de minimisation du nombre maximal de couleurs qui est loin d'être optimum. Comme technique de codage d'ordre, la coloration est loin de l'optimal souhaité par les algorithmiciens [Thierry, 2001], mais son coût est marginal, puisqu'elle correspond à la définition d'une méthode supplémentaire par classe. Elle ressort de l'étude de [Vitek *et al.*, 1997] comme une des plus efficaces : sur le plan temporel, il est peu probable que l'on fasse jamais beaucoup mieux. D'un point de vue spatial, la coloration de classes a autant d'entrées occupées que la taille de la fermeture transitive, soit hN, si h est le nombre moyen de super-classes indirectes d'une classe et N le nombre de classes, à comparer aux N^2 d'une table de fermeture transitive. Les statistiques de [Vitek *et al.*, 1997] donnent à h (resp. N/h), des valeurs de l'ordre de 10 à 30 (resp. 50 à 300).

3.3 Schéma d'algorithme

On décrira ici une heuristique qui donne une solution au problème de la coloration, en commençant par la coloration de classes :

```
chaque classe est munie d'une table illimitée
pour chaque classe C, prise dans un certain ordre :
     affecter à C le premier indice libre dans la table de C ;
     propager l'occupation de cet indice dans les super-classes
          des sous-classes de C ;
finpour
```

Tout repose sur le *certain ordre* : la correction est toujours assurée, mais seule une extension linéaire descendante, de la super-classe à la sous-classe, permet une minimisation. Le choix de l'extension linéaire n'est pas sans effet : mais tout dépend alors de savoir s'il s'agit d'une coloration de classes, de méthodes ou d'attributs. Pour la coloration d'attributs (resp. méthodes), il est possible de partir de la coloration des classes, en plaçant les attributs (resp. méthodes) introduits par une classe par blocs contigus, dans l'ordre des couleurs des classes : la difficulté est que ces blocs ne seraient pas de même taille, d'où des risques de trous. Il est donc plus facile, et pas beaucoup plus lent, de refaire une coloration spécifique pour les attributs (resp. méthodes) :

```
 chaque classe est munie d'une table illimitée
 pour chaque classe C, prise dans un certain ordre :
      pour chaque attribut (resp. méthode) m introduit par C,
1          affecter à m le premier indice libre dans la table de C ;
      finpour
2     propager l'occupation de ces indices dans les super-classes
3                    des sous-classes de C ;
 finpour
```

Si le critère d'optimisation est le même, la coloration des attributs et des méthodes peut se faire en même temps.

En héritage simple et typage statique, l'algorithme de l'implémentation standard est identique, sans propagation, car le premier indice libre est le successeur de la couleur maximum de la super-classe, d'où l'absence de trous.

3.3.1 Coloration bidirectionnelle

La coloration décrite jusqu'ici sous-entendait des couleurs ou indices positifs, les tables s'étendant de l'indice 0 à l'indice maximal occupé. L'intuition et diverses expérimentations [Pugh et Weddel, 1990 ; Myers, 1995 ; Eckel et Gil, 2000] montrent une plus grande compacité avec des tables aux indices aussi bien négatifs que positifs. Dans une coloration bidirectionnelle, la taille de chaque table est la différence entre les indices maximaux et minimaux occupés. Les algorithmes sont modifiés dans la définition *du premier indice libre* : il faut considérer les premiers indices libres, positif et négatif, et choisir celui qui cause le moins de trous, le positif en cas d'égalité. La conjecture suivante est vraisemblable :

Conjecture 3.1 *A extension linéaire identique, la bidirectionnalité réduit au moins de moitié le nombre de trous produits par la coloration.*

On ne peut espérer mieux : un peigne de degré d — une classe avec d super-classes directes incomparables — aura de l'ordre de $d^2/2$ trous en coloration monodirectionnelle, et $d^2/4$ en coloration bidirectionnelle. Dans la figure 3.1, la bidirectionnalité enlèverait tous les trous, en affectant à C l'indice -1.

3.3.2 Coloration en typage dynamique

Le typage dynamique introduit deux complications par rapport au typage statique. Deux méthodes de même nom peuvent être introduites dans plusieurs classes incomparables, sans qu'il soit possible de les distinguer à cause de l'absence d'annotation de type, (« surcharge en typage dynamique ») : cela oblige, dans le schéma d'algorithme, à vérifier la liberté d'un indice (ligne 1) et à propager une affectation dans toutes les classes de définition, pas seulement dans C (ligne 3). La propagation ne peut pas se faire globalement, pour toutes les méthodes d'une classe, mais doit se faire méthode par méthode (lignes 2-3 à déplacer dans le `pour-finpour` précédent). L'absence de vérification statique des types impose aussi une vérification dynamique, à chaque appel de méthode, en passant le sélecteur en paramètre pour que la méthode appelée vérifie que c'est bien le bon : il y a sinon une erreur `message inconnu` [Ducournau, 2001a].

3.4 Implémentation des attributs par coloration des classes

A ce stade, la coloration a un défaut majeur : l'existence de trous dans les tables. Ces trous ne sont pas trop gênants pour les tables de méthodes, où ils seront plus que compensés par la réduction du nombre

de tables et la disparition des décalages. En revanche, les trous dans les instances peuvent provoquer un surcoût sans commune mesure avec la perte des pointeurs sur les tables de méthodes des super-classes : ce surcoût peut être important dans chaque instance et multiplié par un nombre important d'instances. Ils constituent donc un handicap sérieux : leur minimisation est *a priori* inenvisageable, le problème étant au moins NP-difficile. Par ailleurs, l'optimal doit pouvoir conserver un surcoût très significatif.

La solution la plus raisonnable est donc sans doute de ne pas se servir de la coloration pour les attributs, mais de coupler coloration de méthodes et de classes avec une implémentation des attributs par sous-objets, un peu comme dans l'implémentation standard : la différence principale est que ces sous-objets, se partagent tous la même table de méthodes et que les seuls décalages nécessaires sont entre le début de l'objet et un sous-objet. Pour accéder statiquement à ces décalages, la coloration de classes donne une solution simple et élégante : à la couleur est associée, non seulement l'identifiant de la classe mais aussi le décalage pour accéder aux attributs introduits par la classe. La séquence d'accès est alors la même qu'en héritage multiple, dans le cas général avec *casting* ascendant ($\tau_s \neq t_p$) : on y perd donc un peu dans le cas où l'attribut a été défini dans le type statique ($\tau_s = t_p$). Pour remédier à cette perte, on peut reprendre à [Myers, 1995] l'idée suivante. Pour les accès à un attribut sur un receveur autre que `self`, supposés rares si une bonne encapsulation est réalisée, on procède comme ci-dessus. En revanche, pour les accès à `self`, le compilateur génère deux versions de toutes les méthodes, suivant que les indices des attributs sont supposés invariants dans les sous-classes ou pas. Le choix de la bonne méthode s'effecturera alors d'après une analyse globale : typiquement, si la classe n'est spécialisée qu'en héritage simple, la version la plus efficace sera utilisée.

3.5 Comparaison entre coloration et implémentation standard

Les deux techniques sont plus ou moins à égalité pour l'efficacité de l'accès aux attributs. Avec une compilation simple, il y a un léger surcoût pour la coloration dans le cas d'un attribut introduit dans le type statique, alors qu'avec la double compilation, il y a un avantage pour la coloration lorsque le receveur est `self` et que les indices peuvent être globalement invariants. En revanche, la coloration l'emporte nettement sur tous les autres points, où elle est exactement équivalente à l'héritage simple :

- pour l'appel de méthode, où les expérimentations de [Driesen et Hölzle, 1996 ; Driesen, 1999] montrent la réalité du surcoût des décalages ;
- pour le *casting* ascendant, qui disparaît, alors qu'il est omniprésent dans l'implémentation standard ;
- pour le *casting* descendant, en temps constant, et d'un coût analogue au *casting* ascendant de l'implémentation standard, avec un coût mémoire raisonnable ;
- pour la redéfinition covariante, qui ne coûte rien pour le type de retour ou pour la lecture des attributs, et qui se réduit à un *casting* descendant dans les autres cas : *a contrario*, la covariance est très lourde dans l'implémentation standard ;
- pour la généricité bornée, qui s'implémente comme en héritage simple ;
- pour la mémoire dynamique, qui ne présente aucun surcoût et qui se contente d'un *garbage collector* assez simple : de plus, la coloration se prête bien, grâce à l'invariant 2.1, à une implémentation par *wrapper* qui permet, au prix d'une indirection supplémentaire pour les attributs et d'un pointeur dynamique supplémentaire par objet, de compacter simplement le tas ou d'implémenter des fonctionnalités de migration ou de classification d'instances.

Quant à la mémoire statique, les tables de méthodes de la coloration sont celles de l'héritage simple, avec un facteur légèrement supérieur à 1 et, dans toutes les expérimentations, largement inférieur à 2, alors que l'implémentation standard a un nombre quadratique de tables, entraînant un facteur supérieur à 3 dans la cadre d'un usage très limité [Driesen, 1999], auxquels s'ajoutent les décalages ou les *thunks*. Enfin, les décalages qui parsèment le programme augmentent significativement la taille du code dans l'implémentation standard. Dans tous les cas, une expérimentation serait intéressante pour comparer les coûts statiques des deux implémentations. Une chose est sûre : avec la double compilation, *l'application de la coloration à des classes en héritage simple avec une encapsulation des attributs à la* SMALLTALK *donne exactement la même implémentation que l'implémentation standard de l'héritage simple*.

3.6 Compatibilité avec une compilation séparée

La coloration est une technique globale, non incrémentale. Elle n'est donc pas calculable par une compilation séparée, mais rien n'empêche que le compilateur ne génère du code compatible avec une future coloration. En effet, la coloration n'a pas besoin de connaître le code des classes : leur schéma, tel qu'il est déjà accessible au compilateur dans les fichiers d'en-tête, suffit. Le nom des classes, des propriétés, leurs types, les relations de spécialisation et les conventions de nommage du compilateur pour désigner les diverses constantes qui ne sont pas encore calculées, suffisent. Dans le pseudo-code illustrant cet article, ce sont, pour la plupart, les variables préfixées par # : les valeurs exactes peuvent être calculée à l'édition de liens, avant d'être résolues par elle. La double compilation ne pose pas de problème particulier, le choix de la bonne version à inclure dans l'exécutable dépendant de critères globaux simples. Par ailleurs, le surcoût de l'édition de liens devrait être partiellement compensé par la réduction des recompilations nécessaires. En revanche, l'incompatibilité avec une édition de liens incrémentale semble définitive. La technique n'est pas incrémentale et tout s'oppose au recalcul de la coloration en cas d'ajout de classes : d'une part, son coût algorithmique est sans doute trop important pour que cela passe inaperçu, d'autre part, cela nécessiterait de remplacer d'innombrables constantes du code par des variables, ce qui serait beaucoup moins efficace.

Chapitre 4

Comparaisons, évolutions et perspectives

L'opposition entre héritages simple et multiple est réductrice, vu le nombre de langages récents qui sont en héritage simple et sous-typage multiple, comme JAVA, THETA [Myers, 1995 ; Day *et al.*, 1995] tous les langages destinés à la plateforme .NET de MicroSoft comme EIFFEL# [Simon *et al.*, 2000]. Ces langages ont des classes en héritage simple et des interfaces en sous-typage multiple. [Ducournau, 2001a] décrit diverses implémentations possibles dans ce contexte intermédiaire. Ce que nous avons appelé « héritage simple » doit alors être compris comme du « sous-typage simple ». En cas d'héritage simple et de sous-typage multiple, la coloration d'attributs ne produit pas de trous et donne donc une implémentation aussi efficace que l'héritage simple. Ce n'est toutefois pas applicable à JAVA ou EIFFEL# à cause de leur chargement dynamique.

L'implémentation standard de l'héritage multiple en compilation séparée n'est pas la seule possible. [Myers, 1995] en propose une simplification dans le cas du sous-typage multiple, par des tables bidirectionnelles. Une application de leur technique à l'héritage multiple repose sur le décalage des attributs et la double compilation que nous avons repris. Elle réduit le nombre de tables et de décalages, comme l'héritage multiple « non virtuel », mais sans ses inconvénients sémantiques.

Dans les recherches récentes sur l'implémentation des objets, deux thèmes importants ont émergé : la prise en compte de l'architecture des processeurs et la compilation globale. SMALL EIFFEL est typique de cette double évolution [Zendra *et al.*, 1997]. La compilation globale permet une analyse globale des types qui identifie le code mort (jamais appelé), les appels monomorphes (un seul type possible) et, plus généralement, pour toute expression et toute exécution, l'ensemble des types dynamiques possibles. Une telle analyse, malgré son coût [Gil et Itai, 1998], réduit considérablement l'incertitude de la liaison tardive. Les concepteurs de SMALL EIFFEL considèrent qu'après cette analyse, le nombre de types est suffisamment réduit pour qu'une implémentation par des tables soit inutile : l'appel de méthode est implémenté par un arbre de décision binaire (*dispatch tree*), qui énumère tous les types possibles et fait, pour chacun, un appel direct à la méthode correspondante. C'est là qu'intervient l'architecture des processeurs : dans les processeurs actuels, le branchement indirect (typiquement l'appel de méthode par une table) a une latence pouvant atteindre 10 à 15 cycles, alors que le branchement conditionnel est accéléré par des tables de prédiction. Les arbres de décisions semblent donc rendre obsolètes les tables. Mais les processeurs commencent à avoir des capacités de prédiction de branchement indirect aussi efficaces que pour les branchements conditionnels [Driesen, 1999] : il est donc probable que la comparaison se rééquilibre dans un proche avenir. Par ailleurs, la technique des arbres de décision est difficilement compatible avec la compilation séparée.

Chapitre 5

Conclusion et perspectives

L'approche que nous proposons est basée sur plusieurs techniques complémentaires, étudiées par divers auteurs, souvent de façon indépendante. Le principe de la coloration, qui rend l'implémentation globalement invariante vis-à-vis des types statiques, a été introduit, étudié, implémenté et expérimenté par [Dixon *et al.*, 1989 ; Pugh et Weddel, 1990 ; Ducournau, 1991 ; André et Royer, 1992 ; Ducournau *et al.*, 1992 ; Ducournau, 1997]. La coloration de classes offre une implémentation efficace du test de sous-type et du décalage pour les accès aux attributs : elle a été décrite dans le cadre de l'héritage simple par [Cohen, 1991 ; Queinnec, 1997], puis généralisée à l'héritage multiple et étudiée expérimentalement par [Vitek *et al.*, 1997]. La compacité des tables est améliorée si on prend leur taille comme critère de minimisation [Pugh et Weddel, 1990 ; Ducournau, 1991 ; Ducournau, 1997] au lieu du nombre de couleurs [Dixon *et al.*, 1989 ; André et Royer, 1992 ; Vitek *et al.*, 1997]. La bidirectionnalité [Pugh et Weddel, 1990], reprise par [Myers, 1995 ; Eckel et Gil, 2000], l'améliore encore. La coloration d'attributs, due à [Pugh et Weddel, 1990 ; Ducournau, 1991], est une solution pour le sous-typage multiple mais on lui préfèrera, pour l'héritage multiple, le décalage des attributs proposé par [Myers, 1995], complété par le même auteur par une double compilation pour éviter des décalages sur `self`. Enfin, l'indiçage des signatures de méthodes, au lieu des sélecteurs, permet une implémentation efficace de la redéfinition covariante [Myers, 1995]. Dans cet article, notre propre contribution se résume pour l'essentiel à la redécouverte de ces divers résultats, et à leur assemblage cohérent, en en prenant le meilleur, dans le cadre du typage statique et celui, plus original, de la compilation séparée.

L'implémentation des objets offre de très nombreuses perspectives, parmi lesquelles trois pistes majeures nous semblent peu explorées. D'autres techniques globales peuvent être mises en œuvre à l'édition de liens : l'analyse globale nécessiterait juste un analyse locale intra-classe, et une analyse inter-classe qui peut s'effectuer à partir de la description des appels de chaque classe, description qui peut être archivée dans un fichier d'en-tête. L'analyse globale esquissée ci-dessus permettrait d'éliminer le code mort, mais pas de traiter les appels monomorphes de façon spécifique : une compilation séparée dans une machine virtuelle qui abstrait tous les cas différents d'accès à un objet permettrait de résoudre ce problème, comme elle éviterait la double compilation de [Myers, 1995]. L'unité de compilation des langages à objets est la classe : si c'est bien une unité conceptuelle, elle n'a certainement pas les qualités que l'on attend d'un module (interaction interne forte, externe faible) ; de véritables modules autoriseraient une optimisation significative [Szypersky, 1992]. Trois axes complémentaires donc : mise en œuvre séparée des techniques globales, machines virtuelles orientées vers les traits spécifiques à optimiser et compilation modulaire.

Deuxième partie

Coloration de classes

Chapitre 6

Définitions et propriétés

6.1 Définitions

6.1.1 Cœur et couronne

On notera $H = (X, E)$ une hiérarchie de classes, X étant l'ensemble de classes et E l'ensemble des arcs de spécialisation entre une classe et une super-classe directe. Il s'agit du graphe de couverture de la relation d'ordre sous-jacente : les éventuels arcs de transitivité sont censés avoir été enlevés au préalable[1]. On notera $H^* = (X, E^*)$ la fermeture transitive et réflexive de ce graphe, qui est un ordre partiel. On utilisera aussi la notation $x \preceq y$ pour $(x, y) \in E^*$.

Définition 6.1 *Le* cœur *de la hiérarchie $H = (X, E)$ est le sous-graphe $Core(H) = (X', E')$, constitué par sa restriction aux classes qui ont plus d'une super-classe directe, et à leur super-classes indirectes :*
$X' = \{x \mid \exists y, z, z' : y \preceq x \ \ \& \ \ (y, z), (y, z') \in E \ \ \& \ \ z \neq z'\}$

Définition 6.2 *La* couronne *de la hiérarchie $H = (X, E)$ est le sous-graphe $Crown(H) = (X'', E'')$, constitué par sa restriction aux classes qui sont en héritage simple (une seule super-classe directe) et dont les sous-classes sont en héritage simple :*
$X'' = \{x \mid \forall y, z, z' : y \preceq x, (y, z) \ \ \& \ \ (y, z') \in E \Rightarrow z = z'\}$

Définition 6.3 *La* frontière *est constituée par les classes minimales du cœur :*
$X''' = \{x \in X' \mid y \prec x \Rightarrow y \in X''\}$.

Définition 6.4 *L'*attache *d'une classe $C \in X''$ (de la couronne) est la classe $att(C) \in X'$ (du cœur), telle que $C \prec att(C)$ et $C \preceq C' \prec att(C) \Rightarrow C' \in X''$.*

Définition 6.5 *Les* attaches *de la couronne sont les classes du cœur dont une sous-classe directe appartient à la couronne :*
$Att(X) = \{att(x) \mid x \in X''\}$.

Le cœur et la couronne forment une partition de l'ensemble de classes X. La couronne constitue une forêt (figure 6.1). Elle est rattachée au cœur par les super-classes directes des racines de cette forêt, les *attaches* : la couronne augmentée des attaches constitue toujours une forêt.

Le cas de l'héritage simple et du sous-typage multiple JAVA et quelques autres langages comme C# ou EIFFEL# ont la particularité d'avoir une hiérarchie de classes en héritage simple, et une hiérarchie d'interfaces en héritage multiple, ou de pratiquer l'héritage simple des classes et le sous-typage multiple.

Un parallèle rapide pourrait faire croire que les interfaces de JAVA forment le cœur de la hiérarchie. Il n'en est rien. Si l'on fait l'hypothèse que toutes les interfaces servent à quelque-chose, c'est-à-dire qu'elles sont implémentées par au moins une classe, la seule chose que l'on puisse dire est que

[1] Conceptuellement, un arc de transitivité n'apporte rien à une relation transitive comme la spécialisation. Pourtant, certains langages leur font jouer un rôle, sur la base d'une sémantique discutable : en C++, avec l'héritage « non virtuel », ou en EIFFEL, avec l'héritage répété qui permet d'hériter plusieurs fois de la même classe. Le problème se rencontre aussi dans les langages comme CLOS où les arcs de transitivité peuvent avoir un effet non désiré sur les algorithmes de linéarisations [Ducournau *et al.*, 1994].

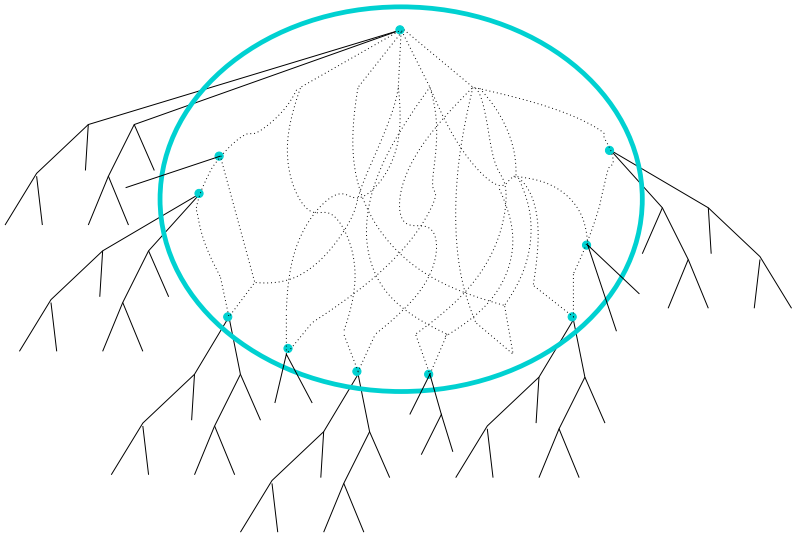

FIG. 6.1 – Le cœur d'une hiérarchie, dans l'ellipse, la couronne et ses attaches

– les interfaces sont dans le cœur,
– par contraposition, la couronne et ses attaches sont formées de classes, de même que la frontière.

Il n'y a donc pas de raison que la structure particulière de la hiérarchie JAVA est une influence sur la coloration, à l'exception de la coloration d'attributs.

6.1.2 Graphe de conflit

Définition 6.6 *Le* graphe de conflit *de la coloration des classes d'une hiérarchie $H = (X, E)$ est le graphe $(X, \hat{E})$ dont l'ensemble de sommets X est l'ensemble de classes à colorer, et la relation $\hat{E}$ est la relation d'incomparabilité de la spécialisation, restreinte aux couples de classes ayant une sous-classe commune :*

$$(x, y) \in \hat{E} \Leftrightarrow x \not\preceq y \ \ \& \ \ y \not\preceq x \ \ \& \ \ \exists z : z \prec x \ \ \& \ \ z \prec y$$

La coloration de classes est exactement le problème de coloration du graphe constitué par l'union du graphe de conflit et du graphe de comparabilité de la relation de spécialisation.

Proposition 6.1 *Les arêtes du graphe de conflit sont incidentes aux sommets du cœur qui n'appartiennent pas à la frontière : $\hat{E} \subseteq (X' - X''')^2$.*

Dit autrement, la couronne et la frontière sont des sommets isolés du graphe de conflit.

6.1.3 Tables de couleurs

Par rapport à la coloration de graphes, la coloration de classes — ainsi que de méthodes ou d'attributs — a ceci de particulier qu'elle est supportée par un ordre partiel et que chaque sommet (classe) hérite des couleurs de ses successeurs (super-classes).

Définition 6.7 *Etant donné une coloration de classes, on note $k(C)$ la couleur d'une classe : la* table de couleurs *d'une classe est l'ensemble des couleurs de la classe et de ses super-classes : $T_C = \{k(C') \mid C \preceq C'\}$*

Les couleurs minimum et maximum d'une classe sont les minimum et maximum de sa table de couleur : $k_{\min}(C) = \min(T_C)$ et $k_{\max}(C) = \max(T_C)$

La taille *d'une table de couleurs est la taille de l'intervalle déterminé par les couleurs maximum et minimum : $size(T_C) = k_{\max}(C) - k_{\min}(C) + 1$.*

Comme la coloration de graphes, la coloration de classes, méthodes ou attributs conduit à un problème de minimisation. Mais au lieu de chercher à minimiser le nombre de couleurs utilisées, c'est-à-dire $\max_C(k_{\max}(C))$, en coloration unidirectionnelle, on cherchera à minimiser la taille des tables de couleurs, c'est-à-dire $\Sigma_C(k_{\max}(C) - k_{\min}(C) + 1)$. En coloration unidirectionnelle, si l'on implémente ces tables par une matrice, dont les tables de couleurs seraient les lignes, cela revient au même. Mais si l'on implémente les tables par des vecteurs, un pour chaque table de couleurs, le problème d'optimisation change complètement. Il n'est même pas sûr que le problème de décision associé soit toujours NP-complet.

Le problème peut donc se formuler, en première instance, comme la minimisation de la taille des tables, c'est-à-dire du nombre d'entrées dans les tables. En second lieu, il faut prendre en compte le contenu de ces entrées, c'est-à-dire le codage des identifiants des classes, ce qui conduit à un nouveau problème d'optimisation. Nous en resterons, pour l'instant, au premier problème d'optimisation, qui consiste à minimiser le nombre d'entrées des tables.

Les couleurs de chaque classe et de ses super-classes occupent au total $\preceq$ entrées, la taille de la fermeture transitive et réflexive de la relation de spécialisation. La minimisation des tables revient à minimiser le nombre d'entrées inoccupées, les *trous*.

Existence d'un plus grand élément Ces définitions posent un petit problème dans le cas où $k_{\min}$ (resp. $k_{\max}$) est strictement positif (resp. négatif), c'est-à-dire lorsque 0 n'appartient pas à l'intervalle $[k_{\min}, k_{\max}]$. Lorsque la racine a un plus grand élément (une racine, unique élément maximal), toute heuristique cherchant à minimiser la taille des tables lui affectera la valeur 0. Si ce n'est pas le cas, il y aura des classes pour lesquelles 0 n'appartient pas à l'intervalle $[k_{\min}, k_{\max}]$. Par ailleurs, même s'il existe un plus grand élément, il est possible de ne pas le faire figurer dans les tables si les tests de type sont toujours statiques (cf. section 8.4.1). Or, si l'on souhaite éviter les tests de débordement de tables (cf. 6.3.3), il est préférable de considérer que 0 appartient toujorus à cet intervalle, ce que l'on peut obtenir, soit en imposant une racine, soit en modifiant la définition de $k_{\min}$ et $k_{\max}$. Toutes nos expérimentations font cette hypothèse, en rajoutant une racine en cas de besoin.

6.2 Propriétés et conjectures

6.2.1 Colorations parfaites

Les résultats qui suivent trouvent, pour partie, leur origine dans [Pugh et Weddel, 1990], où ils s'appliquent à la coloration d'attributs. Ils sont présentés ici de façon très simplifiée grâce à la coloration de classes. La première définition vaut aussi bien pour une coloration de classes, de méthodes ou d'attributs, qu'elle soit uni- ou bi-directionnelle.

Définition 6.8 *Une coloration est dite* parfaite *lorsqu'elle est sans trous : tous les indices sont occupés de la couleur minimum à la couleur maximum de chaque classe : pour tout C, $T_C = [k_{\min}(C), k_{\max}(C)]$.*

La première proposition est élémentaire.

Proposition 6.2 (Coloration unidirectionnelle parfaite) *Il est équivalent de dire :*

1. *il existe une coloration unidirectionnelle parfaite ;*
2. *le graphe de conflits est vide ;*
3. *la hiérarchie est en héritage simple ;*
4. *le schéma d'algorithme (cf. page 30) produit une coloration unidirectionnelle parfaite.*

La généralisation à la bidirectionnalité est déjà plus compliquée.

Proposition 6.3 (Coloration bidirectionnelle parfaite) *Il est équivalent de dire :*

1. *il existe une coloration bidirectionnelle parfaite ;*
2. *le graphe de conflits est biparti ;*
3. *la hiérarchie de classes peut être partitionnée en 2 arborescences ;*

4. *le schéma d'algorithme (cf. page 30) produit une coloration bidirectionnelle parfaite, lorsque l'extension linéaire choisit d'abord des classes dont au moins une classe en conflit a déjà été colorée.*

$1 \Rightarrow 2$. Si deux classes sont en conflit dans une coloration bidirectionnelle parfaite, leurs indices sont de signes opposés : il ne peut donc pas y avoir de cycles de longueur impaire.
$2 \Rightarrow 4$. Pour faire un trou de la coloration, il faut une classe qui soit en conflit avec une classe positive et une classe négative. Or la spécification de l'extension linéaire fait que la première classe d'une composante connexe de conflits va déterminer le signe de toutes les autres : on vérifie aisément que si une classe maximale (éligible pour l'extension linéaire) appartient à la même composante connexe qu'une classe déjà prise, alors il en existe une éligible, de la même composante connexe et en conflit avec une classe déjà prise. L'absence de cycle impair fait que les classes de même signe ne sont pas en conflit.
$1 \Rightarrow 3$. La partition est constituée par les classes de couleur respectivement positive et négative : leurs colorations unidirectionnelles respectives étant parfaites, ce sont donc des arborescences. La réciproque est immédiate.
[Pugh et Weddel, 1990] donne une démonstration de $2 \Rightarrow 1$, pour la coloration d'attributs.

Notons que la précision de l'extension linéaire est nécessaire : soit une chaîne de conflit de longueur 3, si l'extension linéaire commence par prendre les deux extrémités, il n'y aura pas de solution parfaite pour le sommet central.

6.2.2 Réfutation de quelques conjectures

Cette spécification de l'extension linéaire suffit pour obtenir une coloration parfaite lorsqu'elle existe, mais elle n'assure aucun critère d'optimalité dans le cas contraire. Deux conjectures paraissent pourtant vraisemblables, qui sont trivialement vérifiées lorsque la coloration bidirectionnelle est parfaite :

Conjecture 6.1 *Dans le schéma d'algorithme précédent, à extension linéaire identique, la coloration unidirectionnelle produit au moins deux fois plus de trous que la coloration bidirectionnelle.*

On ne peut espérer mieux : un peigne de degré d — une classe avec d super-classes directes incomparables — aura de l'ordre de $d^2/2$ trous en coloration unidirectionnelle, et $d^2/4$ en coloration bidirectionnelle. Le peigne constitue le plus mauvais cas pour la coloration, puisque le taux de trous est de l'ordre de $d/2$: il serait intéressant que le plus mauvais cas pour la coloration soit aussi le plus mauvais cas pour le gain apporté par la bidirectionnalité.

Conjecture 6.2 *Dans le schéma d'algorithme précédent, toute coloration bidirectionnelle ne produit pas plus de trous qu'une coloration unidirectionnelle.*

Autrement dit, la pire coloration bidirectionnelle resterait au moins aussi bonne que la meilleure unidirectionnelle.

Malheureusement, la même expérimentation infirme les deux conjectures : la figure 6.3 donne les résultats de 4 extensions linéaires sur les colorations uni- et bi-directionnelles. Les 4 extensions linéaires sont obtenues en faisant varier la comparaison portant sur le nombre de conflits non encore colorés de la classe maximale considérée. Dans tous les cas, le schéma d'algorithme respecte la priorité donnée aux classes maximales en conflit avec des classes déjà colorées. On constate ainsi que la coloration bidirectionnelle, si elle peut être « infiniment » meilleure lorsqu'elle est parfaite, peut aussi n'apporter qu'une faible amélioration dans certains cas.

Si ces conjectures sont fausses, on pourra chercher à les reformuler, soit en affaiblissant la proposition, par exemple en précisant plus l'algorithme, soit en quantifiant sur toutes les colorations. Par exemple :

Conjecture 6.3 *La meilleure coloration unidirectionnelle produit au moins deux fois plus de trous que la meilleure coloration bidirectionnelle.*

ou encore,

Conjecture 6.4 *A extension linéaire constante, la coloration bidirectionnelle est toujours meilleure que la coloration unidirectionnelle.*

que l'on pourrait renforcer, en voulant que cette amélioration soit uniforme :

Conjecture 6.5 *A extension linéaire constante, chaque classe a moins de trous en coloration bidirectionnelle qu'en coloration unidirectionnelle.*

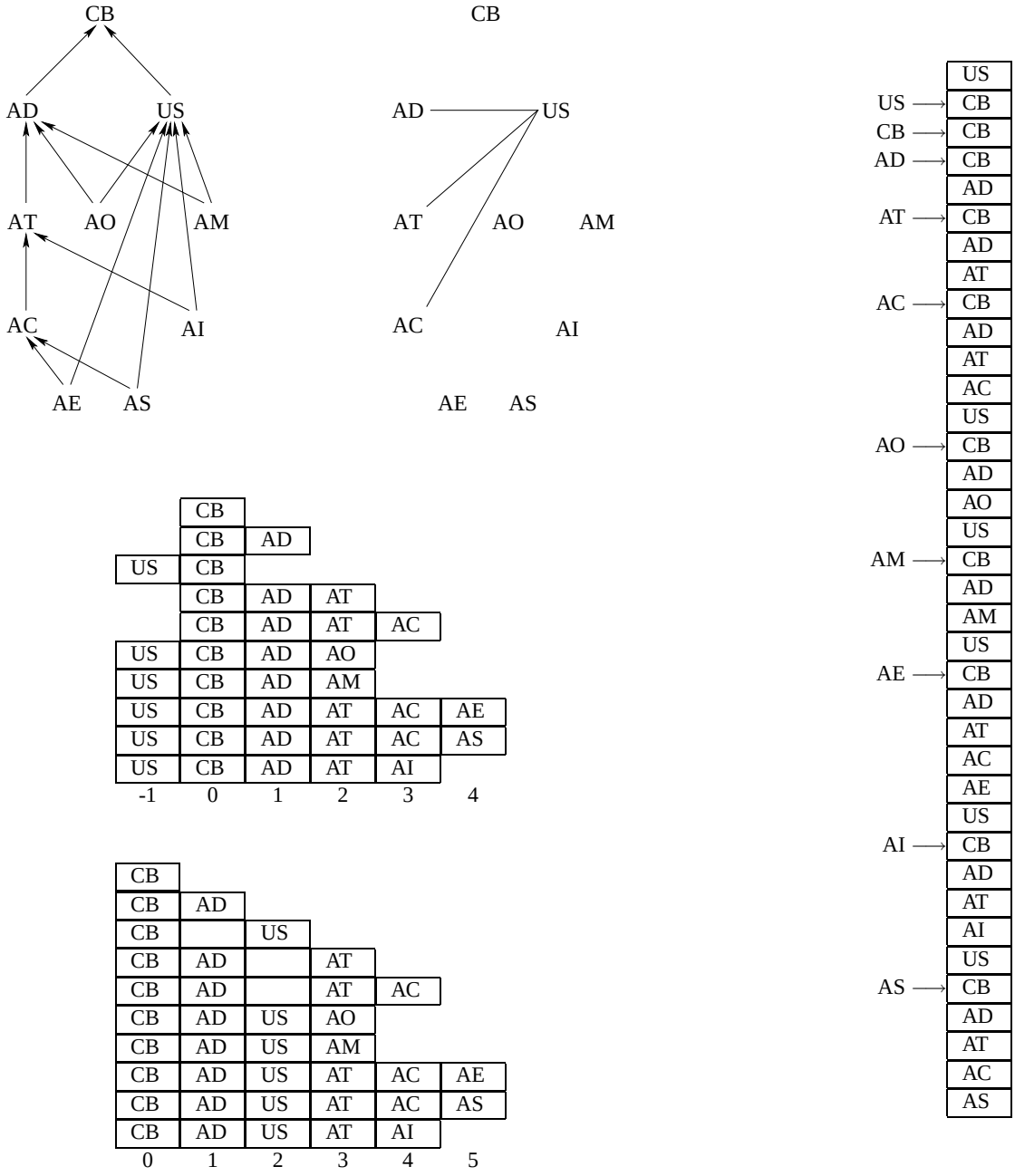

FIG. 6.2 – Coloration de la hiérarchie IDL : en haut, le cœur de la hiérarchie et son graphe de conflit, à gauche les colorations unidirectionnelle et bidirectionnelle, à droite l'implémentation de la coloration bidirectionnelle, dans une seule table évitant le test de longueur de table.

Système	<	<=	>	>=	min
Flavors	109	109	80	85	80
Java-b	125	115	95	93	93
Geode	10866	10833	12685	12678	10833
EIFFEL 2	1613	1621	1547	1529	1529
Self	650	650	1351	1379	650
Unidraw	12	12	12	12	12
total	13478	13443	15844	15855	

Système	<	<=	>	>=	max
Flavors	34	34	18	41	41
Java-b	32	19	15	18	32
Geode	7263	7830	11195	10784	11195
EIFFEL 2	1133	1185	1075	1043	1185
Self	426	414	555	609	609
Unidraw	1	1	3	3	3
total	8890	9500	12840	12500	

FIG. 6.3 – Statistiques de coloration de classes (nombre de trous) suivant l'extension linéaire, en coloration unidirectionnelle (haut) et bidirectionnelle (bas).

Or, comme le montre la figure 8.5, cette dernière conjecture est fausse (le maximum du nombre de couleurs en coloration bidirectionnelle est largement supérieur au minimum de la coloration unidirectionnelle), et elle entraîne la réfutation de la conjecture précédente de façon très simple. Soit la classe qui a plus de trous en coloration bidirectionnelle : il suffit de lui rattacher suffisamment de sous-classes directes en héritage simple, donc dans la couronne, pour multiplier ce trou supplémentaire et dépasser le nombre de trous de la coloration unidirectionnelle.

En revanche, la conjecture 6.3 reste plausible : la figure 8.6 montre que la meilleure coloration bidirectionnelle a toujours près de deux fois moins de trous que la meilleure coloration unidirectionnelle. Les quelques cas où le seuil est dépassé ne sont pas des contre-exemples puisque rien ne prouve que la meilleure coloration ait été atteinte.

D'après un résultat classique [Golumbic, 1980], la coloration est polynomiale, voire linéaire, pour les classes de graphes de comparabilité ou de co-comparabilité. Or le graphe de conflit est une partie du graphe de co-comparabilité, et la coloration de classes peut se traduire par une coloration de l'union du graphe de comparabilité et du graphe de conflit : il serait donc tentant de lui étendre ce résultat. Il est clair que le graphe de conflit, bien que partie du graphe de co-comparabilité, ne fait partie d'aucune classe privilégiée : n'importe quel graphe $G = (V, F)$ peut être le graphe de conflit d'un ordre partiel, en l'occurrence le graphe $G' = (V \cup F, F')$, avec $F' = \{((x, y), x) \mid (x, y) \in F\}$. Cela ne suffit pas pour réfuter cette hypothèse puisque le graphe à colorer est l'union du graphe de conflit et du graphe de comparabilité.

6.3 Variations autour du problème de la coloration de classes

Plusieurs variations sont possibles autour de la coloration : certaines concernent toutes les colorations, d'autres ne s'appliquent qu'à la coloration de classes.

6.3.1 Généralisation : coloration n-dimensionnelle

La coloration bidirectionnelle donne une caractérisation des hiérarchies de classes dont la coloration est parfaite. Une coloration bidirectionnelle permet de décomposer l'ordre d'origine en deux sous-ordres — les classes de couleur positive ou négative. Chaque classe C peut donc se représenter comme une paire (C^+, C^-), où $C \in \{C^+, C^-\}$, l'autre classe étant la super-classe de couleur opposée maximale. Deux classes C et D sont alors sous-classes l'une de l'autre ssi $C^+ \preceq D^+$ et $C^- \preceq D^-$. Si la coloration unidirectionnelle des deux sous-ordres est parfaite, ce sont des arborescences : on peut donc coder l'ordre,

au total, par le quadruplet d'une double double numérotation.

On peut aussi généraliser l'approche de la coloration, en considérant que la coloration bidirectionnelle revient à avoir 2 tables de couleurs : on généralise alors à n tables de couleurs, en allouant de nouvelles tables au fur et à mesure des vrais conflits rencontrés dans le déroulement de l'algorithme. Chaque classe s'identifie alors par sa couleur, sa table de couleur et l'identifiant de la classe parmi les classes de même couleur et table. Il n'est pas évident que l'algorithme glouton naturel, qui consiste à allouer une nouvelle table quand la classe considérée entre en conflit avec toutes les tables existantes, soit optimal pour le nombre de tables.

Mais l'optimisation nécessite surtout un compromis entre le nombre de trous dans les tables et le nombre de tables, sachant qu'une table supplémentaire doit être référencée par chaque classe : elle équivaut donc à un nombre de trous proportionnel au nombre de classes ($2N$ si chaque entrée occupe 2 octets). Dans nos hiérarchies de test, seules 2 ou 3 ont plus de 2 trous par classe en moyenne, ce qui pourrait justifier d'essayer avec une table de plus. Mais en admettant qu'il y ait un gain, celui-ci sera trop faible pour justifier de compliquer ainsi la structure.

Cette généralisation ressemble un peu à celle de [Agrawal *et al.*, 1989] dont la proposition généralise la double numérotation par autant d'intervalles qu'il est nécessaire.

6.3.2 Technique mixte : coloration du cœur

Lorsque le cœur est proportionnellement petit devant la hiérarchie totale, ce qui est le cas de toutes les hiérarchies à l'exclusion des 3 applications EIFFEL, il est encore possible de compacter le codage obtenu, avec une technique mixte, basée sur la coloration du cœur et sur la double numérotation de la couronne.

– un parcours en profondeur de la forêt de la couronne, y compris les attaches dans le cœur, produit une double numérotation n_1 et n_2 des classes, souvent appelée *Schubert's numbering* ou *relative numbering* [Schubert *et al.*, 1983 ; Raynaud et Thierry, 2001 ; Thierry, 2001]. n_2 peut être défini par

$$n_2(C) = \max_{D \preceq C}(n_1(D)) \tag{6.1}$$

Alors :

$$D \prec C \Leftrightarrow n_1(C) < n_1(D) \leq n_2(C) \tag{6.2}$$

Seuls deux petits entiers sont nécessaires, dont l'un (n_1) peut servir à identifier la classe.
Lorsque la cible C est toujours connue statiquement, il est possible de se contenter de stocker n_1, l'identifiant de la classe. Mais cette hypothèse peut être insuffisante pour la vérification de type lors des affectations à des attributs redéfinis de façon covariante [Ducournau, 2001a ; Ducournau, 2001b].

– les autres classes du cœur sont numérotées de telle sorte qu'elles soient incomparables avec toute autre classe par la condition 6.2 : on prendra par exemple $n_2(C) < n_1(C) < 0$;
– les classes du cœur sont colorées par vecteurs avec identifiant global : la couleur d'une classe peut être identifiée à $n_2(C)$;
– les classes de la couronne partagent le table de couleurs de leur attache.

Pour tester si $D \preceq C$, suivant le statut de la classe cible C :

– si C est une classe du cœur, il faut que $T_D[k(C)] = n_1(C)$: en effet, c'est la condition normale pour la coloration du cœur, si D en est, et sinon, c'est la condition qui doit être vérifiée par l'attache de D, car $T_D = T_{att(D)}$;
– si C est une classe de la couronne, il faut vérifier la condition 6.2.

La satisfaction d'une des deux conditions suffit, sans qu'il soit besoin de tester l'appartenance de C au cœur, c'est-à-dire $n_1(C) < 0$. Lorsqu'il est possible de savoir statiquement que C appartient au cœur, la compilation peut se traduire par l'inclusion du premier test. C'est le cas, en particulier, quand C a plusieurs super-classes directes. Mais il n'est jamais possible de déterminer statiquement l'appartenance de C à la couronne en compilation séparée. L'inclusion des deux tests est donc nécessaire, sans surcoût à l'exécution si l'édition de liens sait gérer des inclusions conditionnelles de code.

Enfin, le partage des tables de couleurs des attaches nécessite une indirection entre la structure de données qui contient n_1 et n_2 et la table de couleurs. On peut par exemple imaginer mettre n_1 et n_2 dans la table de méthodes ainsi qu'un pointeur sur la table de couleurs, ou mettre le pointeur sur la table de

couleurs dans l'objet, à côté de celui sur la table de méthodes, voire n_1 et n_2 aussi. Dans tous les cas, on voit que la double numérotation purement statique, sans stockage de n_2 ne sert sans doute pas à grand chose puisque n_1 et n_2 ne sont pas stockés avec la table de couleurs et qu'il n'y a rien pour compléter le mot dans lequel serait stocké n_1 seul.

Mais mettre des données communes supplémentaires dans les objets est sans doute un peu contradictoire avec l'objectif de cette technique de réduire l'occupation mémoire. La justification de cette technique étant la petite taille du cœur, il est donc raisonnable de considérer que la plupart des tests mettront en jeu la condition 6.2, et donc de permettre un accès plus rapide à n_1 et n_2 qu'à la table de couleurs.

L'allongement très significatif de la séquence de code fait sans doute réserver cette technique mixte à des cas de figure précis :

- pour de très grosses hiérarchies, avec un petit cœur et un test de sous-typage implémenté par une fonction qui n'est pas compilée en ligne : toutes les compactions sont alors admissibles si la hiérarchie est vraiment grosse ;
- pour une compilation globale, le test statique n'implémentant jamais que l'une des deux conditions ; on peut étendre ce cas à une compilation séparée dans une machine virtuelle sans chargement dynamique.

6.3.3 Minimisation des tables suivant le codage des entrées

Une fois calculée une coloration, on peut l'implémenter par plusieurs types de tables :

- une matrice repose sur une minimisation du nombre de couleurs utilisées, alors que des vecteurs, un par classe, permettent une minimisation de la taille totale des tables : cette dernière solution est imposée par la coloration bidirectionnelle ;
- qu'il s'agisse d'une matrice ou de vecteurs, le contenu de chaque entrée de la table peut être de taille fixe, ou de taille variable.

Dans le cas de la coloration de classes, chaque entrée de la table doit contenir un identifiant de la classe. On peut alors considérer :

- un identifiant global, qui nécessite $\lceil \log_2(N+1) \rceil$ bits ;
- un identifiant local à chaque couleur, implémenté de façon uniforme sur $\lceil \log_2(\max_i(N_i)+1) \rceil$ bits, où N_i est le nombre de classes de couleur i ;
- un identifiant local à chaque couleur, qui nécessite seulement $\lceil \log_2(N_i+1) \rceil$ bits, pour les classes de couleur i.

On réservera les termes de tableau, matrice et vecteur aux cas où les entrées sont uniformes : on parlera sinon de pseudo-tableau, pseudo-matrice ou pseudo-vecteurs. Un identifiant local uniforme transforme manifestement le problème de minimisation.

On peut enfin chercher à aligner les entrées :

- sans alignement ;
- de façon à ce qu'elles ne soient pas à cheval sur 2 mots ;
- à l'octet ;

Le choix du codage de l'identifiant a un effet sur la définition du problème d'optimisation, de même que le choix de l'alignement.

Codage uniforme sur un octet

Dans toutes nos hiérarchies, le codage des couleurs ne nécessite guère plus de 8 bits. Il était donc raisonnable d'essayer de limiter à 255 le nombre de classes par couleur, dans une version de l'algorithme dans laquelle une couleur n'est libre que lorsque moins de 255 classes l'ont comme couleur propre. Les expérimentations montrent une augmentation très significative du nombre de trous, mais une réduction tout aussi significative de la place occupée dans les versions uniformes (matrice et vecteur), avec des performances presqu'aussi bonne que dans les versions non uniformes (pseudo-tableaux).

Ces bons résultats s'expliquent par le faible dépassement du seuil de 8 bits et donc par la faible occupation du second octet dans les versions alignées. Dans le cas général, cela introduirait une nette dégradation. Limiter le nombre de classes par couleur revient en effet à échanger une somme par un produit : soit un arbre avec une racine ayant n sous-classes directes, avec $n = 255k$ et $1 < k < 255$. Les n sous-classes

peuvent se partager la même couleur, ce qui entraîne une occupation de $2n$ octets. En limitant le nombre de classes par couleur à 255, k couleurs sont nécessaires, au lieu d'une, et une coloration unidirectionnelle nécessitera $255k(k+1)/2$ octets, soit $(k+1)n/2$. Il faut donc que $k \leq 3$ pour que ce soit une amélioration. On constate que dans toutes nos hiérarchies, le nombre de bits par couleur ne dépasse pas 11, soit $k = 3$.

En pratique, la seule modification de l'algorithme concerne le calcul des couleurs libres pour la frontière et la couronne : en effet, le nombre de classes par couleur est, pour le cœur, toujours nettement inférieur à 255.

L'expérimentation confirme qu'il est préférable de modifier l'algorithme pour limiter à 255 le nombre de classes dans une couleur plutôt que de dédoubler les couleurs ayant plus de 255 classes.

Le codage sur un octet a un effet secondaire intéressant : il donne un identifiant global des classes sur 2 octets, constitués par la couleur et l'identifiant dans la couleur. Le nombre de couleurs reste lui aussi loin de 255 puisqu'il ne dépasse guère 50.

Test de débordement de table

Les implémentations par vecteurs nécessitent *a priori* un test sur la longueur de la table. Pourtant, comme remarqué par [Cohen, 1991], un identifiant global des classes permet d'éviter ce test dans l'implémentation par vecteurs. Il faut alors implémenter tous les vecteurs de façon contiguë, dans une unique table : si l'on dispose en premier une classe qui maximise la couleur négative et en dernier une classe qui maximise la couleur positive, aucun test sur la longueur de la table ne sera nécessaire (figure 6.2). On suppose alors que 0 appartient bien à l'intervalle $[k_{\min}, k_{\max}]$ de chaque table.

Toute autre implémentation par vecteurs nécessitera un test sur la longueur de la table et donc la présence de cette longueur dans la structure de données des tables, longueur pour laquelle il faudra donc réserver de la place supplémentaire (2 octets, soit de l'ordre de 20 % des versions les plus compactes). Cette donnée supplémentaire réduit sensiblement le gain procuré par des identifiants locaux, sans l'annuler.

En revanche, les implémentations par matrice ne nécessitent aucun test, toutes les lignes étant de même longueur, que l'identifiant soit global ou local, uniforme ou pas.

Chapitre 7

Algorithmes et heuristiques

7.1 Schéma d'algorithme

Une heuristique plus sophistiquée est sans doute nécessaire pour obtenir de meilleurs résultats.

Etant donné une extension linéaire, on oriente le graphe de conflits et pour toute classe, on note d^- (resp. d^+) le nombre de conflits de la classe parmi les classes qui la précèdent (resp. la suivent) dans l'extension.

Le schéma d'algorithme se précise ainsi :

```
chaque classe est munie d'une table illimitée ;
calculer le graphe de conflit des classes ;
calculer une extension linéaire de la spécialisation, restreinte au cœur
   * en choisissant, lorsqu'il y a plusieurs maximaux,
         celui qui a déjà des conflits colorés (d^- > 0)
               en suivant une certaine préférence concernant
                          les conflits pas encore colorés ;
pour chaque classe C du cœur, prise dans cet ordre :
   * calculer les indices libres, positifs et négatifs, de la table de C ;
   * choisir l'indice libre qui cause le moins de trous s'il est unique,
     et sinon choisir parmi ces indices suivant une certaine politique,
   * propager l'occupation de l'indice choisi dans les sous-classes de C
             et dans les classes en conflit avec C ;
pour chaque classe C de la couronne,
     dans l'ordre d'une extension linéaire quelconque
   * hériter des indices occupés par les super-classes indirectes,
            par copie de la table de la super-classe directe
   * choisir un indice dans les trous s'il y en a,
            sinon le premier indice libre, positif ou négatif.
finpour
```

Le schéma d'algorithme est maintenant paramétré par deux choix et un cas particulier :

- le choix de la classe maximale suivante, dans le cas où il y a plusieurs candidates ;
- le choix de la couleur, dans le cas où plusieurs couleurs libres provoquent le même nombre de trous ;
- ces deux choix sont court-circuités lorsqu'il existe un maximal dont l'une des couleurs libres est déjà occupée dans toutes les classes non encore colorées avec lesquelles il est en conflit (couleur parfaite) : l'affectation de cette couleur à cette classe est indépendante de tout autre choix car elle ne peut rajouter aucun trou dans le futur.

Les préférences exprimées par les deux critères ne sont pas absolues : les meilleures heuristiques donnent un meilleur résultat en moyenne, mais de moins bons dans certains cas. De plus, les conclusions sont incertaines : certaines grosses hiérarchies suffisent à classer une heuristique, dont les résultats sont majoritairement opposées sur les autres hiérarchies. A elle seule la hiérarchie Geode a autant de trous que toutes les autres.

7.2 Heuristiques

Les heuristiques et le résultat des expérimentations sont détaillés dans les sections suivantes.

7.2.1 Heuristiques du choix de la couleur

Nous avons essayé trois heuristiques sur le choix de la couleur parmi les *couleurs libres* d'une classe, c'est-à-dire l'ensemble des couleurs possibles compte tenu des colorations déjà faites, qui minimisent le nombre de trous dans la classe :

- dans tous les cas, il y a préférence pour la couleur parfaite, qui ne peut servir à aucune classe en conflit : le choix de la couleur parfaite est toujours le meilleur choix, quels que soient les choix ultérieurs ;
- pas d'autre préférence (heuristique 0) ;
- préférence à la couleur qui minimise le nombre de classes en conflit pour lesquelles la couleur est libre (heuristique 1) ;
- préférence à la couleur qui minimise le nombre de classes en conflit pour lesquelles cette couleur est une couleur libre mimimale (heuristique 2) ;
- préférence à la couleur qui minimise le nombre de classes en conflit pour lesquelles cette couleur est une couleur libre mimimale et, en second lieu, le nombre de classes en conflit qui peuvent se servir de la couleur (heuristique 3, combinaison des heuristiques 1 et 2).

7.2.2 Heuristiques du choix de la classe à colorer

- dans tous les cas, il y a préférence pour les classes qui ont une couleur parfaite ;
- pas d'autre préférence (heuristique 0) ;
- préférence à la classe qui minimise le nombre de classes en conflit (heuristique 1) ;
- préférence à la classe qui minimise le score de sa meilleure couleur libre (heuristique 2) ;
- préférence à la classe qui minimise le total des scores de ses couleurs libres (heuristique 3) ;
- préférence à la classe qui minimise le total des scores de ses couleurs libres et, en second lieu, qui minimise le score de la meilleur couleur (heuristique 4, combinaison de 2 et 3).

7.2.3 Heuristique de la bipartition

Une autre approche est possible. Une extension linéaire détermine, par l'algorithme de coloration bidirectionnelle, une bipartition du graphe de conflit. Cette bipartition n'est pas suffisante pour caractériser la coloration bidirectionnelle, l'ordre lui-même restant important. On constate sur des exemples que la meilleure et la pire solution peuvent se partager la même bipartition. L'idée est donc de calculer une première bipartition par l'algorithme précédent, puis d'essayer de raffiner la coloration en conservant la même bipartition.

Définition 7.1 *Etant donné une bipartition du graphe des conflits, un* vrai conflit *est un conflit entre deux classes de mêmes signes.*

On calcule alors une seconde extension linéaire dont le critère de choix serait de choisir, parmi les classes maximales du bon signe, celles qui minimisent le nombre de vrais conflits avec des classes pas encore colorées. L'expérimentation ne montre pas d'amélioration, plutôt une légère dégradation, mais elle n'a pas encore été effectuée, soit sur les bonnes heuristiques, soit sur une version purement aléatoire.

7.2.4 Variations stochastiques

L'heuristique résultant de la combinaison de ces deux critères de préférence sur les couleurs et les classes maximales, donne encore lieu à 2 variantes :

- une variante déterministe, dans laquelle l'algorithme prend la première couleur ou classe qui maximise le critère de préférence ;

– une variante stochastique dans laquelle l'algorithme choisit aléatoirement parmi l'ensemble des valeurs qui maximisent le critère.

Les statistiques montrent que la meilleure heuristique est l'heuristique 3-3, qui consiste à choisir la classe qui minimise le score de sa meilleure couleur, c'est-à-dire le nombre de classes en conflit pour lesquelles cette couleur est une couleur libre minimale.

7.2.5 Heuristiques tenant compte de la couronne

Les heuristiques que nous avons implémentées ou discutées ont l'avantage de ne s'appuyer que sur le cœur de la hiérarchie, ce qui réduit souvent la complexité. Mais la médaille a un revers : la couronne n'intervient pas. Ce n'est pas un problème si ses attaches sont régulièrement réparties dans le cœur, mais cela peut avoir un effet désastreux dans les mauvais cas, par exemple si la couronne a une très faible profondeur et est rattachée principalement aux classes du cœur qui ont le plus de trous. Les statistiques (figure 8.3) montrent que les classes du cœur peuvent avoir jusqu'à une trentaine de trous. Si toute la couronne est rattachée à ces classes là, avec une très faible profondeur, ces trous seront recopiés systématiquement et en quasi totalité dans chaque classe de la couronne. Une bonne heuristique devrait donc tenir compte de la couronne.

Cas de SELF

Les statistiques montrent cependant que l'heuristique basée sur le cœur se comporte plutôt bien lorsque l'on étend la coloration à la couronne, à l'exception notable du cas de SELF, pour lequel le gain entre la coloration unidirectionnelle minimisant le nombre de couleurs et la coloration bidirectionnelle n'est que de 1.3 pour la hiérarchie totale, alors qu'il est de 1.8 pour le cœur (figure 8.2). Ce n'est pas la coloration bidirectionnelle qui serait mauvaise, avec 2 % de trous, mais la coloration unidirectionnelle avec minimisation des couleurs qui est exceptionnellement bonne. Cela s'explique par la structure de la hiérarchie. Le cœur est proportionnellement petit (9 % des classes), mais les statistiques sur les super-classes indirectes sont étonnantes : leur nombre moyen est de 30.88, avec un écart-type de 5.79 et un maximum de 41 pour la hiérarchie totale, alors que le maximum sur le cœur est de 39, pour une moyenne de 19.86. Les colorations optimisant le nombre de couleurs, le coût par classe de la coloration minimisant les couleurs est à peine supérieur pour la hiérarchie que pour le cœur, mais les tables sont beaucoup plus remplies.

On constate aussi que la dispersion de l'algorithme est très faible sur le cœur — inférieur à 2 (624/327) — alors qu'il est très important sur la couronne — à peine inférieur à 9 (5396/610) : cela montre bien qu'il faudrait tenir compte de la couronne lorsque l'on colorie le cœur.

7.3 Coloration de classes : algorithme détaillé

7.3.1 Schéma général

Le schéma général de l'algorithme de coloration de classes est le suivant :

Algorithme 7.1 (Coloration de classes) *L'algorithme se décompose dans les étapes suivantes :*

1. *calcul d'une* extension linéaire descendante *de l'ensemble des classes ;*
2. *calculs préliminaires ;*
   – *calcul de la* profondeur *de chaque classe (optionnel) ;*
   – *calcul d'une* extension linéaire des super-classes de chaque classe *et élimination des* arcs de transitivité *: ces extensions linéaires sont appelées dans la suite* linéarisations *et notées* $lin(C)$ *;*
   – *calcul du* cœur*, de la* frontière *et de la* couronne *: chaque classe est étiquetée par l'un des mots-clés* `:core`*,* `:border` *ou* `:crown` *;*
3. *calcul du* graphe de conflit *sur les éléments du cœur ;*
4. *coloration proprement dite :*
   – coloration des classes *du cœur ;*
   – coloration des classes *de la frontière puis de la couronne ;*

5. calcul des tables de couleurs.

Commentaires. L'extension linéaire de l'étape 1 est indispensable. En revanche, l'étape 2 nécessite juste l'élimination des acrs de transitivité et le calcul des ensembles $lin(C)$, pas forcément ordonnés : les extensions linéaires ne coûtent pas plus cher que de collecter de simples ensembles, qui sont indispensables pour le graphe de conflit. Les étapes 2 et 3 peuvent être menées en parallèle, classe par classe.

Graphe de conflit

Parmi les étapes préliminaires, seul l'algorithme de calcul du graphe de conflit nécessite quelques indications. La technique retenue est la suivante :

Algorithme 7.2 (Graphe de conflit) *Pour chaque classe C, dans l'ordre de la linéarisation, soit C_1, C_2 ... C_n ses n super-classes directes. 4 cas se présentent :*

- $n \leq 1$ *: ne rien faire*
- $n = 2$ *:*
  calculer les différences entre les 2 linéarisations $D_{12} = lin(C_1) - lin(C_2)$ et $D_{21} = lin(C_2) - lin(C_1)$ et rajouter au graphe de conflit les éléments du produit $D_{12} \times D_{21}$ qui n'y sont pas déjà ;
- $n = 3$ *:*
  pour tout $i = 1..n$, calculer $L_i = \cup_{j \neq i} lin(C_j)$ et les différences $D'_i = lin(C_i) - L_i$ et $D''_i = L_i - lin(C_i)$ et rajouter au graphe de conflit les éléments du produit $D'_i \times D''_i$ qui n'y sont pas déjà ;
- $n > 3$ *:*
  opérer comme pour $n = 3$ puis affecter un compteur $count(x) = 0$ à chaque élément x de $lin(C)$: pour tout $i = 1..n$ incrémenter de 1 le compteur de toutes les classes de $lin(C_i)$; soit alors $F = \{x \in lin(C) \mid 1 < count(x) < n - 1\}$: pour tout $(x, y) \in F \times F$, rajouter (x, y) au graphe de conflit si la paire n'y est pas déjà et que x et y sont incomparables par la relation de spécialisation.

Commentaires : pour être complet, avec n super-classes directes, il faudrait effectuer $2^{n-1}-1$ produits. Dès que $n > 3$, les n produits laissent donc échapper les conflits incidents à une classe appartenant à au moins 2 linéarisations et au plus $n - 2$, d'où le calcul de F. L'algorithme ne prétend pas être efficace pour n'importe quel ordre, mais, dans les hiérarchies de classes, les cas où $n > 3$ sont rares et F est alors très petit. Un algorithme plus brutal, consistant à faire le calcul de $lin(C) \times lin(C)$, nécessiterait d'avoir un test de fermeture transitive efficace...

7.3.2 Coloration du cœur

Le principe de l'algorithme consiste à maintenir 3 listes de classes maximales, ordonnées suivant un certain critère heuristique, à faire un choix parmi ces maximaux et à lui affecter une couleur choisie suivant un autre critère heuristique. Les deux critères peuvent être corrélés, dans la mesure où le premier peut être fonction du second.

Chaque classe a un statut dépendant de son état courant dans le déroulement de l'algorithme : ce statut peut prendre les valeurs `:colored`, `:max-cf`, `:max-ncf`, `:max-border`, `:notmax-cf`, `:notmax-ncf` ou `:notmax-border`. Ces valeurs se décodent simplement : `cf` indique une classe dont un conflit a déjà été coloré, alors que `ncf` indique une classe du cœur dont aucun conflit n'a encore été coloré. Les transitions entre ces états sont décrites dans la figure 7.1 :

1. à l'origine toutes les classes du cœur sont `:notmax-ncf` et celle de la frontière `:notmax-border` ;
2. une classe devient maximale lorsque toutes ses super-classes directes sont `:colored` : les *couleurs libres minimales* et le *score* de la classe sont alors calculés, suivant l'heuristique choisie ; la classe est insérée dans la liste de maximaux correspondant, dans l'ordre des scores croissants ;
3. une classe passe de `ncf` à `cf` lorsque l'un de ses conflits vient d'être coloré ;
4. on colore d'abord les `:max-cf`,
5. avant les `:max-ncf`,

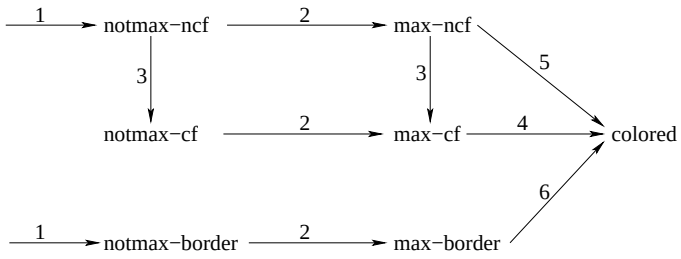

FIG. 7.1 – Transitions pendant la coloration

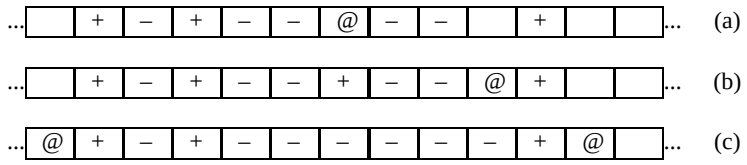

FIG. 7.2 – Couleurs libres minimales : "–" désigne une couleur héritée et "+" une couleur occupée par un conflit. Les couleurs libres minimales ("@") sont, soit les couleurs libres à l'intérieur des couleurs héritées (a), soit la ou les couleurs les plus proches à l'extérieur des couleurs héritées (b et c).

6. pour finir par les `:max-border`.

L'algorithme démarre en rendant la racine maximale. Quand une classe est colorée,
- sa table de couleurs est propagée à ses sous-classes directes, qui peuvent devenir maximales,
- sa couleur est propagées aux classes avec lesquelles elle est en conflit, ce qui peut modifier leur statut ainsi que leurs couleurs libres minimales et leur score.

La propagation se fait en 2 temps : d'abord calcul des couleurs libres minimales des classes maximales nouvelles ou modifiées, puis calcul de leur score et insertion dans les listes de maximaux, ordonnés par leur score. Les couleurs libres de 2 classes sont indépendantes, de même que leurs scores, mais, suivant l'heuristique, le score de chacune peut dépendre des couleurs libres de l'autre.

Enfin, la coloration de la frontière et de la couronne a la particularité de ne pas pouvoir produire de nouvaux trous : les classes sont donc colorées dans un ordre quelconque pour la frontière (qui est une anti-chaîne) et en suivant une extension linéaire pour la couronne, en prenant la première couleur libre.

En cas de coloration bidirectionnelle, il est possible d'optimiser un critère supplémentaire, le nombre total de couleurs utilisées : on peut donc choisir une couleur libre qui n'augmente pas ce nombre et, si ce n'est pas possible, on prend la couleur libre positive. On évite ainsi d'augmenter aussi bien le nombre de couleurs positives que le nombre de couleurs négatives[1]

Couleurs libres minimales d'une classe maximale

Le but de l'algorithme est d'affecter une couleur à une classe : cette couleur sera dite *couleur propre* de la classe.

Avant cette affectation et au fur et à mesure du déroulement de l'algorithme, la table de couleurs de la classe est remplie de la façon suivante :
- les couleurs propres de ses super-classes indirectes sont occupées par les classes correspondantes (propagation des super-classes) : on parlera de *couleur héritée* ;
- les couleurs propres des classes en conflit déjà colorées sont occupées par la liste des classes correspondantes (propagation des conflits) : on parlera de *couleur occupée*.

Dans la variante de l'algorithme qui implémente chaque couleur sur un octet, une couleur sera *saturée* dès que 255 classes l'ont comme couleur propre.

[1] Cette dernière optimisation sert à limiter la taille de la matrice obtenue, ce qui permet une comparaison plus réaliste. Elle ne présente aucun intérêt pratique dans l'implémentation de la coloration bidirectionnelle : elle peut même avoir un effet négatif pour une implémentation par pseudo-vecteurs qui gagnerait à augmenter le nombre total de couleurs pour réduire le nombre de classes de la même couleur.

Les autres couleurs sont *libres*. Les *couleurs libres minimales* sont définies de la façon suivante (figure 7.2) :

1. soit il y a une ou plusieurs couleurs libres dans l'intervalle entre les couleurs héritées maximale et minimale : ce sont alors les couleurs libres minimales ;
2. on calcule sinon, les plus petites couleurs libres positives et négatives : si elles sont à égale distance des couleurs héritées extrêmes, ce sont les couleurs libres minimales ;
3. on prend sinon, celle qui est la plus proche de l'extrêmité correspondante.

En coloration unidirectionnelle, les points 2 et 3 sont remplacés par :

2'. on prend sinon la plus petite couleur libre.

En coloration bidirectionnelle, la taille de la table de couleurs d'une classe est déterminée par les valeurs maximales et minimales de ses couleurs propre ou héritées.

Score d'une couleur

Le score d'une couleur présente deux cas, suivant qu'elle est *parfaite* ou qu'il lui faut subir un traitement heuristique :

- une *couleur parfaite* est une couleur libre minimale qui n'est libre dans aucune des classes non encore colorées avec lesquelles la classe est en conflit : lorsqu'une classe n'a pas de conflits, par exemple si elle appartient à la frontière ou si elle est la dernière de sa composante connexe, toutes ses couleurs libres minimales sont parfaites ;
- le traitement des couleurs qui ne sont pas parfaites dépend de l'heuristique choisie : il s'agit du nombre de classes en conflit où la couleur est libre (heuristique 1) ou libre minimale (heuristique 2), ou encore une combinaison des deux, l'heuristique 1 servant à départager les égaux de l'heuristique 2 (heuristique 3).

Score d'une classe maximale

Le score d'une classe présente deux cas, suivant qu'elle a une *couleur parfaite* ou qu'il lui faut subir un traitement heuristique :

- lorsqu'une classe a une *couleur parfaite*, son score est négatif, ce qui fait que les classes qui ont une couleur parfaite sont toujours traitées d'abord[2]
- sinon, la classe a un score dépendant de l'heuristique choisie : nombre de classes en conflit (heuristique 1), minimum (heuristique 2) ou somme (heuristique 3) des scores de ses couleurs libres, ou combinaison des 2 (heuristique 4).

7.3.3 Coloration de la couronne

La coloration de la couronne est une version simplifiée dans laquelle il n'y a plus de conflit : toutes les couleurs libres sont alors parfaites. C'est la version de la coloration en héritage simple [Cohen, 1991 ; Queinnec, 1997] :

Algorithme 7.3 *Pour chaque classe C de la couronne, dans l'ordre d'une extension linéaire descendante :*

- *remplir la table de couleurs de C par copie de la table de couleurs de sa super-classe directe ;*
- *affecter à la classe l'une de ses couleurs libres minimales.*

Comme il n'y a pas de couleurs occupées par un conflit, les couleurs libres sont, soit les trous dans les couleurs héritées, soit les premières couleurs libres positive et négative (en cas de coloration bidirectionnelle) : cas 1 et 2 ou 2'. Dans la variante où chaque couleur doit être implémentée sur un octet, on préfèrera la couleur libre qui minimise le nombre de classes dont elle est la couleur propre.

[2] Ce traitement privilégié n'a aucun effet sur l'algorithme puisqu'une couleur parfaite reste parfaite : il sert à évacuer les classes aux couleurs parfaites avant un éventuel traitement stochastique.

7.3.4 Variantes

Coloration du cœur par bipartition

Chapitre 8

Expérimentations et statistiques

8.1 Statistiques

Nous avons donc implémenté le schéma d'algorithme général décrit précédemment et l'avons testé, dans les diverses variantes présentées précédemment, sur les hiérarchies de classes décrites en annexe A.2.

8.1.1 Graphe de conflit

La figure 8.1 donne des statistiques sur le graphe de conflit des différentes hiérarchies, en particulier sur sa connexité. Il apparaît que ce graphe est en général formé d'une grosse composante connexe, elle même formée d'une grosse composante 2-connexe : de très petites composantes connexes et 2-connexes accompagnent ces grosses composantes. Ces statistiques montrent que les conflits sont relativement indécomposables et que le cœur en héritage multiple est fortement intriqué.

8.1.2 Variantes déterministes

Les figures suivantes donnent les résultats des meilleures heuristiques essayées (3-3 et 4-3), dans leur versions déterministes. La figure 8.2 donne les résultats globaux de la coloration suivant qu'elle est unidirectionnelle ou bidirectionnelle et, dans le premier cas, suivant le critère de minimisation (nombre de couleurs ou taille des tables). Les statistiques sont relatives au cœur, en haut, ou à la totalité de la hiérarchie, en bas. On voit que le gain de la coloration bidirectionnelle relativement à la coloration unidirectionnelle minimisant le nombre de couleurs (technique [Vitek *et al.*, 1997]) est en moyenne de l'ordre de 2.5, et un peu meilleur, toujours en moyenne et à l'exception de SELF, pour la hiérarchie globale.

Enfin, la figure 8.3 donne le détail des résultats de la coloration, en nombre de couleurs, taille des tables et nombre et taux de trous pour le cœur des hiérarchies. On constate que la coloration unidirectionnelle donne, sur toutes les hiérarchies, un résultat optimum en ce qui concerne le nombre de couleurs, qui est toujours égal au nombre maximum de super-classes indirectes d'une classe : il y a donc une classe qui utilise toutes les couleurs. Notre heuristique est donc apparemment meilleure[1] que celle de [Vitek *et al.*, 1997], qui montre un léger éloignement de l'optimum dans le cas des 3 hiérarchies EIFFEL. C'est aussi le cas pour la bi-coloration, mais les statistiques sur toutes les heuristiques montrent une plus grande dispersion du nombre de couleurs malgré hiérarchie, malgré un gain total important (figure 8.5).

On constate aussi que le facteur χ, qui représente le surcoût de la coloration par rapport à la fermeture transitive et qui vaut 1 lorsque le graphe de conflit est biparti, est inférieur à 1.55 dans toutes les hiérarchies considérées et égal, en moyenne, à 1.15.

[1] Dans notre expérimentation, l'optimum est atteint pour le *cœur* des hiérarchies et non sur la totalité comme dans l'article cité. Si toutes les classes qui ont le plus grand nombre de super-classes indirectes appartiennent à la couronne, l'optimalité globale n'implique pas forcément l'optimalité du cœur. En revanche, l'optimalité du cœur implique toujours l'optimalité globale, car une classe de la couronne bouche toujours les trous hérités de sa super-classe avant de faire appel à une nouvelle couleur.

Système	taille	nb	nb d'arêtes			comp. connexes			comp. 2-connexes		
	cœur	sommets	moy	max	total	nb	moy	max	nb	moy	max
Java-a	34	19	1.7	6	16	3	6.3	8	13	2.0	2
Java-b	105	61	3.9	17	120	4	15.2	34	18	4.1	20
Java-1.18	299	168	5.1	33	425	8	21.0	152	65	3.3	59
Java-1.22	966	444	6.0	152	1327	24	18.5	347	161	3.5	202
Java-1.30	1140	546	6.5	191	1777	28	19.5	414	186	3.7	247
IDL	11	4	1.5	3	3	1	4.0	4	3	2.0	2
Unidraw	25	14	2.1	5	15	2	7.0	9	7	2.6	5
Lov-obj-ed	271	159	15.6	81	1241	2	79.5	157	18	9.7	141
Geode	989	500	22.5	258	5613	4	125.0	482	42	12.8	444
Eiffel	910	593	17.1	197	5068	5	118.6	584	61	10.6	511
Laure	57	39	3.9	14	77	3	13.0	29	10	4.5	25
Self	154	104	27.4	62	1427	2	52.0	101	15	7.7	88
Cecil	306	167	6.1	46	511	8	20.9	130	33	5.7	90
Total	5267	2818	12.5	258	17620	94	30.0	584	632	5.2	511

FIG. 8.1 – Graphe de conflit

8.1.3 Variantes stochastiques

Les variantes stochastiques permettent à la fois d'étudier la variabilité des résultats et de comparer les diverses variantes.

La figure 8.5 donne le résultat global de chaque heuristique, résumé par les nombres totaux de trous et maximaux de couleurs, avec leur variation (minimum, moyenne et maximum). Les résultats proviennent de plus d'une centaine d'exécutions de chaque version. Il apparaît ainsi que seule l'heuristique 3-3 et, dans une moindre mesure, la 4-3 se révèlent meilleures, en moyenne, que toute autre heuristique et meilleures que le choix purement aléatoire de l'absence d'heuristique (0-0). En revanche, de nombreuses heuristiques apparaissent comme mauvaises, au sens où elles donnent de moins bons résultats que ce choix purement aléatoire. La figure 8.7 donne le détail des meilleures variantes pour chaque hiérarchie : on constate ainsi que l'heuristique 3-3 est non seulement meilleure en moyenne sur la totalité des hiérarchies, elle l'est aussi sur chacune d'elles, à quelques exceptions près.

On peut constater aussi à quel point certaines des conjectures énoncées précédemment peuvent être fausses : ainsi, pour le cœur, le maximum de la coloration bidirectionnelle est du même ordre que le minimum de la coloration unidirectionnelle, alors qu'il est supérieur pour la hiérarchie globale. Plus précisément, sur les grosses hiérarchies JAVA, la coloration unidirectionnelle est souvent meilleure que la bidirectionnelle.

8.2 Implémentation des tables

Les figures 8.10, 8.8 et 8.9 donnent les tailles totales résultant des diverses stratégies d'implémentation, dans les variantes suivantes :

- sans alignement, chaque entrée étant constitué d'un champ de bits pouvant être à cheval sur deux mots consécutifs ;
- avec alignement au mot, un mot peut contenir plusieurs entrées, mais une entrée ne peut être à cheval sur deux mots ;
- en octets, chaque entrée est constituée d'un nombre entier d'octets ;
- en double-octets, chaque entrée est constituée de deux octets (ou petit entier) : il n'y a alors plus de différence entre tableau et pseudo-tableau ;
- en octet unique : chaque entrée est constituée d'un octet et l'algorithme ne considère qu'une couleur est libre que si elle est occupée par moins de 255 classes.

Dans tous les cas, chaque vecteur ou ligne de matrice occupe un nombre entier de mots de 32 bits et, pour les colorations bidirectionnelles, la couleur 0 est alignée sur un mot.

Système	nb classes	coloration unidirectionnelle min. nombre		coloration unidirectionnelle minimise taille		coloration bidirection. minimise taille		gain taille	gain trous	taux trous	ferm. trans.
Java-a	34	272	137	148	13	135	0	2.0	∞	0.00	135
Java-b	105	1470	1017	540	87	473	20	3.1	51.	0.04	453
Java-1.18	299	4784	3413	1697	326	1462	91	3.3	38.	0.07	1371
Java-1.22	966	16422	11531	7040	2149	5465	574	3.0	20.	0.12	4891
Java-1.30	1140	21660	15712	8611	2663	6867	919	3.2	17.	0.15	5948
IDL	11	77	29	51	3	48	0	1.6	∞	0.00	48
Unidraw	25	225	113	124	12	113	1	2.0	113.	0.01	112
Lov-obj-ed	271	6504	4057	4145	1698	3490	1043	1.9	4.	0.43	2447
Geode	989	49450	34087	25948	10585	22480	7117	2.2	5.	0.46	15363
Eiffel	910	35490	26056	14053	4619	12225	2791	2.9	9.	0.30	9434
Laure	57	912	504	472	64	416	8	2.2	63.	0.02	408
Self	154	6006	2948	3681	623	3394	336	1.8	9.	0.11	3058
Cecil	306	6120	3768	2733	381	2569	217	2.4	17.	0.09	2352
Total (3/3)	5267	149392	103372	69243	23223	59137	13117	2.5	8.	0.29	46020
Java-a	226	1808	868	953	13	940	0	1.9	∞	0.00	940
Java-b	604	8456	5654	2962	160	2822	20	3.0	283.	0.01	2802
Java-1.18	1704	27264	19853	8068	657	7575	164	3.6	121.	0.02	7411
Java-1.22	4339	73763	54822	23475	4534	20393	1452	3.6	38.	0.08	18941
Java-1.30	5438	103322	79550	31518	7746	25961	2189	4.0	36.	0.09	23772
IDL	67	603	283	323	3	320	0	1.9	∞	0.00	320
Unidraw	614	6140	3672	2480	12	2469	1	2.5	3672.	0.00	2468
Lov-obj-ed	436	10464	6757	6177	2470	5356	1649	2.0	4.	0.44	3707
Geode	1318	65900	47458	30846	12404	26759	8317	2.5	6.	0.45	18442
Eiffel	1999	77961	60402	24313	6754	21522	3963	3.6	15.	0.23	17559
Laure	293	4688	2296	2492	100	2400	8	2.0	287.	0.00	2392
Self	1802	73882	18243	57114	1475	56716	1077	1.3	17.	0.02	55639
Cecil	932	21436	15404	6594	562	6319	287	3.4	54.	0.05	6032
Total (3/3)	19772	475687	315262	197315	36890	179552	19127	2.6	16.	0.12	160425

FIG. 8.2 – Statistiques de coloration suivant le critère de minimisation, pour le cœur en haut et le total en bas : chaque coloration a deux colonnes, pour la taille totale et les trous (heuristique 3-3)

Système	nb classes	fermeture transit.	coloration unidirectionnelle						χ	coloration bidirectionnelle						χ
			nb couleurs			nb trous				nb couleurs			nb trous			
			max	moy	total	max	moy	total		max	moy	total	max	moy	total	
Java-a	34	135	8	4.35	148	1	0.38	13	1.10	8	3.97	135	0	0.00	0	1.00
Java-b	105	453	14	5.14	540	6	0.83	87	1.19	14	4.50	473	2	0.19	20	1.04
Java-1.18	299	1371	16	5.68	1697	11	1.09	326	1.24	16	4.89	1462	5	0.30	91	1.07
Java-1.22	966	4891	17	7.29	7040	13	2.22	2149	1.44	17	5.66	5465	8	0.59	574	1.12
Java-1.30	1140	5948	19	7.55	8611	15	2.34	2663	1.45	19	6.02	6867	8	0.81	919	1.15
IDL	11	48	7	4.64	51	1	0.27	3	1.06	7	4.36	48	0	0.00	0	1.00
Unidraw	25	112	9	4.96	124	2	0.48	12	1.11	9	4.52	113	1	0.04	1	1.01
Lov-obj-ed	271	2447	24	15.30	4145	19	6.27	1698	1.69	24	12.88	3490	17	3.85	1043	1.43
Geode	989	15363	50	26.24	25948	43	10.70	10585	1.69	50	22.73	22480	25	7.20	7117	1.46
Eiffel	910	9434	39	15.44	14053	28	5.08	4619	1.49	39	13.43	12225	16	3.07	2791	1.30
Laure	57	408	16	8.28	472	5	1.12	64	1.16	16	7.30	416	2	0.14	8	1.02
Self	154	3058	39	23.90	3681	31	4.05	623	1.20	39	22.04	3394	15	2.18	336	1.11
Cecil	306	2352	20	8.93	2733	12	1.25	381	1.16	20	8.40	2569	12	0.71	217	1.09
Total (3/3)	5267	46020	50	13.15	69243	43	4.41	23223	1.50	50	11.23	59137	25	2.49	13117	1.29
Java-a	226	940	8	4.22	953	1	0.06	13	1.01	8	4.16	940	0	0.00	0	1.00
Java-b	604	2802	14	4.90	2962	6	0.26	160	1.06	14	4.67	2822	2	0.03	20	1.01
Java-1.18	1704	7411	16	4.73	8068	11	0.39	657	1.09	16	4.45	7575	5	0.10	164	1.02
Java-1.22	4339	18941	17	5.41	23475	13	1.04	4534	1.24	17	4.70	20393	8	0.33	1452	1.08
Java-1.30	5438	23772	19	5.80	31518	15	1.42	7746	1.33	19	4.77	25961	8	0.40	2189	1.09
IDL	67	320	9	4.82	323	1	0.04	3	1.01	9	4.78	320	0	0.00	0	1.00
Unidraw	614	2468	10	4.04	2480	2	0.02	12	1.00	10	4.02	2469	1	0.00	1	1.00
Lov-obj-ed	436	3707	24	14.17	6177	19	5.67	2470	1.67	24	12.28	5356	17	3.78	1649	1.44
Geode	1318	18442	50	23.40	30846	43	9.41	12404	1.67	50	20.30	26759	25	6.31	8317	1.45
Eiffel	1999	17559	39	12.16	24313	28	3.38	6754	1.38	39	10.77	21522	16	1.98	3963	1.23
Laure	293	2392	16	8.51	2492	5	0.34	100	1.04	16	8.19	2400	2	0.03	8	1.00
Self	1802	55639	41	31.69	57114	31	0.82	1475	1.03	41	31.47	56716	15	0.60	1077	1.02
Cecil	932	6032	23	7.08	6594	12	0.60	562	1.09	23	6.78	6319	12	0.31	287	1.05
Total (3/3)	19772	160425	50	9.98	197315	43	1.87	36890	1.23	50	9.08	179552	25	0.97	19127	1.12

FIG. 8.3 – Statistiques de coloration pour l'heuristique (3-3)

Geode	0	1	2	3	4
0	8937	7710	9386	6581	8124
1	8990	7383	7340	6769	6781
2	8983	7172	7587	6796	7139
3	9007	6884	8400	6964	6989

sf Geode	0	1	2	3	4
0	7163	10541	7459	6844	6566
1	7118	9897	9569	9176	9012
2	6843	9692	6998	7223	6423
3	6723	9466	6548	6036	6167

FIG. 8.4 – Comparaison des heuristiques sur Geode et les autres hiérarchies

8.2.1 Conclusions

La figure 8.10 résume les occupations mémoire obtenues suivant les différentes implémentations. On est donc conduit à trouver un compromis :

- entre la taille des tables, dans un rapport de 1 à 5 entre les pseudo-vecteurs de bits et les matrices de double-octets ;
- la taille et l'efficacité du code implémentant le test, y compris l'éventuelle extraction du champ de bits visé et le test de débordement de table.

Il apparaît que l'implémentation par vecteurs avec un codage uniforme par octet ou double-octet est certainement le meilleur compromis espace-temps. Si l'efficacité temporelle n'est pas le critère principal, ou pour une hiérarchie d'un ordre de grandeur très supérieur, une implémentation par pseudo-vecteur est possible, à condition qu'elle soit effectuée par un appel de fonction — par exemple autant de fonctions qu'il y a de types cibles — pour factoriser la place occupée par le test de débordement et les instructions d'extraction du champ de bits.

8.3 Technique mixte

8.4 Performance du codage d'ordre

Enfin, la figure 8.12 résume les performances de la coloration de classes comme codage d'ordre, dans ses deux versions les plus compactes :

- les pseudo-vecteurs sans aucun alignement avec le codage des longueurs le plus compact possible (sur $\lceil \log_2(1 + \max(c_i)) \rceil$ et $\lceil \log_2(1 + \min(c_i)) \rceil$) ;
- la technique mixte, utilisant la précédente pour le cœur et le codage le plus compact de n_1 et n_2 en $\lceil \log_2(N) \rceil$ bits.

Pour chaque technique, la colonne $/c^2$ donne le nombre de bits par classe au carré, c'est-à-dire le rapport entre les tables de colorations et la matrice de fermeture transitive : le gain apporté par la coloration est significatif, de l'ordre de la racine carrée du nombre de classes. Enfin, la colonne $/| \preceq |$ donne le rapport entre le nombre de bits et la taille de la fermeture transitive $| \preceq |$: dans une implémentation avec identifiant global, ce rapport serait égal à $\chi\lceil \log_2(N + 1) \rceil$. Les théoriciens se posent la question de savoir s'il existe un codage de taille $| \preceq |$, donc avec un rapport 1 [Thierry, 2001].

La technique mixte apporte encore une amélioration : seule la version complète, non statique, avec stockage de n_1 et n_2 a été considérée car le compactage absolu est relativement incompatible avec une compilation en ligne statique.

8.4.1 Test et codage statiques ou dynamiques

Le test de sous-typage peut-être statique ou dynamique, suivant que la classe cible est connue statiquement ou pas.

heur.			itér.	coloration unidirectionnelle						coloration bidirectionnelle					
				cœur			total			cœur			total		
3	3		122	22509	23283.	24454	36192	37341.	39130	11265	**13005.**	15716	16018	19243.	24329
4	3		121	22508	23447.	24497	35130	37062.	39127	11401	**13161.**	15806	16309	19449.	23999
4	2		41	22982	23250.	23583	38745	39162.	39809	11834	**13567.**	15681	17000	21073.	24750
3	2		41	23035	23224.	23472	38809	39181.	39788	12356	**14022.**	16816	18006	21307.	26344
2	2		46	22404	23455.	24780	36196	38243.	40332	12661	**14589.**	16708	19888	23746.	27798
0	2		134	22358	27590.	37842	35604	48644.	74786	11071	**15832.**	23950	14847	25973.	41841
3	1		41	30014	30015.	30017	58749	58751.	58754	14793	**15949.**	17749	25085	28382.	32735
0	0		202	22175	27881.	38880	33923	49341.	79934	10866	**16107.**	23895	15563	26612.	42284
0	1		170	21734	27614.	37271	34202	48958.	74692	10796	**16115.**	23444	15365	26604.	42860
1	2		84	32775	33151.	33562	67455	68374.	69257	14644	**16868.**	19379	26866	31377.	35581
2	1		46	32332	32339.	32362	66303	66320.	66537	14342	**16914.**	19959	26695	31744.	37691
1	1		172	27902	32498.	37643	51620	66921.	83297	13391	**17287.**	21332	24160	32925.	42737
2	0		46	21901	23652.	25684	35258	38909.	42852	10825	**16849.**	21460	17398	26324.	36215
1	0		170	27930	33630.	39811	51733	69550.	87669	13436	**18256.**	23245	24055	34487.	45053
3	3		122	278	278.0	278	286	286.0	286	278	**278.5**	282	286	286.5	290
4	3		121	278	278.0	278	286	286.0	286	278	**278.6**	284	286	286.5	291
4	2		41	278	278.2	279	286	286.2	287	278	**279.9**	285	286	287.5	292
3	2		41	278	278.0	278	286	286.0	286	278	**279.5**	283	286	287.5	291
2	2		46	278	278.7	280	286	286.7	288	278	**279.3**	285	286	287.3	293
0	2		134	278	278.0	279	286	286.0	287	278	**278.3**	285	286	286.3	292
3	1		41	278	278.0	278	286	286.0	286	278	**281.4**	284	286	289.4	292
0	0		202	278	278.0	279	286	286.0	287	278	**278.4**	292	286	286.4	299
0	1		170	278	278.0	278	286	286.0	286	278	**278.1**	285	286	286.1	293
1	2		84	279	279.0	279	287	287.0	287	281	**282.8**	287	289	290.8	295
2	1		46	279	279.0	279	287	287.0	287	280	**282.8**	286	288	290.8	294
1	1		172	278	278.2	279	286	286.2	287	278	**279.2**	287	286	287.2	295
2	0		46	278	278.3	280	286	286.3	288	278	**279.1**	287	286	287.1	294
1	0		170	278	278.4	282	286	286.4	290	278	**279.4**	292	286	287.4	300

FIG. 8.5 – Bilan des heuristiques, classées par nombre moyen croissant de trous pour la coloration bidirectionnelle du cœur (colonne en gras).

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	min. absolu	heuristique 3-3			moy. absolu	max. absolu	min. absolu	heuristique 3-3			moy. absolu	max. absolu
		min.	moy.	max.				min.	moy.	max.		
Java-a	13	13	13.	13	13.	14	0	0	0.	0	0.	0
Java-b	76	87	87.	87	95.	141	9	19	20.	21	18.	48
Java-1.18	267	326	326.	326	356.	671	67	80	87.	95	108.	220
Java-1.22	1566	2149	2149.	2149	2433.	6262	515	515	625.	981	956.	2986
Java-1.30	1926	2663	2664.	2667	3079.	8452	736	778	1044.	1483	1311.	3811
IDL	3	3	3.	3	3.	3	0	0	0.	0	0.	0
Unidraw	12	12	12.	12	13.	14	1	1	1.	2	2.	3
Lov-obj-ed	1300	1698	1698.	1698	1662.	2267	711	857	1015.	1299	1029.	1583
Geode	10086	10086	10655.	11354	13119.	15334	4640	6289	6964.	7854	8938.	12457
Eiffel	4286	4404	4609.	5077	5794.	8616	2240	2240	2724.	3413	3083.	4871
Laure	59	64	64.	64	73.	102	7	8	8.	8	15.	34
Self	621	623	623.	623	761.	962	327	334	337.	340	470.	624
Cecil	381	381	381.	381	479.	713	107	144	180.	220	179.	366
Total (3/3)	20596	22509	23283.	24454	27881.	43551	9360	11265	13005.	15716	16107.	27003
Java-a	13	13	13.	13	13.	14	0	0	0.	0	0.	0
Java-b	105	160	160.	160	145.	240	9	19	20.	21	19.	59
Java-1.18	394	657	657.	657	681.	2007	84	133	149.	168	150.	385
Java-1.22	3657	4534	4534.	4534	6701.	18665	1072	1163	1569.	2520	2501.	7864
Java-1.30	5092	7746	7747.	7750	8878.	26149	1560	1903	2792.	3946	3413.	10400
IDL	3	3	3.	3	3.	3	0	0	0.	0	0.	0
Unidraw	12	12	12.	12	13.	14	1	1	1.	2	2.	3
Lov-obj-ed	1845	2470	2470.	2470	2397.	3291	896	1122	1450.	1831	1444.	2322
Geode	11797	11797	12482.	13312	15834.	18847	5454	7337	8117.	9150	10622.	14662
Eiffel	6262	6663	7126.	8082	9644.	17296	3090	3090	3830.	5304	4731.	9156
Laure	73	100	100.	100	154.	341	7	8	8.	8	19.	75
Self	1274	1475	1475.	1475	4068.	4306	610	1070	1076.	1081	3466.	5396
Cecil	530	562	562.	562	809.	2099	118	172	231.	298	246.	728
Total (3/3)	31057	36192	37341.	39130	49341.	93272	12901	16018	19243.	24329	26612.	51050

FIG. 8.6 – Situation de l'heuristique 3-3 par rapport à la version totalement stochastique du schéma d'algorithme.

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	cœur			total			cœur			total		
Java-b	87	87.	87	160	160.	160	19	20.	21	19	20.	21
Java-1.18	326	326.	326	657	657.	657	80	87.	95	133	149.	168
Java-1.22	2149	2149.	2149	4534	4534.	4534	515	625.	981	1163	1569.	2520
Java-1.30	2663	2664.	2667	7746	7747.	7750	778	1044.	1483	1903	2792.	3946
Unidraw	12	12.	12	12	12.	12	1	1.	2	1	1.	2
Lov-obj-ed	1698	1698.	1698	2470	2470.	2470	857	1015.	1299	1122	1450.	1831
Geode	10086	10655.	11354	11797	12482.	13312	6289	6964.	7854	7337	8117.	9150
Eiffel	4404	4609.	5077	6663	7126.	8082	2240	2724.	3413	3090	3830.	5304
Laure	64	64.	64	100	100.	100	8	8.	8	8	8.	8
Self	623	623.	623	1475	1475.	1475	334	337.	340	1070	1076.	1081
Cecil	381	381.	381	562	562.	562	144	180.	220	172	231.	298
Total (3/3)	22509	23283.	24454	36192	37341.	39130	11265	13005.	15716	16018	19243.	24329
Java-b	87	87.	87	160	160.	160	19	20.	21	19	20.	21
Java-1.18	326	326.	326	657	657.	657	80	85.	90	135	144.	158
Java-1.22	2149	2149.	2149	4534	4534.	4534	527	687.	1019	1207	1673.	2545
Java-1.30	2663	2683.	2709	6684	7283.	7751	817	1055.	1613	2053	2801.	4114
Unidraw	12	12.	12	12	12.	12	1	2.	2	1	2.	2
Lov-obj-ed	1698	1698.	1698	2470	2470.	2470	861	1027.	1304	1133	1476.	1834
Geode	10086	10790.	11354	11797	12642.	13312	6341	6989.	7836	7321	8134.	9132
Eiffel	4403	4618.	5078	6663	7151.	8078	2268	2766.	3353	3183	3873.	4806
Laure	64	64.	64	100	100.	100	8	8.	8	8	8.	8
Self	623	623.	623	1475	1475.	1475	334	337.	340	1070	1075.	1081
Cecil	381	381.	381	562	562.	562	145	186.	220	179	243.	298
Total (4/3)	22508	23447.	24497	35130	37062.	39127	11401	13161.	15806	16309	19449.	23999
Java-b	87	87.	87	160	160.	160	19	21.	24	19	21.	24
Java-1.18	345	345.	345	673	673.	673	83	87.	95	135	146.	162
Java-1.22	1916	1981.	2578	4377	4461.	5235	609	759.	885	1643	1865.	2071
Java-1.30	2813	2959.	3237	7655	7911.	8017	743	967.	1339	1627	2694.	3820
Unidraw	12	12.	12	12	12.	12	1	2.	2	1	2.	2
Lov-obj-ed	1495	1510.	1542	2115	2145.	2169	930	1163.	1526	1213	1623.	2322
Geode	10732	11176.	12336	12599	13169.	14545	7612	8401.	8915	8749	9710.	10384
Eiffel	5401	5846.	6273	8528	10025.	11556	2570	2973.	3530	3609	4321.	5306
Laure	64	64.	64	100	100.	100	8	8.	8	8	8.	8
Self	623	623.	623	1475	1475.	1475	335	382.	406	1071	1972.	3148
Cecil	385	385.	385	549	549.	549	160	190.	242	205	248.	334
Total (2/3)	23889	25004.	27498	38259	40696.	44507	13070	14953.	16972	18280	22610.	27581
Java-a	13	13.	14	13	13.	14	0	0.	0	0	0.	0
Java-b	78	95.	141	105	145.	240	10	18.	48	10	19.	54
Java-1.18	277	356.	671	394	681.	2007	73	108.	171	88	150.	289
Java-1.22	1632	2433.	5474	3878	6701.	15393	524	956.	2315	1191	2501.	5509
Java-1.30	2263	3079.	6893	5362	8878.	20391	832	1311.	2879	1718	3413.	7333
Unidraw	12	13.	14	12	13.	14	1	2.	3	1	2.	3
Lov-obj-ed	1358	1662.	2148	1899	2397.	3199	774	1029.	1345	1065	1444.	1958
Geode	10712	13119.	14960	13068	15834.	18847	5818	8938.	12169	6898	10622.	14129
Eiffel	4698	5794.	6785	6812	9644.	13122	2352	3083.	3965	3250	4731.	6875
Laure	60	73.	102	82	154.	299	7	15.	33	7	19.	75
Self	677	761.	962	1765	4068.	4306	362	470.	624	1199	3466.	5396
Cecil	392	479.	713	530	809.	2099	113	179.	343	136	246.	663
Total (0/0)	22175	27881.	38880	33923	49341.	79934	10866	16107.	23895	15563	26612.	42284

FIG. 8.7 – Statistiques de nombres de trous dans les versions stochastiques du schéma d'algorithme, avec des heuristiques décroissantes de haut en bas.

Système	identifiant local					pseudo-matrice			bits	identifiant global			
	bits	matrice			1octet					matrice			2octets
		bit	mot	octet		bit	mot	octet		bit	mot	octet	
Java-a	4	34	34	68	68	34	34	68	6	68	68	68	136
Java-b	5	315	315	420	368	210	210	420	7	420	420	420	735
Java-1.18	7	1196	1196	1196	1196	598	598	1196	9	1495	1794	2392	2392
Java-1.22	8	4830	4830	4830	4106	2898	2898	4830	10	5796	5796	8694	8694
Java-1.30	8	5700	5700	5700	5415	4560	4560	5700	11	7980	11400	11400	11400
IDL	2	11	11	22	20	11	11	22	4	11	11	22	44
Unidraw	3	25	25	75	57	25	25	75	5	50	50	75	125
Lov-obj-ed	6	1355	1355	1626	1626	813	813	1626	9	1897	2168	3252	3252
Geode	7	10879	12857	12857	12363	6923	6923	12857	10	15824	16813	24725	24725
Eiffel	8	9100	9100	9100	8873	4550	5460	9100	10	11830	11830	18200	18200
Laure	4	114	114	228	228	114	114	228	6	171	228	228	456
Self	5	1078	1078	1540	1502	462	462	1540	8	1540	1540	1540	3080
Cecil	6	1224	1224	1530	1530	918	918	1530	9	1836	2142	3060	3060
Total (3/3)	8	36682	37970	42007	40295	24291	25280	42007	NIL	52665	58007	78846	81094
Java-a	7	452	452	452	452	226	226	452	8	452	452	452	904
Java-b	8	2416	2416	2416	2114	1208	1812	2416	10	3020	3020	4228	4228
Java-1.18	10	8520	10224	13632	6816	5112	5112	8520	11	10224	13632	13632	13632
Java-1.22	11	26034	39051	39051	22780	17356	17356	26034	13	30373	39051	39051	39051
Java-1.30	11	38066	54380	54380	32628	21752	27190	38066	13	43504	54380	54380	54380
IDL	5	134	134	201	151	67	67	201	7	134	201	201	335
Unidraw	8	1842	1842	1842	1535	1228	1228	1842	10	2456	2456	3070	3070
Lov-obj-ed	7	2616	2616	2616	2616	1744	1744	2616	9	3052	3488	5232	5232
Geode	8	17134	17134	17134	16475	9226	10544	17134	11	23724	32950	32950	32950
Eiffel	10	25987	25987	39980	19491	11994	11994	21989	11	27986	39980	39980	39980
Laure	6	879	1172	1172	1172	586	586	1172	9	1465	1758	2344	2344
Self	9	21624	25228	37842	18471	7208	9010	19822	11	27030	37842	37842	37842
Cecil	8	5592	5592	5592	5359	3728	3728	5592	10	7456	7456	11184	11184
Total (3/3)	11	163155	198230	228748	146246	87703	95064	152659	NIL	197720	250422	258916	259502

FIG. 8.8 – Statistiques d'occupation mémoire de la coloration unidirectionnelle, dans une implémentation par matrice ou pseudo-matrice : l'unité est le mot de 32 bits. Les colonnes indiquent l'alignement — pas d'alignement (bit), pas de chevauchement de mots (mot), ou à l'octet (octet) — ou une implémentation uniforme en simple ou double octet.

Système	identifiant local							identifiant global				
	bits	vecteur			1octet	pseudo-vecteur			bits	vecteur			2octets
		bit	mot	octet		bit	mot	octet		bit	mot	octet	
Java-a	4	60	60	78	51	59	59	78	6	59	59	61	84
Java-b	5	212	212	245	171	197	197	245	7	212	212	212	293
Java-1.18	6	688	696	720	515	635	635	720	9	693	707	886	886
Java-1.22	8	2505	2505	2505	1850	2424	2461	2505	10	2494	2494	3234	3234
Java-1.30	8	3064	3064	3064	2287	2740	2740	3064	11	3613	4025	4025	4025
IDL	2	19	19	25	18	19	19	25	4	19	19	19	28
Unidraw	3	43	43	60	41	40	40	60	5	41	41	47	69
Lov-obj-ed	5	934	934	1205	1008	788	789	1205	9	1220	1375	1883	1883
Geode	6	5511	5659	6865	6115	5000	5122	6865	10	7825	8059	11754	11754
Eiffel	8	4273	4273	4273	3512	2957	3005	4273	10	4500	4601	6544	6544
Laure	3	125	125	174	133	116	116	174	6	129	129	148	232
Self	5	721	767	1007	926	374	374	1007	8	943	943	943	1754
Cecil	5	862	862	1005	796	781	781	1005	9	1011	1055	1436	1436
Total (3/3)	8	19017	19219	21226	17423	16130	16338	21226	13	22759	23719	31192	32222
Java-a	7	480	480	480	348	443	443	480	8	445	445	445	568
Java-b	8	1453	1453	1453	1008	1392	1392	1453	10	1474	1474	1792	1792
Java-1.18	10	4503	4512	5427	2843	3880	3880	4723	11	4746	4975	4975	4975
Java-1.22	11	12851	13925	13925	9795	10486	10752	12963	13	12576	12999	12999	12999
Java-1.30	11	16503	18016	18016	13635	13482	13555	16585	13	16060	16390	16390	16390
IDL	5	147	147	161	114	131	131	161	7	136	136	136	199
Unidraw	8	1419	1419	1419	925	1266	1266	1419	10	1330	1330	1668	1668
Lov-obj-ed	6	1537	1620	1893	1557	1359	1458	1893	9	1911	2133	2910	2910
Geode	8	8387	8387	8387	7349	6418	6541	8387	11	10658	14115	14115	14115
Eiffel	10	9523	9847	12591	6449	6141	6319	8796	11	9871	11913	11913	11913
Laure	7	946	946	946	747	674	674	946	9	943	959	1310	1310
Self	9	18196	21195	30146	15256	6744	6887	17627	11	21430	29012	29012	29012
Cecil	8	2727	2727	2727	2046	2421	2435	2727	10	2767	2767	3690	3690
Total (3/3)	11	78672	84674	97571	62072	54837	55733	78160	15	84347	98648	101355	101541

FIG. 8.9 – Statistiques d'occupation mémoire de la coloration bidirectionnelle, dans une implémentation par vecteur ou pseudo-vecteur : l'unité est le mot de 32 bits. Les colonnes indiquent l'alignement — pas d'alignement (bit), pas de chevauchement de mots (mot), ou à l'octet (octet) — ou une implémentation uniforme en simple ou double octet.

Système	matrices id. global	id. unif.	pseudo-m.	vecteurs id. global	id. unif.	pseudo-v.
non aligné	52665	36682	24291	22759	19017	16130
aligné au mot	58007	37970	25280	23719	19219	16338
à l'octet	78846	42007	42007	31192	21226	21226
simple octet		40295			17423	
double octet	81094			32222		
non aligné	197720	163155	87703	84347	78672	54837
aligné au mot	250422	198230	95064	98648	84674	55733
à l'octet	258916	228748	152659	101355	97571	78160
simple octet		146246			62072	
double octet	259502			101541		

FIG. 8.10 – Statistiques d'occupation mémoire, suivant une implémentation par matrice, pseudo-matrice, vecteur et pseudo-vecteurs, et suivant les alignements et les unités d'implémentation (taille en mots)

Système	cœur classes	tables	total classes	tables	mixte 2 octets dyn.	gain	stat.	mixte 1 octet dyn.	gain	stat.
Java-a	34	84	226	568	310	0.55	197	277	0.80	164
Java-b	105	293	604	1792	897	0.50	595	775	0.77	473
Java-1.18	299	886	1704	4975	2590	0.52	1738	2219	0.78	1367
Java-1.22	966	3234	4339	12999	7573	0.58	5404	6189	0.63	4020
Java-1.30	1140	4025	5438	16390	9463	0.58	6744	7725	0.57	5006
IDL	11	28	67	199	95	0.48	62	85	0.75	52
Unidraw	25	69	614	1668	683	0.41	376	655	0.71	348
Lov-obj-ed	271	1883	436	2910	2319	0.80	2101	1444	0.93	1226
Geode	989	11754	1318	14115	13072	0.93	12413	7433	1.01	6774
Eiffel	910	6544	1999	11913	8543	0.72	7544	5511	0.85	4512
Laure	57	232	293	1310	525	0.40	379	426	0.57	280
Self	154	1754	1802	29012	3556	0.12	2655	2728	0.18	1827
Cecil	306	1436	932	3690	2368	0.64	1902	1728	0.84	1262
Total (3/3)	5267	32222	19772	101541	51994	0.51	42108	37195	0.60	27309

FIG. 8.11 – Bilan de la technique mixte, avec un codage global et uniforme sur 2 octets

Système	nombre de		coloration (bits)				tech. mixte (bits)			
	classes	couleurs	total	/c	/c^2	/$\mid \preceq \mid$	total	/c	/c^2	/$\mid \preceq \mid$
Java-a	34	135	457	13.4	0.40	3.4				
Java-b	105	473	2163	20.6	0.20	4.8				
Java-1.18	299	1462	8601	28.8	0.10	6.3				
Java-1.22	966	5465	39113	40.5	0.04	8.0				
Java-1.30	1140	6867	49102	43.1	0.04	8.3				
IDL	11	48	103	9.4	0.85	2.1				
Unidraw	25	113	332	13.3	0.53	3.0				
Lov-obj-ed	271	3490	15228	56.2	0.21	6.2				
Geode	989	22480	121870	123.2	0.12	7.9				
Eiffel	910	12225	63478	69.8	0.08	6.7				
Laure	57	416	1316	23.1	0.41	3.2				
Self	154	3394	7012	45.5	0.30	2.3				
Cecil	306	2569	13054	42.7	0.14	5.6				
Total	5267	59137	321829	61.1	0.01	7.0				
Java-a	226	940	5586	24.7	0.11	5.9	4073	18.0	0.08	4.3
Java-b	604	2822	19047	31.5	0.05	6.8	14243	23.6	0.04	5.1
Java-1.18	1704	7575	62345	36.6	0.02	8.4	46089	27.0	0.02	6.2
Java-1.22	4339	20393	182996	42.2	0.01	9.7	151927	35.0	0.01	8.0
Java-1.30	5438	25961	240428	44.2	0.01	10.1	190490	35.0	0.01	8.0
IDL	67	320	1313	19.6	0.29	4.1	1041	15.5	0.23	3.3
Unidraw	614	2469	16878	27.5	0.04	6.8	12612	20.5	0.03	5.1
Lov-obj-ed	436	5356	26875	61.6	0.14	7.2	23076	52.9	0.12	6.2
Geode	1318	26759	156053	118.4	0.09	8.5	150866	114.5	0.09	8.2
Eiffel	1999	21522	126725	63.4	0.03	7.2	107456	53.8	0.03	6.1
Laure	293	2400	11202	38.2	0.13	4.7	6590	22.5	0.08	2.8
Self	1802	56716	149234	82.8	0.05	2.7	46656	25.9	0.01	0.8
Cecil	932	6319	41865	44.9	0.05	6.9	31694	34.0	0.04	5.3
Total	19772	179552	1040547	52.6	0.00	6.5	914989	46.3	0.00	5.7

FIG. 8.12 – Performance du codage d'ordre

Système	nb classes	bicolor	mixte	BPE	NHE	PQE	CPQE
Java-a	226	24.7	18.0	24.	19.	1.	8.
Java-b	604	31.5	23.6	–	–	–	–
Java-1.18	1704	36.6	27.0	–	39.	25.	9.
Java-1.22	4339	42.2	35.0	–	62.	36.	10.
Java-1.30	5438	44.2	35.0	–	65.	41.	18.
IDL	67	19.6	15.5	24.	17.	0.	8.
Unidraw	614	27.5	20.5	55.	30.	2.	8.
Lov-obj-ed	436	61.6	52.9	86.	57.	42.	21.
Geode	1318	118.4	114.5	149.	95.	80.	39.
Eiffel	1999	63.4	53.8	149.	72.	65.	27.
Laure	293	38.2	22.5	55.	23.	6.	8.
Self	1802	82.8	25.9	118.	53.	39.	9.
Cecil	932	44.9	34.0	–	–	–	–

FIG. 8.13 – Comparaisons avec d'autres techniques

Lorsque le test est toujours statique, on dira que le codage est statique. Diverses optimisations sont alors possibles :

- il est inutile de stocker n_2, de même que la couleur k des classes ou toute information sur le codage de cette couleur ;
- lorsque la hiérarchie a une racine (unique classe sans super-classes), il est inutile de faire figurer cette classe dans la table de couleurs ;
- lorsque la table de couleurs est pointée par une structure de données associée à la classe, l'entrée de la couleur la plus nombreuse peut être mise dans cette structure de données (*inlining)* et non pas dans la table de couleurs, ce qui économise une instruction de chargement (temps et espace) pour les classes de cette couleur.

La structure de données des classes contient alors un mot pour n_1 et l'entrée de la couleur *inlinée*, et un pointeur sur la table de couleurs.

Lorsque le test peut être dynamique, la classe cible étant calculée à partir d'un objet ou de la classe de cet objet, par exemple en cas de redéfinition covariante des attributs,

- il faut stocker n_2, de même que la couleur k des classes et toute information sur son codage (masque ou position et longueur du champ de bits) ;
- même lorsque la hiérarchie a une racine, il faut faire figurer cette classe dans la table de couleurs.

Au total, la structure de données des classes contient alors un mot pour n_1 et n_2, un mot ou une partie de mot pour la couleur, et le pointeur sur la table de couleurs.

Dans les deux cas, toute la table de couleurs peut être *inlinée* dans la structure de données des classes, si elle est implémentée par matrice ou si des tests de débordement sont effectués, ce qui nécessite d'inclure la longueur des tables.

Dans le test $D \preceq C$ statique, les informations comme $n_1(C)$, $n_2(C)$ ou la couleur $k(C)$ de la classe cible C sont des valeurs immédiates dans le code qui exécute le test. La table de couleurs T_D et $n_1(D)$ sont extraits de la structure de données de la classe source D.

Dans le test dynamique, toutes les informations comme $n_1(C)$, $n_2(C)$ ou la couleur $k(C)$ de la classe cible sont extraites de sa structure de données.

8.4.2 Comparaisons

La figure 8.13 compare les résultats obtenus avec différentes techniques :

- *bit-packed encoding* (BPE) [Vitek *et al.*, 1997] correspond à la coloration de classes par pseudo-matrice alignée au mot ;
- *near optimal hierarchy encoding* (NHE) [Krall *et al.*, 1997] ;
- *PQ-encoding* (PQE) et *Coalesced PQ-encoding* (CPQE) [Zibin et Gil, 2001].

Ces comparaisons sont peu significatives dans la mesure où seules les structures de données sont considérées, et pas le code, dans le cadre d'une implémentation supposée des classes dont seul le surcoût est considéré, ce qui permet d'avoir une valeur nulle pour certains codages.

Ce constat conduit à proposer 2 façons de comparer la taille des codages, suivant qu'ils sont statiques ou dynamiques :

- codage dynamique : on compte la taille de la totalité des structures de données plus la taille de la fonction qui réalise le test dynamique à partir des structures de données des 2 classes à comparer ;
- codage statique : on compte la taille de la totalité des structures de données nécessaires pour réaliser les tests statiques, plus la taille du code nécessaire pour réaliser les tests statiques pour chaque classe.

Troisième partie

Coloration de méthodes ou d’attributs

Chapitre 9

Coloration de méthodes ou d'attributs : résultats

La coloration de méthodes ou d'attributs est une généralisation de la coloration de classes : en effet, la coloration de classes est une coloration de méthodes pour laquelle il y a une et une seule méthode introduite dans chaque classe.

9.1 Propriétés

La coloration des méthodes et des attributs se ramène, quand elle est parfaite et en typage statique uniquement[1], à la coloration de classes. On peut d'abord noter que deux méthodes (resp. attributs) sont en conflit si elles appartiennent à la même classe, sans que les classes qui les introduisent soient sous-classes l'une de l'autre : autrement dit, deux méthodes sont en conflit si et seulement si les classes qui les introduisent sont en conflit.

Proposition 9.1 (Coloration de méthodes parfaite) *L'existence d'une coloration parfaite des classes entraîne celle d'une coloration parfaite des méthodes (resp. des attributs).*

La réciproque n'est vraie que si chaque classe introduit au moins une méthode (resp. un attribut).

Par conséquent, l'existence d'une coloration parfaite ne dépend pas de la répartition des méthodes dans les classes, mais uniquement de la structure de la hiérarchie.

La construction de la coloration est immédiate : dans chaque classe, les méthodes (resp. attributs) sont placés dans l'ordre des couleurs des classes qui les introduisent. Les classes de même signe n'étant pas en conflit, leurs méthodes non plus.

Il n'y a aucun problème parce qu'il n'y a pas de trous. Mais la présence de trous rend cette technique difficile à mettre en œuvre, puisque chaque trou représente un conflit avec plusieurs classes dont le nombre de méthodes (resp. attributs) varie. La solution de facilité consiste à remplacer chaque trou de la coloration de classes par le maximum des nombres de méthodes des différentes classes en conflit de la couleur du trou : on obtient donc une coloration de méthodes correcte mais non optimale car les trous auraient pu être en partie bouchés, dans les sous-classes, au fur et à mesure de la raréfaction du nombre de classes en conflit.

En ce qui concerne l'implémentation, la coloration de méthodes est globalement simplifiée par le fait que toutes ses entrées sont de taille uniforme (des adresses) et que le typage statique rend inutile le test de débordement de table. Ce dernier point reste valable pour la coloration d'attributs ou pour l'implémentation des attributs par décalage dans la coloration de classe : dans les deux cas, l'implémentation non uniforme semble une complication inutile.

[1] Pour éviter la surcharge propre au typage dynamique, qui fait de SMALLTALK un contre-exemple immédiat de la proposition 9.1.

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	381 381. 381	562 562. 562	144 180. 220	172 231. 298
4	3	381 381. 381	562 562. 562	145 186. 220	179 243. 298
4	2	437 437. 437	623 623. 623	153 161. 168	184 189. 200
3	2	427 427. 427	610 610. 610	154 169. 229	183 207. 314
2	2	438 438. 438	624 624. 624	129 183. 229	147 238. 324
0	2	406 482. 676	553 829. 1882	115 159. 366	133 217. 728
3	1	636 636. 636	1306 1306. 1306	173 173. 173	274 274. 274
0	0	392 479. 713	530 809. 2099	113 179. 343	136 246. 663
0	1	405 479. 676	549 807. 1688	115 174. 303	131 236. 532
1	2	665 665. 665	1451 1451. 1451	174 198. 225	279 334. 399
2	1	636 638. 640	1306 1308. 1310	140 194. 258	211 314. 424
1	1	665 665. 665	1451 1451. 1451	171 195. 230	272 325. 395
2	0	438 449. 459	624 649. 673	120 203. 292	146 264. 709
1	0	665 665. 665	1451 1451. 1451	144 230. 308	214 400. 556

FIG. 9.1 – Comparaison des heuristiques pour Cecil

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	10086 10655. 11354	11797 12482. 13312	6289 6964. 7854	7337 8117. 9150
4	3	10086 10790. 11354	11797 12642. 13312	6341 6989. 7836	7321 8134. 9132
4	2	10559 10575. 10590	12175 12190. 12204	6291 7139. 8053	7239 8152. 9100
3	2	10525 10560. 10575	12132 12173. 12189	5945 6796. 8136	6843 7813. 9327
2	2	10859 11482. 12289	12500 13225. 14116	7142 7587. 8291	8121 8691. 9651
0	2	10779 13079. 15334	12810 15808. 18709	6055 8984. 12457	7011 10641. 14662
3	1	10858 10858. 10858	14156 14156. 14156	6571 6769. 7024	7766 8490. 8842
0	0	10712 13119. 14960	13068 15834. 18847	5818 8938. 12169	6898 10622. 14129
0	1	10426 12941. 14741	12512 15617. 18575	5868 8991. 11818	6931 10645. 14452
1	2	11034 11048. 11112	14476 14490. 14554	6106 7172. 8521	7678 9222. 10875
2	1	10793 10795. 10797	14088 14090. 14092	5997 7340. 8673	7689 9323. 11117
1	1	11051 11226. 11497	14363 14569. 14894	6110 7383. 8319	7797 9309. 10725
2	0	10369 11283. 12222	12063 13064. 14363	5076 9386. 11879	5945 10898. 14085
1	0	11040 11397. 12082	14309 14777. 15642	5943 7710. 8948	7614 9714. 11215

FIG. 9.2 – Comparaison des heuristiques pour Geode

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	623 623. 623	1475 1475. 1475	334 337. 340	1070 1076. 1081
4	3	623 623. 623	1475 1475. 1475	334 337. 340	1070 1075. 1081
4	2	656 656. 656	4000 4000. 4000	330 393. 423	1323 2860. 3501
3	2	656 656. 656	4000 4000. 4000	327 332. 337	1318 1325. 1330
2	2	656 656. 656	4000 4000. 4000	414 419. 422	3492 3497. 3500
0	2	675 759. 891	3513 4091. 4235	350 468. 616	610 3442. 5341
3	1	688 688. 688	1542 1542. 1542	340 342. 343	1076 1081. 1084
0	0	677 761. 962	1765 4068. 4306	362 470. 624	1199 3466. 5396
0	1	668 758. 929	2872 4071. 4273	380 476. 597	1392 3673. 5364
1	2	621 621. 621	1464 1464. 1464	334 338. 342	1070 1077. 1083
2	1	676 676. 676	1507 1507. 1507	364 367. 371	1094 1127. 1168
1	1	621 704. 771	1274 2753. 3640	334 390. 458	875 1894. 3296
2	0	656 656. 656	4000 4000. 4000	404 422. 506	2849 3250. 5001
1	0	621 700. 771	1283 2660. 3640	334 398. 463	862 2025. 3302

FIG. 9.3 – Comparaison des heuristiques pour Self

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	64 64. 64	100 100. 100	8 8. 8	8 8. 8
4	3	64 64. 64	100 100. 100	8 8. 8	8 8. 8
4	2	75 75. 75	208 208. 208	10 10. 10	17 17. 17
3	2	75 75. 75	208 208. 208	10 10. 10	17 17. 17
2	2	75 75. 75	208 208. 208	10 10. 10	17 17. 17
0	2	60 71. 102	87 155. 341	7 12. 28	7 15. 63
3	1	63 63. 63	95 95. 95	8 8. 8	8 8. 8
0	0	60 73. 102	82 154. 299	7 15. 33	7 19. 75
0	1	59 72. 101	73 153. 314	7 14. 34	7 19. 65
1	2	61 61. 61	83 83. 83	8 8. 8	8 8. 8
2	1	63 63. 63	95 95. 95	8 10. 12	8 11. 15
1	1	61 62. 64	79 92. 108	8 9. 12	8 11. 18
2	0	75 75. 75	208 208. 208	10 15. 25	17 23. 41
1	0	61 62. 64	79 92. 108	8 9. 12	8 11. 18

FIG. 9.4 – Comparaison des heuristiques pour Laure

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	4404 4609. 5077	6663 7126. 8082	2240 2724. 3413	3090 3830. 5304
4	3	4403 4618. 5078	6663 7151. 8078	2268 2766. 3353	3183 3873. 4806
4	2	5165 5293. 5438	9199 9385. 9645	2285 2517. 2961	3225 3567. 4095
3	2	5189 5318. 5443	9172 9419. 9735	3030 3448. 4352	4754 5733. 7469
2	2	4286 4451. 4625	6262 6562. 6788	2495 3274. 4067	3645 5175. 6910
0	2	4974 5705. 6644	7262 9417. 11715	2384 2981. 3780	3170 4534. 6711
3	1	7500 7501. 7503	14641 14643. 14646	3519 3781. 4302	5697 6380. 7640
0	0	4698 5794. 6785	6812 9644. 13122	2352 3083. 3965	3250 4731. 6875
0	1	4841 5735. 6884	6822 9522. 12757	2329 2998. 3864	3170 4533. 6512
1	2	7623 7950. 8252	14992 15859. 16622	3412 3863. 4322	5905 7195. 8104
2	1	7501 7503. 7506	14652 14655. 14659	3409 3831. 4456	5989 7017. 7983
1	1	6976 7697. 8215	13041 15076. 16432	3057 3871. 4341	5593 7123. 8259
2	0	4426 4784. 5235	6375 7007. 7762	2670 3408. 4374	3962 5412. 7614
1	0	6994 7986. 8616	13049 15830. 17296	3282 4103. 4871	5809 7590. 9156

FIG. 9.5 – Comparaison des heuristiques pour Eiffel

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	1698 1698. 1698	2470 2470. 2470	857 1015. 1299	1122 1450. 1831
4	3	1698 1698. 1698	2470 2470. 2470	861 1027. 1304	1133 1476. 1834
4	2	1451 1518. 1559	2039 2147. 2210	1085 1130. 1222	1510 1581. 1738
3	2	1515 1515. 1515	2152 2154. 2154	1077 1129. 1220	1511 1591. 1735
2	2	1470 1655. 1893	2069 2314. 2641	899 1078. 1200	1147 1570. 1805
0	2	1354 1637. 2193	1845 2354. 3291	734 991. 1345	970 1388. 2070
3	1	1784 1784. 1784	2636 2636. 2636	923 1019. 1128	1222 1428. 1637
0	0	1358 1662. 2148	1899 2397. 3199	774 1029. 1345	1065 1444. 1958
0	1	1318 1598. 2138	1887 2288. 3196	711 1005. 1410	896 1408. 2142
1	2	1751 1752. 1755	2511 2512. 2515	1096 1188. 1385	1504 1626. 1830
2	1	1784 1784. 1784	2636 2636. 2636	938 1037. 1338	1229 1406. 1973
1	1	1688 1730. 1751	2356 2459. 2522	940 1192. 1402	1272 1625. 1853
2	0	1414 1621. 1811	2001 2336. 2703	899 1181. 1358	1150 1654. 1977
1	0	1688 1965. 2267	2356 2773. 3219	1008 1299. 1583	1237 1789. 2095

FIG. 9.6 – Comparaison des heuristiques pour Lov-obj-ed

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	326 326. 326	657 657. 657	80 87. 95	133 149. 168
4	3	326 326. 326	657 657. 657	80 85. 90	135 144. 158
4	2	331 358. 382	665 708. 762	82 98. 129	142 174. 278
3	2	331 334. 336	665 687. 715	74 90. 102	105 152. 180
2	2	315 362. 407	635 713. 778	84 98. 111	141 172. 256
0	2	267 347. 632	449 657. 1732	69 100. 166	84 140. 305
3	1	315 315. 315	672 672. 672	88 104. 120	128 179. 224
0	0	277 356. 671	394 681. 2007	73 108. 171	88 150. 289
0	1	275 356. 528	404 669. 1386	67 107. 210	85 147. 369
1	2	313 345. 387	671 706. 759	81 90. 102	109 143. 178
2	1	325 325. 325	675 675. 675	82 104. 127	119 181. 219
1	1	313 362. 483	669 805. 1215	87 114. 157	116 185. 268
2	0	317 359. 417	641 715. 841	84 113. 140	121 185. 262
1	0	313 378. 483	662 832. 1232	91 119. 220	116 198. 385

FIG. 9.7 – Comparaison des heuristiques pour Java-1.18

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	2149 2149. 2149	4534 4534. 4534	515 625. 981	1163 1569. 2520
4	3	2149 2149. 2149	4534 4534. 4534	527 687. 1019	1207 1673. 2545
4	2	2268 2284. 2378	4450 4495. 4727	854 1049. 1354	1419 1686. 2117
3	2	2277 2292. 2385	4484 4531. 4755	982 1035. 1080	1694 1788. 1869
2	2	2267 2271. 2284	4434 4439. 4452	720 831. 936	1225 1469. 1707
0	2	1600 2415. 5186	3860 6640. 14706	554 923. 2364	1145 2426. 5458
3	1	2724 2724. 2724	7454 7454. 7454	955 1074. 1234	2593 2909. 3499
0	0	1632 2433. 5474	3878 6701. 15393	524 956. 2315	1191 2501. 5509
0	1	1566 2416. 4679	3657 6662. 13077	563 975. 2385	1072 2490. 5799
1	2	2876 2876. 2876	7900 7900. 7900	1063 1364. 1619	3590 4394. 5182
2	1	2722 2723. 2734	7444 7453. 7654	1052 1248. 1405	2880 3789. 4820
1	1	2741 3741. 6214	7500 10694. 18603	1086 1672. 2839	3334 4910. 7854
2	0	2167 2269. 2418	4054 4554. 4804	705 1003. 1341	1414 2085. 2776
1	0	2752 4078. 6262	7628 11759. 18665	1022 1755. 2986	3324 5051. 7864

FIG. 9.8 – Comparaison des heuristiques pour Java-1.22

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	2663 2664. 2667	7746 7747. 7750	778 1044. 1483	1903 2792. 3946
4	3	2663 2683. 2709	6684 7283. 7751	817 1055. 1613	2053 2801. 4114
4	2	1929 1941. 1957	5211 5232. 5255	723 1049. 1339	1920 2825. 3682
3	2	1929 1935. 1949	5211 5224. 5247	736 992. 1328	1560 2660. 4081
2	2	1927 1954. 2002	5289 5983. 6550	747 1088. 1420	1932 2895. 3606
0	2	2139 2975. 6027	5092 8523. 17943	793 1199. 2803	1707 3153. 6461
3	1	5327 5327. 5327	16055 16055. 16055	2203 2663. 3396	6308 7617. 9506
0	0	2263 3079. 6893	5362 8878. 20391	832 1311. 2879	1718 3413. 7333
0	1	2072 3137. 6441	5291 8994. 19195	746 1357. 2781	1671 3431. 7570
1	2	7712 7713. 7714	23715 23716. 23717	2357 2631. 2837	6710 7364. 7904
2	1	7713 7714. 7718	23708 23709. 23717	2339 2764. 3290	7463 8548. 9927
1	1	3675 6187. 7834	10731 18825. 24191	1586 2442. 3544	4881 7517. 10020
2	0	1928 2045. 2280	5117 6202. 7323	836 1094. 1517	1773 2529. 3722
1	0	3685 6273. 8452	10757 19177. 26149	1592 2613. 3811	4859 7683. 10400

FIG. 9.9 – Comparaison des heuristiques pour Java-1.30

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	13 13. 13	13 13. 13	0 0. 0	0 0. 0
4	3	13 13. 13	13 13. 13	0 0. 0	0 0. 0
4	2	13 13. 13	13 13. 13	0 0. 0	0 0. 0
3	2	13 13. 13	13 13. 13	0 0. 0	0 0. 0
2	2	13 13. 13	13 13. 13	0 0. 0	0 0. 0
0	2	13 13. 14	13 13. 14	0 0. 0	0 0. 0
3	1	13 13. 13	13 13. 13	0 0. 0	0 0. 0
0	0	13 13. 14	13 13. 14	0 0. 0	0 0. 0
0	1	13 13. 14	13 13. 14	0 0. 0	0 0. 0
1	2	13 13. 13	13 13. 13	0 0. 0	0 0. 0
2	1	13 13. 13	13 13. 13	0 0. 0	0 0. 0
1	1	13 13. 13	13 13. 13	0 0. 0	0 0. 0
2	0	13 13. 13	13 13. 13	0 0. 0	0 0. 0
1	0	13 13. 13	13 13. 13	0 0. 0	0 0. 0

FIG. 9.10 – Comparaison des heuristiques pour Java-a

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	87 87. 87	160 160. 160	19 20. 21	19 20. 21
4	3	87 87. 87	160 160. 160	19 20. 21	19 20. 21
4	2	83 83. 83	147 147. 147	20 20. 20	20 20. 20
3	2	83 83. 83	147 147. 147	20 20. 20	20 20. 20
2	2	83 83. 83	147 147. 147	20 20. 20	20 20. 20
0	2	76 93. 126	105 143. 201	9 15. 23	9 16. 40
3	1	91 91. 91	164 164. 164	12 14. 19	12 14. 19
0	0	78 95. 141	105 145. 240	10 18. 48	10 19. 54
0	1	76 94. 123	107 145. 200	9 18. 39	9 19. 52
1	2	91 91. 91	164 164. 164	12 14. 16	12 14. 16
2	1	91 91. 91	164 164. 164	12 18. 27	12 26. 43
1	1	83 97. 121	128 169. 213	11 17. 27	11 22. 46
2	0	83 83. 83	147 147. 147	20 22. 25	20 22. 25
1	0	83 97. 121	131 171. 239	11 18. 40	11 25. 59

FIG. 9.11 – Comparaison des heuristiques pour Java-b

heuristiques		coloration unidir.		coloration bidir.	
		cœur	total	cœur	total
3	3	12 12. 12	12 12. 12	1 1. 2	1 1. 2
4	3	12 12. 12	12 12. 12	1 2. 2	1 2. 2
4	2	12 12. 12	12 12. 12	1 2. 2	1 2. 2
3	2	12 12. 12	12 12. 12	1 1. 2	1 1. 2
2	2	12 12. 12	12 12. 12	1 1. 2	1 1. 2
0	2	12 13. 14	12 13. 14	1 2. 2	1 2. 2
3	1	12 12. 12	12 12. 12	1 2. 2	1 2. 2
0	0	12 13. 14	12 13. 14	1 2. 3	1 2. 3
0	1	12 12. 14	12 12. 14	1 2. 3	1 2. 3
1	2	12 12. 12	12 12. 12	1 2. 2	1 2. 2
2	1	12 12. 12	12 12. 12	1 1. 2	1 1. 2
1	1	12 12. 12	12 12. 12	1 2. 3	1 2. 3
2	0	12 12. 12	12 12. 12	1 2. 3	1 2. 3
1	0	12 12. 12	12 12. 12	1 2. 3	1 2. 3

FIG. 9.12 – Comparaison des heuristiques pour Unidraw

Quatrième partie

Annexes

Annexe A

Métrologie

A.1 Paramètres

Les paramètres à prendre en compte sont de deux ordres : il concernent la hiérarchie de classes et les propriétés (attributs et méthodes) définies dans ces classes.

A.1.1 Hiérarchie de classes

Sans prétendre à l'exhaustivité, la hiérarchie de classes peut se caractériser :
- par ses propriétés graphiques (sommets et arêtes) globales :
  - le nombre total de classes ;
  - le nombre total de relations de spécialisation directes entre 2 classes (taille du graphe de couverture) ;
  - le nombre total de relations de spécialisation directes ou indirectes entre 2 classes (taille de la fermeture transitive) ;
- par les caractéristiques propres à chaque classe
  - le nombre de super-classes directes ;
  - le nombre de super-classes directes ou indirectes ;
  - la profondeur (longueur de la plus grande chaîne de spécialisation dans les super-classes) ;
  - la largeur (taille de la plus grande antichaîne dans les super-classes) ;

  dont on peut considérer diverses statistiques, comme la moyenne ou le maximum, et dont on peut considérer aussi le dual, c'est-à-dire les sous-classes.
- par les caractéristiques locales à la structure, comme
  - le nombre de classes de même profondeur, ou ayant le même nombre de super-classes, etc.
  - les statistiques habituelles, comme le nombre maximum de classes de même profondeur et la profondeur de ce maximum ;
- par les caractéristiques d'un sous-ensemble de la structure : on s'intéressera en particulier au sous-graphe des classes en héritage multiple et à son complémentaire (figure 6.1), définis précisément ci-dessous. Les statistiques effectuées sur la hiérarchie toute entière peuvent s'appliquer aussi au cœur, à la couronne, ou aux attaches ;
- par les caractéristiques d'une structure invariante calculée sur la structure, comme le graphe de conflit pour la coloration ;
- par les caractéristiques d'une structure non invariante calculée sur la structure, par exemple en remplaçant profondeur par couleur dans la rubrique précédente.

L'étude de cette partition se justifie :
- *a posteriori*, par les statistiques qui montrent que le cœur représente, en moyenne, le quart des hiérarchies ;
- *a priori*, par la problématique de la coloration, dont les algorithmes reposent, pour l'essentiel, sur le cœur ;

– enfin, par les éventuelles techniques mixtes qui peuvent allier l'héritage multiple pour le cœur et l'héritage simple pour la couronne : on remarquera néanmoins que cette partition ne coïncide pas avec la distinction entre interfaces et classes en JAVA.

A.1.2 Attributs et méthodes

Pour les attributs (resp. méthodes), on peut considérer :
– des caractéristiques globales :
  – le nombre total de sélecteurs ou d'attributs ;
  – le nombre total de couples attributs (resp. méthodes)-classes valides ;
  – le nombre d'accès à un attribut, ou le nombre d'appels de méthodes dans le code ;
– des caractéristiques propres à chaque classe :
  – le nombre d'attributs (resp. méthodes) introduits par la classe ;
  – le nombre d'attributs (resp. méthodes) connus, c'est-à-dire introduits ou hérités, par la classe ;
  – pour les méthodes seulement, le nombre de méthodes définies dans la classe.
  dont on peut aussi considérer diverses statistiques, comme la moyenne ou le maximum.
– des caractéristiques propres à chaque méthode : nombre de classes de définition (en typage dynamique mais aussi pour les interfaces de JAVA), nombre d'accès à un attribut, d'appels de méthode, en distinguant les appels à `self` des autres, et avec les statistiques usuelles.

A.1.3 Code, appel de méthodes et accès aux attributs

Le code lui-même peut faire l'objet de statistiques :
– la taille du code lui-même, dont la mesure exacte est difficile : le nombre de lignes est peu significatif car il dépend trop des styles de programmation et d'indentation, et des langages ; le nombre d'instructions du progamme source est plus révélateur mais dépend toujours fortement des styles et des langages ; le nombre d'instructions du code objet est plus significatif, à condition de réussir à ne pas compter la mémoire statique ;
– le nombre d'appels de méthodes est un paramètre clé, que l'on peut décliner dans toutes ses variantes : appel sur `self`, sur un receveur typé par une classe ou une interface, à `super`, etc. ;
– même chose pour les accès aux attributs , en distinguant en plus lecture et écriture ;
– les diverses formes de *casting*, les cas de redéfinition covariante, etc. sont aussi à dénombrés.

Au lieu de mesures absolues, on peut aussi s'intéresser à des taux, comme le nombre d'instructions entre deux appels de méthodes. Toutes ces statistiques peuvent enfin être statiques, par analyse du code à la compilation, ou dynamiques, par trace d'exécutions : les paramètres résultants sont bien entendu incommensurables.

A.1.4 Efficacité temporelle et spatiale

Enfin, une dernière série de paramètres fournit des mesures de l'efficacité temporelle et spatiale :
– taille du code et de la mémoire statique ;
– taille de la mémoire dynamique ;
– temps d'exécution.

Ces paramètres peuvent être mesurés physiquement, calculés suivant un modèle ou simulés sur un simulateur de processeur. Dans tous les cas, ces mesures doivent être faites sur des *benchmarks* significatifs, aussi publiques que possibles.

Enfin, toutes ces mesures peuvent être faites sur le compilateur lui-même.

A.2 Statistiques

Les statistiques existantes sont partielles — seul le sous-ensemble de paramètres intéressant chaque auteur est cité — et trop peu nombreuses pour ne pas être biaisées par le choix du langage (les statistiques sur SMALLTALK sont innombrables mais de peu d'utilité ici). Deux biais ont été reconnus depuis longtemps :

	d'après [Ducournau, 1997]					
	NEXTSTEP[b]	SMALLTALK[a]			VISUALAGE[b]	FLAVORS[c]
c	310	**698**	774	1 370	3 241	563
s	2 707	**4 518**	5 086	7 685	17 342	2245[h]
m	4 324	**7 623**	8 540	14 057	37 058	
M	71 334	**164 839**	178 230	337 721	1 045 333	
$s_{\texttt{Object}}$	—	**110**	—	122	—	
M/c	230	**236**	230	272	322	
x	434[e]	**411**	399[e]	419	750[e]	
e	—	**31 280**[f]	35 042[g]	57 680[f]	152 059[f]	
C	3 174[d]	**—**	—	—	—	
s/c	8.7	**6.5**	6.6	5.6	5.4	4.0

[a] La version centrale d'après [Driesen et Hölzle, 1995], les deux autres (v4.0, en gras, et v4.1) utilisées pour nos propres tests. Ces trois versions diffèrent encore marginalement de celle utilisée dans [André et Royer, 1992] et [Ducournau *et al.*, 1992].
[b] D'après [Driesen et Hölzle, 1995].
[c] D'après [Pugh et Weddel, 1990].
[d] En K-octets, d'après [Vitek et Horspool, 1994].
[e] Valeur extrapolée à partir de l'évaluation de la coloration dans [Driesen et Hölzle, 1995].
[f] Valeur extrapolée à partir de celle de [Driesen *et al.*, 1995] en supposant que e/m est une constante.
[g] D'après [Driesen *et al.*, 1995].
[h] Il s'agit ici d'atributs et non de méthodes.

FIG. A.1 – Statistiques tournant pour l'essentiel autour de SMALLTALK.

– les grosses applications sont en général propriétaires et confidentielles ;
– les applications écrites en C++ le sont le plus souvent — ou l'on été longtemps — dans un style objet très emprunté (méthodes non virtuelles, héritage non virtuel, etc.[1]).

Différentes statistiques sont reprises et analysées ici. Ce sont souvent déjà des compilations et elles se partagent certaines données : les différences constatées d'une statistique à l'autre s'expliquent sans doute par des variations dans la définition des paramètres.

La plus grande partie de ces statistiques provient des sites suivants :
`http ://www.cs.ucsb.edu/labs/oocsb/classhierarchies/` et
`http ://www.cs.technion.ac.il/Labs/ssdl/thesis/on_going/2001/yoav/database.zip`.

A.2.1 Statistiques sur le nombre de classes et de méthodes

La figure A.1 est empruntée à [Ducournau, 1997][2] : elle est centrée sur SMALLTALK mais elle donne des informations intéressantes extrapolables sans doute dans d'autres contexte.

Dans un contexte assez cohérent, il apparaît ainsi que le nombre de méthodes par classe (ligne M/c) croît faiblement quand le nombre de classes augmente, passant de 230 à 322, alors que dans le même remps le nombre de méthodes introduites par chaque classe diminue, lui aussi faiblement, passant de 8.7 à 5.4. On constate ainsi que, si la coloration de classes revient à introduire une méthode supplémentaire par classe, le surcoût n'est pas tout à fait marginal : de 11.5 % à 18.5 %.

[Driesen, 1999] permet d'affiner ces données, par des statistiques de sous-ensembles de SMALLTALK, et de les compléter pour d'autres langages (figure A.2). Comme pour les statistiques précédentes, nous les complétons par s/c et M/c : notons que le paramètre M n'est pas toujours fiable puisqu'il ne prend évidemment pas en compte les méthodes héritées. C'est flagrant dans le cas de SMALLTALK où la classe `Object` introduit plus de 100 sélecteurs qui n'apparaissent pas ici dans les premières bibliothèques.

On peut à nouveau tirer des conclusions instructives de ces statistiques :
– comme remarqué dans [Ducournau, 1997] sur la figure A.1, tous les exemples de la figure A.2 exhibent un nombre moyen de méthodes par sélecteur (m/s) inférieur à 3, et la plupart sont même inférieurs à 2 : les seules exceptions sont un des sous-ensembles de SMALLTALK (Magnitude, 2.4),

[1] De même que les statistiques sur l'usage de l'héritage multiple en SMALLTALK sont formelles : le besoin n'en apparaît dans aucun programme significatif.
[2] Mais LaTeX refuse obstinément de me laisser le dire dans la légende.

d'après [Driesen, 1999]										
Système	Bibliothèque	c	s	M	m	p	s/c	M/c	m/s	γ
	Set	9	94	450	144	–	10.4	50	1.53	78
	Stream	16	126	1122	210	–	7.9	70	1.67	108
	Magnitude	18	240	1381	568	–	13.3	77	2.37	148
Parcplace	Collection	51	402	4926	805	–	7.9	97	2.00	151
SMALLTALK	VisualComp.	53	529	4253	875	–	10.0	80	1.65	133
	Obj. w/o meta.	383	4026	61755	6835	–	10.5	161	1.70	260
	Object (Pp1)	774	5086	178230	8540	–	6.6	230	1.68	403
	Parcplace2	1956	13474	608456	23720	–	6.9	311	1.76	546
Digitalk	ST/V 2.0	534	4482	154585	6853	–	8.3	289	1.53	672
SMALLTALK	ST/V 3.0	1356	10051	613654	17097	–	7.4	453	1.70	1078
IBM	SMALLTALK 2.0	2320	14009	485321	25994	–	6.0	209	1.86	653
SMALLTALK	VISUALAGE 2.0	3241	17342	1045333	37058	–	5.4	322	2.14	749
OBJECTIVE-C NEXTSTEP		310	2707	71334	4324	–	8.7	230	1.60	434
SELF	System 4.0	1801	10103	1038514	29411	1.02	5.6	577	2.91	962
C++	ET++	370	628	14816	1746	0.76	1.7	40	2.78	138
	Unidraw	613	1146	13387	3153	0.78	1.9	22	2.75	96
LOV	Lov+Obj-Ed	436	2901	36052	5007	1.78	6.7	83	1.73	286
(EIFFEL)	Geode	1318	6555	302717	14202	2.11	5.0	230	2.17	885

FIG. A.2 – Nombre de classes et de sélecteurs

VisualAge (2.1), SELF (2.9), C++ (2.8) et Geode (2.2) ; la liaison tardive ne porte donc, en pratique, que sur un sous-ensemble des appels de méthodes, pour discriminer un nombre de méthodes assez réduit : même les techniques d'analyse de types les plus grossières doivent donner de bons résultats.

- concernant SMALLTALK, on constate que le paramètre s/c « réel » est plus proche de 10 que de 6 : ce sont les méta-classes, qui doublent mécaniquement le nombre de classes sans introduire de nouvelles méthodes (1060 sélecteurs nouveaux pour 391 métaclasses), qui baissent ce paramètre ;
- le cas de SELF est un peu identique : c'est un langage de prototypes, donc sans classes, qui rétablit celles-ci implicitement avec des structures de factorisation des méthodes : le paramètre c est donc bien supérieur au réel nombre de « classes » ; par ailleurs, il n'y a pas de différence entre méthodes et attributs, ce qui augmente significativement les paramètres s et m, donc M ;
- concernant C++, il est manifeste que les deux exemples considérés font un très pauvre usage de la programmation par objets : il n'est donc pas possible de tenir compte de ces statistiques pour rejeter la coloration de classes au prétexte d'une augmentation de 59 % de la taille des tables de méthodes ;
- les statistiques d'EIFFEL sont certainement beaucoup plus proches d'un usage orthodoxe en typage statique ;
- la dernière colonne donne le nombre de couleurs nécessaire pour une coloration minimisant ce nombre : même dans le cas où le taux de remplissage ainsi obtenu ($M/\gamma c$) est faible — en C++ par exemple — c'est une très bonne estimation du nombre maximum de méthodes dans une classe. En effet, toutes les expériences de coloration de SMALLTALK le montrent et les bibliothèques C++ considérées sont quasiment en héritage simple.

A.2.2 Statistiques sur la structure des hiérarchies

Statistiques sur le cœur et la couronne

La figure A.3 donne des statistiques sur le nombre de classes et la taille de la fermeture transitive pour les différentes hiérarchies, en détaillant les parts respectives du cœur et de la couronne, ainsi que le nombre de classes ayant des conflits de coloration. La figure A.4 détaille ces premières données en y rajoutant des statistiques sur les arcs de transitivités, ainsi que sur la profondeur et le nombre de super-classes, directes et indirectes.

- Ces statistiques montrent deux groupes de hiérarchies : les hiérarchies EIFFEL ont un gros cœur, compris entre 46 et 75 % du total de la hiérarchie, alors que toutes les autres hiérarchies dépassent à

Système	nombre de classes						fermeture transitive		
	total	cœur	conflit	front.	couron.	taux	total	cœur	taux
Java-a	226	34	19	14	192	0.15	940	135	0.14
Java-b	604	105	61	41	499	0.17	2802	453	0.16
Java-1.18	1704	299	168	129	1405	0.18	7411	1371	0.18
Java-1.22	4339	966	444	518	3373	0.22	18941	4891	0.26
Java-1.30	5438	1140	546	588	4298	0.21	23772	5948	0.25
IDL	67	11	4	5	56	0.16	320	48	0.15
Unidraw	614	25	14	8	589	0.04	2468	112	0.05
Lov-obj-ed	436	271	159	110	165	0.62	3707	2447	0.66
Geode	1318	989	500	487	329	0.75	18442	15363	0.83
Eiffel	1999	910	593	314	1089	0.46	17559	9434	0.54
Laure	293	57	39	13	236	0.19	2392	408	0.17
Self	1802	154	104	49	1648	0.09	55639	3058	0.05
Cecil	932	306	167	132	626	0.33	6032	2352	0.39
Total	19772	5267	2818	2408	14505	0.27	160425	46020	0.29

FIG. A.3 – Statistiques du cœur et de la couronne : nombre de classes et fermeture transitive

peine 20 %, avec une moyenne plus proche de 15 %.

– Les arcs de transitivités apparaissent ainsi assez nombreux en EIFFEL ainsi que dans une hiérarchie JAVA. Il est probable que leur présence en EIFFEL traduit l'usage d'un héritage répété qu'il est difficile de mieux exploiter dans ces statistiques. En JAVA en revanche, ils proviennent du fait que la relation d'implémentation (*implements*) entre une classe et une interface, de même que la relation d'extension (*extends*) entre interfaces, peut être aussi bien redondante qu'absente : le sous-typage entre interfaces est en fait calculé.
– La structure de la hiérarchie SELF diffère sensiblement des autres. Le cœur est proportionnellement petit (9 % des classes), mais les statistiques sur les super-classes indirectes sont étonnantes : pour la couronne, leur nombre moyen est de presque 32, avec un maximum de 41. Toutes les classes ont donc en moyenne un très grand nombre de super-classes indirectes. Mais SELF est un langage peu représentatif : c'est un langage de prototypes, auxquels s'ajoutent des *maps* qui permettent de factoriser le code.
– Enfin, la hiérarchie FLAVORS est incomplète : seul le cœur est décrit dans une figure de [Pugh et Weddel, 1990], ainsi que le nombre de classes total. Une partie des statistiques n'est donc pas significative.

A.2.3 Statistiques de code

On trouve des statistiques de code essentiellement dans la littérature sur l'analyse de type [Bacon et Sweeney, 1996 ; Diwan *et al.*, 1996 ; Gil et Itai, 1998] et sur l'évaluation du coût de l'envoi de message suivant les architectures de processeurs [Driesen *et al.*, 1995 ; Driesen et Hölzle, 1996 ; Driesen, 1999].

Comme déjà remarqué, les statistiques prennent leurs données essentiellement dans des programmes C++, dont la plupart des méthodes ne sont pas virtuelles : certaines statistiques sont doubles, en considérant, ou non, les méthodes comme virtuelles [Driesen et Hölzle, 1996].

Les statistiques d'usage du *casting* descendant sont rares : [Wang et Smith, 2001] est une heureuse exception (figure A.5). Ces statistiques, appliquées à différents programmes JAVA, sont partielles. On voit qu'il y a, en moyenne, un *cast* tous les 70 lignes de code, mais la ligne est un étalon très élastique : le nombre d'appels de méthodes serait plus significatif. Cela étant, deux conclusions très générales peuvent être tirées de ces statistiques : le *casting* descendant est suffisamment fréquent pour qu'on s'occupe d'une implémentation efficace[3] et l'analyse de types permet d'éviter un grand nombre de vérifications dynamiques, ce qui en rend l'implémentation transparente lorsque l'invariance de référence (invariant 2.1) est vérifiée. En revanche, dans l'implémentation standard (invariant 2.4), s'assurer que le *cast* est sûr ne dispense que de la levée d'exception : le décalage reste à faire dynamiquement.

[3] Ce dont nous nous doutions déjà [Ducournau, 2001b].

Système	nb	super-cl. indirectes			profondeur		super-cl. directes			transitivité	
	classes	moy	max	total	moy	max	moy	max	total	max	total
Java-a	226	4.16	8	940	4.07	8	1.06	2	239	2	35
Java-b	604	4.64	14	2802	4.31	10	1.08	3	650	2	95
Java-1.18	1704	4.35	16	7411	4.04	10	1.10	11	1871	0	0
Java-1.22	4339	4.37	17	18941	3.81	10	1.19	14	5145	0	0
Java-1.30	5438	4.37	19	23772	3.80	10	1.17	14	6383	0	0
IDL	67	4.78	9	320	4.55	8	1.06	2	71	0	0
Unidraw	614	4.02	10	2468	3.99	10	1.01	2	623	0	0
Lov-obj-ed	436	8.50	24	3707	4.90	10	1.71	10	747	2	27
Geode	1318	13.99	50	18442	6.70	14	1.89	16	2486	6	369
Eiffel	1999	8.78	39	17559	6.76	18	1.28	9	2563	5	45
Laure	293	8.16	16	2392	7.57	12	1.07	3	314	1	1
Self	1802	30.88	41	55639	9.74	18	1.05	9	1889	0	0
Cecil	932	6.47	23	6032	5.11	13	1.21	3	1127	0	0
Total	19772	8.11	50	160425	5.02	18	1.22	16	24108	6	572

Système	nb	super-cl. indirectes			profondeur		super-cl. directes		
	classes	moy	max	total	moy	max	moy	max	total
Java-a	34	3.97	8	135	3.56	7	1.38	2	47
Java-b	105	4.31	14	453	3.50	7	1.44	3	151
Java-1.18	299	4.59	16	1371	3.54	7	1.56	11	466
Java-1.22	966	5.06	17	4891	3.64	9	1.83	14	1772
Java-1.30	1140	5.22	19	5948	3.73	9	1.83	14	2085
IDL	11	4.36	7	48	3.91	6	1.36	2	15
Unidraw	25	4.48	9	112	4.00	7	1.36	2	34
Lov-obj-ed	271	9.03	24	2447	4.87	10	2.15	10	582
Geode	989	15.53	50	15363	7.06	14	2.18	16	2157
Eiffel	910	10.37	39	9434	7.14	18	1.62	9	1474
Laure	57	7.16	16	408	6.23	11	1.37	3	78
Self	154	19.86	39	3058	6.40	14	1.56	9	241
Cecil	306	7.69	20	2352	5.42	10	1.64	3	501
Total	5267	8.74	50	46020	5.17	18	1.82	16	9603

Système	nb	super-cl. indirectes			profondeur		prof. racines			taille arbres	
	classes	moy	max	total	moy	max	nb	moy	max	moy	max
Java-a	192	4.19	8	805	4.16	8	15	3.53	6	13.80	100
Java-b	499	4.71	11	2349	4.48	10	26	3.00	6	20.19	252
Java-1.18	1405	4.30	14	6040	4.14	10	81	3.11	6	18.35	788
Java-1.22	3373	4.17	16	14050	3.86	10	241	3.56	7	15.00	1622
Java-1.30	4298	4.15	17	17824	3.81	10	280	3.62	7	16.35	2124
IDL	56	4.86	9	272	4.68	8	9	4.22	6	7.22	23
Unidraw	589	4.00	10	2356	3.99	10	9	4.33	7	66.44	370
Lov-obj-ed	165	7.64	24	1260	4.94	10	51	4.90	9	4.24	32
Geode	329	9.36	39	3079	5.62	12	95	5.32	11	4.46	67
Eiffel	1089	7.46	38	8125	6.44	18	169	6.64	17	7.44	402
Laure	236	8.41	16	1984	7.89	12	24	6.42	10	10.83	101
Self	1648	31.91	41	52581	10.05	18	56	9.29	14	30.43	471
Cecil	626	5.88	23	3680	4.96	13	86	4.86	10	8.28	138
Total	14505	7.89	41	114405	4.97	18	1142	4.64	17	13.70	2124

FIG. A.4 – De haut en bas, statistiques de hiérarchies globales, du cœur et de la couronne

d'après [Wang et Smith, 2001]				
Programme	lignes	méthodes	*casts*	sûrs (%)
jlex	7835	398	65	100.0
toba	6417	777	63	22.2
javacup	10592	532	459	89.3
jtar	11904	1446	10	100.0
bloat	18841	1053	205	30.2
self	23122	1304	130	93.8
javadoc	–	2314	310	77.7
sablecc	23111	2811	519	61.7
javac	–	2933	606	50.1

FIG. A.5 – Statistiques sur le *casting*

A.3 Modèles

Un système particulier peut être modélisé en choisissant quelques paramètres, présumés significatifs, et en étudiant l'évolution du système suivant différentes valeurs de ces paramètres. Pour avoir un minimum de généralité, les paramètres choisis doivent être en grande partie statistiques, des moyennes par exemple. Mais choisir la moyenne d'un paramètre suppose que les valeurs effectives ne s'écartent pas trop de cette moyenne. Or le choix d'un tel paramètre est loin d'être évident, en même temps qu'innocent.

Ainsi du nombre de méthodes (ou attributs) par exemple. On peut modéliser une distribution régulière en supposant un nombre moyen de méthodes par classe, ou un nombre moyen de méthodes introduites par classe. Dans le premier cas, les tables de méthodes de l'implémentation standard sont quadratiques, dans le second cubiques !

A.3.1 Modèle d'introduction uniforme des méthodes

Prenons le cas de SMALLTALK sans les méta-classes et supposons que l'introduction des méthodes soit uniforme, à l'exception de la classe `Object` qui en introduit 110 (figure A.1). De la figure **??**, on peut extrapoler que chaque classe a en moyenne 5 ou 6 super-classes, et que la profondeur est au plus de 13. Le nombre moyen de méthodes par classe (M/c) devrait donc se situer aux environ de $110+6\times10.5 = 173.5$, et le nombre maximum de $110 + 13 \times 10.5 = 246.5$: les statistiques donnent respectivement 161 et 260. Le modèle d'*introduction uniforme des méthodes dans les sous-classes de* `Object` a donc une certaine vraisemblance[4].

Mais ce modèle n'est plus vraisemblable si on y inclut `Object` et les méta-classes.

[4] Nous n'avons pas tenu compte ici de la *surcharge* particulière au typage dynamique, dont les statistiques montrent qu'elle est relativement marginale [Ducournau, 1997].

Annexe B

Coloration : Statistiques complètes

B.1 Variantes déterministes

B.2 Variantes stochastiques

Système	nb classes	coloration unidirectionnelle min. nombre		coloration unidirectionnelle minimise taille		coloration bidirection. minimise taille		gain taille	gain trous	taux trous	ferm. trans.
Java-a	34	272	137	148	13	135	0	2.01	∞	0.00	135
Java-b	105	1470	1017	552	99	473	20	3.11	51.	0.04	453
Java-1.18	299	4784	3413	1690	319	1469	98	3.26	35.	0.07	1371
Java-1.22	966	16422	11531	6879	1988	5455	564	3.01	20.	0.12	4891
Java-1.30	1140	21660	15712	7958	2010	7212	1264	3.00	12.	0.21	5948
IDL	11	77	29	51	3	48	0	1.60	∞	0.00	48
Unidraw	25	225	113	124	12	113	1	1.99	113.	0.01	112
Lov-obj-ed	271	6504	4057	3935	1488	3575	1128	1.82	4.	0.46	2447
Geode	989	49450	34087	25348	9985	23206	7843	2.13	4.	0.51	15363
Eiffel	910	35490	26056	14919	5485	12011	2577	2.95	10.	0.27	9434
Laure	57	912	504	479	71	416	8	2.19	63.	0.02	408
Flavors	67	804	476	422	94	345	17	2.33	28.	0.05	328
Self	154	6006	2948	3806	748	3486	428	1.72	7.	0.14	3058
Total (3/2)	5028	144076	100080	66311	22315	57944	13948	2.49	7.	0.32	43996
Java-a	226	1808	868	953	13	940	0	1.92	∞	0.00	940
Java-b	604	8456	5654	2965	163	2822	20	3.00	283.	0.01	2802
Java-1.18	1704	27264	19853	8080	669	7580	169	3.60	117.	0.02	7411
Java-1.22	4339	73763	54822	23893	4952	20192	1251	3.65	44.	0.07	18941
Java-1.30	5438	103322	79550	29094	5322	27053	3281	3.82	24.	0.14	23772
IDL	67	603	283	323	3	320	0	1.88	∞	0.00	320
Unidraw	614	6140	3672	2480	12	2469	1	2.49	3672.	0.00	2468
Lov-obj-ed	436	10464	6757	5789	2082	5308	1601	1.97	4.	0.43	3707
Geode	1318	65900	47458	30104	11662	27699	9257	2.38	5.	0.50	18442
Eiffel	1999	77961	60402	27068	9509	21105	3546	3.69	17.	0.20	17559
Laure	293	4688	2296	2543	151	2400	8	1.95	287.	0.00	2392
Flavors	67	804	476	422	94	345	17	2.33	28.	0.05	328
Self	1802	73882	18243	59731	4092	59411	3772	1.24	5.	0.07	55639
Total (3/2)	18907	455055	300334	193445	38724	177644	22923	2.56	13.	0.15	154721

FIG. B.1 – Statistiques de coloration suivant le critère de minimisation, pour le cœur en haut et le total en bas : chaque coloration a deux colonnes, pour la taille totale et les trous (heuristique 3-2)

Système	nb classes	coloration unidirectionnelle min. nombre		coloration unidirectionnelle minimise taille		coloration bidirection. minimise taille		gain taille	gain trous	taux trous	ferm. trans.
Java-a	34	272	137	148	13	135	0	2.01	∞	0.00	135
Java-b	105	1470	1017	552	99	473	20	3.11	51.	0.04	453
Java-1.18	299	4784	3413	1753	382	1472	101	3.25	34.	0.07	1371
Java-1.22	966	16422	11531	6879	1988	5538	647	2.97	18.	0.13	4891
Java-1.30	1140	21660	15712	8045	2097	6959	1011	3.11	16.	0.17	5948
IDL	11	77	29	51	3	48	0	1.60	∞	0.00	48
Unidraw	25	225	113	124	12	113	1	1.99	113.	0.01	112
Lov-obj-ed	271	6504	4057	4133	1686	3436	989	1.89	4.	0.40	2447
Geode	989	49450	34087	27736	12373	23779	8416	2.08	4.	0.55	15363
Eiffel	910	35490	26056	13958	4524	12217	2783	2.90	9.	0.29	9434
Laure	57	912	504	479	71	416	8	2.19	63.	0.02	408
Flavors	67	804	476	422	94	353	25	2.28	19.	0.08	328
Self	154	6006	2948	3806	748	3483	425	1.72	7.	0.14	3058
Total (2/2)	5028	144076	100080	68086	24090	58422	14426	2.47	7.	0.33	43996
Java-a	226	1808	868	953	13	940	0	1.92	∞	0.00	940
Java-b	604	8456	5654	2965	163	2822	20	3.00	283.	0.01	2802
Java-1.18	1704	27264	19853	8143	732	7579	168	3.60	118.	0.02	7411
Java-1.22	4339	73763	54822	23893	4952	20457	1516	3.61	36.	0.08	18941
Java-1.30	5438	103322	79550	28933	5161	26482	2710	3.90	29.	0.11	23772
IDL	67	603	283	323	3	320	0	1.88	∞	0.00	320
Unidraw	614	6140	3672	2480	12	2469	1	2.49	3672.	0.00	2468
Lov-obj-ed	436	10464	6757	6078	2371	5108	1401	2.05	5.	0.38	3707
Geode	1318	65900	47458	32941	14499	28265	9823	2.33	5.	0.53	18442
Eiffel	1999	77961	60402	24206	6647	21821	4262	3.57	14.	0.24	17559
Laure	293	4688	2296	2543	151	2400	8	1.95	287.	0.00	2392
Flavors	67	804	476	422	94	353	25	2.28	19.	0.08	328
Self	1802	73882	18243	59731	4092	59408	3769	1.24	5.	0.07	55639
Total (2/2)	18907	455055	300334	193611	38890	178424	23703	2.55	13.	0.15	154721

FIG. B.2 – Statistiques de coloration suivant le critère de minimisation, pour le cœur en haut et le total en bas : chaque coloration a deux colonnes, pour la taille totale et les trous (heuristique 2-2)

Système	nb classes	coloration					gain		taux	ferm.
		unidirectionnelle			bidirection.		taille	trous	trous	trans.
		min. nombre		minimise taille						
Java-a	34	272	137	148	13	135	0	2.01	∞	0.00	135
Java-b	105	1470	1017	544	91	466	13	3.15	78.	0.03	453
Java-1.18	299	4784	3413	1696	325	1461	90	3.27	38.	0.07	1371
Java-1.22	966	16422	11531	7613	2722	5962	1071	2.75	11.	0.22	4891
Java-1.30	1140	22800	16852	13661	7713	8988	3040	2.54	6.	0.51	5948
IDL	11	77	29	51	3	48	0	1.60	∞	0.00	48
Unidraw	25	225	113	124	12	113	1	1.99	113.	0.01	112
Lov-obj-ed	271	6504	4057	4231	1784	3453	1006	1.88	4.	0.41	2447
Geode	989	49450	34087	26156	10793	21731	6368	2.28	5.	0.41	15363
Eiffel	910	35490	26056	16935	7501	12786	3352	2.78	8.	0.36	9434
Laure	57	912	504	471	63	420	12	2.17	42.	0.03	408
Flavors	67	804	476	417	89	356	28	2.26	17.	0.09	328
Self	154	6006	2948	3734	676	3428	370	1.75	8.	0.12	3058
TOTAL (2/1)	5028	145216	101220	75781	31785	59347	15351	2.45	7.	0.35	43996
Java-a	226	1808	868	953	13	940	0	1.92	∞	0.00	940
Java-b	604	8456	5654	2966	164	2815	13	3.00	435.	0.00	2802
Java-1.18	1704	27264	19853	8086	675	7531	120	3.62	165.	0.02	7411
Java-1.22	4339	73763	54822	26385	7444	22041	3100	3.35	18.	0.16	18941
Java-1.30	5438	108760	84988	47480	23708	33393	9621	3.26	9.	0.40	23772
IDL	67	603	283	323	3	320	0	1.88	∞	0.00	320
Unidraw	614	6140	3672	2480	12	2469	1	2.49	3672.	0.00	2468
Lov-obj-ed	436	10464	6757	6343	2636	5008	1301	2.09	5.	0.35	3707
Geode	1318	65900	47458	32530	14088	26751	8309	2.46	6.	0.45	18442
Eiffel	1999	77961	60402	32211	14652	23713	6154	3.29	10.	0.35	17559
Laure	293	4688	2296	2487	95	2407	15	1.95	153.	0.01	2392
Flavors	67	804	476	417	89	356	28	2.26	17.	0.09	328
Self	1802	73882	18243	57146	1507	56806	1167	1.30	16.	0.02	55639
TOTAL (2/1)	18907	460493	305772	219807	65086	184550	29829	2.50	10.	0.19	154721

FIG. B.3 – Statistiques de coloration suivant le critère de minimisation, pour le cœur en haut et le total en bas : chaque coloration a deux colonnes, pour la taille totale et les trous (heuristique 2-1)

Système	nb classes	coloration					gain		taux trous	ferm. trans.
		unidirectionnelle			bidirection.		taille	trous		
		min. nombre		minimise taille						
Java-a	34	272	137	148	13	135	0	2.01	∞	0.00	135
Java-b	105	1470	1017	544	91	466	13	3.15	78.	0.03	453
Java-1.18	299	4784	3413	1693	322	1467	96	3.26	36.	0.07	1371
Java-1.22	966	16422	11531	7767	2876	6346	1455	2.59	8.	0.30	4891
Java-1.30	1140	22800	16852	13662	7714	9002	3054	2.53	6.	0.51	5948
IDL	11	77	29	51	3	48	0	1.60	∞	0.00	48
Unidraw	25	225	113	124	12	113	1	1.99	113.	0.01	112
Lov-obj-ed	271	6504	4057	4156	1709	3642	1195	1.79	3.	0.49	2447
Geode	989	49450	34087	26558	11195	22437	7074	2.20	5.	0.46	15363
Eiffel	910	35490	26056	17412	7978	12861	3427	2.76	8.	0.36	9434
Laure	57	912	504	469	61	416	8	2.19	63.	0.02	408
Flavors	67	804	476	417	89	356	28	2.26	17.	0.09	328
Self	154	6006	2948	3679	621	3400	342	1.77	9.	0.11	3058
TOTAL (1/1)	5028	145216	101220	76680	32684	60689	16693	2.39	6.	0.38	43996
Java-a	226	1808	868	953	13	940	0	1.92	∞	0.00	940
Java-b	604	8456	5654	2966	164	2815	13	3.00	435.	0.00	2802
Java-1.18	1704	27264	19853	8084	673	7539	128	3.62	155.	0.02	7411
Java-1.22	4339	73763	54822	26841	7900	23765	4824	3.10	11.	0.25	18941
Java-1.30	5438	108760	84988	47489	23717	33208	9436	3.28	9.	0.40	23772
IDL	67	603	283	323	3	320	0	1.88	∞	0.00	320
Unidraw	614	6140	3672	2480	12	2469	1	2.49	3672.	0.00	2468
Lov-obj-ed	436	10464	6757	6155	2448	5378	1671	1.95	4.	0.45	3707
Geode	1318	65900	47458	33102	14660	27674	9232	2.38	5.	0.50	18442
Eiffel	1999	77961	60402	33452	15893	23841	6282	3.27	10.	0.36	17559
Laure	293	4688	2296	2475	83	2400	8	1.95	287.	0.00	2392
Flavors	67	804	476	417	89	356	28	2.26	17.	0.09	328
Self	1802	73882	18243	57103	1464	56717	1078	1.30	17.	0.02	55639
TOTAL (1/1)	18907	460493	305772	221840	67119	187422	32701	2.46	9.	0.21	154721

FIG. B.4 – Statistiques de coloration suivant le critère de minimisation, pour le cœur en haut et le total en bas : chaque coloration a deux colonnes, pour la taille totale et les trous (heuristique 1-1)

Système	nb classes	fermeture transit.	coloration unidirectionnelle						χ	coloration bidirectionnelle						χ
			nb couleurs			nb trous				nb couleurs			nb trous			
			max	moy	total	max	moy	total		max	moy	total	max	moy	total	
Java-a	34	135	8	4.35	148	1	0.38	13	1.10	8	3.97	135	0	0.00	0	1.00
Java-b	105	453	14	5.26	552	6	0.94	99	1.22	14	4.50	473	2	0.19	20	1.04
Java-1.18	299	1371	16	5.65	1690	11	1.07	319	1.23	16	4.91	1469	5	0.33	98	1.07
Java-1.22	966	4891	17	7.12	6879	13	2.06	1988	1.41	17	5.65	5455	7	0.58	564	1.12
Java-1.30	1140	5948	19	6.98	7958	14	1.76	2010	1.34	**21**	6.33	7212	10	1.11	1264	1.21
IDL	11	48	7	4.64	51	1	0.27	3	1.06	7	4.36	48	0	0.00	0	1.00
Unidraw	25	112	9	4.96	124	2	0.48	12	1.11	9	4.52	113	1	0.04	1	1.01
Lov-obj-ed	271	2447	24	14.52	3935	21	5.49	1488	1.61	**25**	13.19	3575	14	4.16	1128	1.46
Geode	989	15363	50	25.63	25348	40	10.10	9985	1.65	50	23.46	23206	30	7.93	7843	1.51
Eiffel	910	9434	39	16.39	14919	29	6.03	5485	1.58	39	13.20	12011	18	2.83	2577	1.27
Laure	57	408	16	8.40	479	5	1.25	71	1.17	16	7.30	416	2	0.14	8	1.02
Flavors	67	328	12	6.30	422	7	1.40	94	1.29	12	5.15	345	3	0.25	17	1.05
Self	154	3058	39	24.71	3806	29	4.86	748	1.24	39	22.64	3486	19	2.78	428	1.14
Total (3/2)	5028	43996	50	13.19	66311	40	4.44	22315	1.51	50	11.52	57944	30	2.77	13948	1.32
Java-a	226	940	8	4.22	953	1	0.06	13	1.01	8	4.16	940	0	0.00	0	1.00
Java-b	604	2802	14	4.91	2965	6	0.27	163	1.06	14	4.67	2822	2	0.03	20	1.01
Java-1.18	1704	7411	16	4.74	8080	11	0.39	669	1.09	16	4.45	7580	5	0.10	169	1.02
Java-1.22	4339	18941	17	5.51	23893	13	1.14	4952	1.26	17	4.65	20192	7	0.29	1251	1.07
Java-1.30	5438	23772	19	5.35	29094	14	0.98	5322	1.22	**21**	4.97	27053	10	0.60	3281	1.14
IDL	67	320	9	4.82	323	1	0.04	3	1.01	9	4.78	320	0	0.00	0	1.00
Unidraw	614	2468	10	4.04	2480	2	0.02	12	1.00	10	4.02	2469	1	0.00	1	1.00
Lov-obj-ed	436	3707	24	13.28	5789	21	4.78	2082	1.56	**25**	12.17	5308	14	3.67	1601	1.43
Geode	1318	18442	50	22.84	30104	40	8.85	11662	1.63	50	21.02	27699	30	7.02	9257	1.50
Eiffel	1999	17559	39	13.54	27068	29	4.76	9509	1.54	39	10.56	21105	18	1.77	3546	1.20
Laure	293	2392	16	8.68	2543	5	0.52	151	1.06	16	8.19	2400	2	0.03	8	1.00
Flavors	67	328	12	6.30	422	7	1.40	94	1.29	12	5.15	345	3	0.25	17	1.05
Self	1802	55639	41	33.15	59731	29	2.27	4092	1.07	41	32.97	59411	19	2.09	3772	1.07
Total (3/2)	18907	154721	50	10.23	193445	40	2.05	38724	1.25	50	9.40	177644	30	1.21	22923	1.15

FIG. B.5 – Statistiques de coloration pour l'heuristique (3-2)

Système	nb classes	fermeture transit.	coloration unidirectionnelle						χ	coloration bidirectionnelle						χ
			nb couleurs			nb trous				nb couleurs			nb trous			
			max	moy	total	max	moy	total		max	moy	total	max	moy	total	
Java-a	34	135	8	4.35	148	1	0.38	13	1.10	8	3.97	135	0	0.00	0	1.00
Java-b	105	453	14	5.26	552	6	0.94	99	1.22	14	4.50	473	2	0.19	20	1.04
Java-1.18	299	1371	16	5.86	1753	11	1.28	382	1.28	16	4.92	1472	5	0.34	101	1.07
Java-1.22	966	4891	17	7.12	6879	13	2.06	1988	1.41	17	5.73	5538	8	0.67	647	1.13
Java-1.30	1140	5948	19	7.06	8045	14	1.84	2097	1.35	19	6.10	6959	10	0.89	1011	1.17
IDL	11	48	7	4.64	51	1	0.27	3	1.06	7	4.36	48	0	0.00	0	1.00
Unidraw	25	112	9	4.96	124	2	0.48	12	1.11	9	4.52	113	1	0.04	1	1.01
Lov-obj-ed	271	2447	24	15.25	4133	22	6.22	1686	1.69	**26**	12.68	3436	15	3.65	989	1.40
Geode	989	15363	50	28.04	27736	46	12.51	12373	1.81	50	24.04	23779	26	8.51	8416	1.55
Eiffel	910	9434	39	15.34	13958	33	4.97	4524	1.48	39	13.43	12217	20	3.06	2783	1.29
Laure	57	408	16	8.40	479	5	1.25	71	1.17	16	7.30	416	2	0.14	8	1.02
Flavors	67	328	12	6.30	422	7	1.40	94	1.29	12	5.27	353	4	0.37	25	1.08
Self	154	3058	39	24.71	3806	29	4.86	748	1.24	39	22.62	3483	18	2.76	425	1.14
Total (2/2)	5028	43996	50	13.54	68086	46	4.79	24090	1.55	50	11.62	58422	26	2.87	14426	1.33
Java-a	226	940	8	4.22	953	1	0.06	13	1.01	8	4.16	940	0	0.00	0	1.00
Java-b	604	2802	14	4.91	2965	6	0.27	163	1.06	14	4.67	2822	2	0.03	20	1.01
Java-1.18	1704	7411	16	4.78	8143	11	0.43	732	1.10	16	4.45	7579	5	0.10	168	1.02
Java-1.22	4339	18941	17	5.51	23893	13	1.14	4952	1.26	17	4.71	20457	8	0.35	1516	1.08
Java-1.30	5438	23772	19	5.32	28933	14	0.95	5161	1.22	19	4.87	26482	10	0.50	2710	1.11
IDL	67	320	9	4.82	323	1	0.04	3	1.01	9	4.78	320	0	0.00	0	1.00
Unidraw	614	2468	10	4.04	2480	2	0.02	12	1.00	10	4.02	2469	1	0.00	1	1.00
Lov-obj-ed	436	3707	24	13.94	6078	22	5.44	2371	1.64	**26**	11.72	5108	15	3.21	1401	1.38
Geode	1318	18442	50	24.99	32941	46	11.00	14499	1.79	50	21.45	28265	26	7.45	9823	1.53
Eiffel	1999	17559	39	12.11	24206	33	3.33	6647	1.38	39	10.92	21821	20	2.13	4262	1.24
Laure	293	2392	16	8.68	2543	5	0.52	151	1.06	16	8.19	2400	2	0.03	8	1.00
Flavors	67	328	12	6.30	422	7	1.40	94	1.29	12	5.27	353	4	0.37	25	1.08
Self	1802	55639	41	33.15	59731	29	2.27	4092	1.07	41	32.97	59408	18	2.09	3769	1.07
Total (2/2)	18907	154721	50	10.24	193611	46	2.06	38890	1.25	50	9.44	178424	26	1.25	23703	1.15

FIG. B.6 – Statistiques de coloration pour l'heuristique (2-2)

Système	nb classes	fermeture transit.	coloration unidirectionnelle						χ	coloration bidirectionnelle						χ
			nb couleurs			nb trous				nb couleurs			nb trous			
			max	moy	total	max	moy	total		max	moy	total	max	moy	total	
Java-a	34	135	8	4.35	148	1	0.38	13	1.10	8	3.97	135	0	0.00	0	1.00
Java-b	105	453	14	5.18	544	6	0.87	91	1.20	14	4.44	466	2	0.12	13	1.03
Java-1.18	299	1371	16	5.67	1696	11	1.09	325	1.24	16	4.89	1461	5	0.30	90	1.07
Java-1.22	966	4891	17	7.88	7613	13	2.82	2722	1.56	17	6.17	5962	9	1.11	1071	1.22
Java-1.30	1140	5948	**20**	11.98	13661	17	6.77	7713	2.30	**22**	7.88	8988	14	2.67	3040	1.51
IDL	11	48	7	4.64	51	1	0.27	3	1.06	7	4.36	48	0	0.00	0	1.00
Unidraw	25	112	9	4.96	124	2	0.48	12	1.11	9	4.52	113	1	0.04	1	1.01
Lov-obj-ed	271	2447	24	15.61	4231	21	6.58	1784	1.73	**25**	12.74	3453	14	3.71	1006	1.41
Geode	989	15363	50	26.45	26156	43	10.91	10793	1.70	50	21.97	21731	31	6.44	6368	1.41
Eiffel	910	9434	39	18.61	16935	31	8.24	7501	1.80	39	14.05	12786	21	3.68	3352	1.36
Laure	57	408	16	8.26	471	6	1.11	63	1.15	16	7.37	420	2	0.21	12	1.03
Flavors	67	328	12	6.22	417	7	1.33	89	1.27	12	5.31	356	2	0.42	28	1.09
Self	154	3058	39	24.25	3734	32	4.39	676	1.22	39	22.26	3428	15	2.40	370	1.12
TOTAL (2/1)	5028	43996	50	15.07	75781	43	6.32	31785	1.72	50	11.80	59347	31	3.05	15351	1.35
Java-a	226	940	8	4.22	953	1	0.06	13	1.01	8	4.16	940	0	0.00	0	1.00
Java-b	604	2802	14	4.91	2966	6	0.27	164	1.06	14	4.66	2815	2	0.02	13	1.00
Java-1.18	1704	7411	16	4.75	8086	11	0.40	675	1.09	16	4.42	7531	5	0.07	120	1.02
Java-1.22	4339	18941	17	6.08	26385	13	1.72	7444	1.39	17	5.08	22041	9	0.71	3100	1.16
Java-1.30	5438	23772	**20**	8.73	47480	17	4.36	23708	2.00	**22**	6.14	33393	14	1.77	9621	1.40
IDL	67	320	9	4.82	323	1	0.04	3	1.01	9	4.78	320	0	0.00	0	1.00
Unidraw	614	2468	10	4.04	2480	2	0.02	12	1.00	10	4.02	2469	1	0.00	1	1.00
Lov-obj-ed	436	3707	24	14.55	6343	21	6.05	2636	1.71	**25**	11.49	5008	14	2.98	1301	1.35
Geode	1318	18442	50	24.68	32530	43	10.69	14088	1.76	50	20.30	26751	31	6.30	8309	1.45
Eiffel	1999	17559	39	16.11	32211	31	7.33	14652	1.83	39	11.86	23713	21	3.08	6154	1.35
Laure	293	2392	16	8.49	2487	6	0.32	95	1.04	16	8.22	2407	2	0.05	15	1.01
Flavors	67	328	12	6.22	417	7	1.33	89	1.27	12	5.31	356	2	0.42	28	1.09
Self	1802	55639	41	31.71	57146	32	0.84	1507	1.03	41	31.52	56806	15	0.65	1167	1.02
TOTAL (2/1)	18907	154721	50	11.63	219807	43	3.44	65086	1.42	50	9.76	184550	31	1.58	29829	1.19

FIG. B.7 – Statistiques de coloration pour l'heuristique (2-1)

Système	nb classes	fermeture transit.	coloration unidirectionnelle						χ	coloration bidirectionnelle						χ
			nb couleurs			nb trous				nb couleurs			nb trous			
			max	moy	total	max	moy	total		max	moy	total	max	moy	total	
Java-a	34	135	8	4.35	148	1	0.38	13	1.10	8	3.97	135	0	0.00	0	1.00
Java-b	105	453	14	5.18	544	6	0.87	91	1.20	14	4.44	466	2	0.12	13	1.03
Java-1.18	299	1371	16	5.66	1693	11	1.08	322	1.23	16	4.91	1467	5	0.32	96	1.07
Java-1.22	966	4891	17	8.04	7767	14	2.98	2876	1.59	**20**	6.57	6346	**15**	1.51	1455	1.30
Java-1.30	1140	5948	**20**	11.98	13662	17	6.77	7714	2.30	**22**	7.90	9002	14	2.68	3054	1.51
IDL	11	48	7	4.64	51	1	0.27	3	1.06	7	4.36	48	0	0.00	0	1.00
Unidraw	25	112	9	4.96	124	2	0.48	12	1.11	9	4.52	113	1	0.04	1	1.01
Lov-obj-ed	271	2447	24	15.34	4156	20	6.31	1709	1.70	24	13.44	3642	17	4.41	1195	1.49
Geode	989	15363	50	26.85	26558	44	11.32	11195	1.73	50	22.69	22437	32	7.15	7074	1.46
Eiffel	910	9434	39	19.13	17412	31	8.77	7978	1.85	39	14.13	12861	21	3.77	3427	1.36
Laure	57	408	16	8.23	469	6	1.07	61	1.15	16	7.30	416	2	0.14	8	1.02
Flavors	67	328	12	6.22	417	7	1.33	89	1.27	12	5.31	356	2	0.42	28	1.09
Self	154	3058	39	23.89	3679	31	4.03	621	1.20	39	22.08	3400	15	2.22	342	1.11
TOTAL (1/1)	5028	43996	50	15.25	76680	44	6.50	32684	1.74	50	12.07	60689	32	3.32	16693	1.38
Java-a	226	940	8	4.22	953	1	0.06	13	1.01	8	4.16	940	0	0.00	0	1.00
Java-b	604	2802	14	4.91	2966	6	0.27	164	1.06	14	4.66	2815	2	0.02	13	1.00
Java-1.18	1704	7411	16	4.74	8084	11	0.39	673	1.09	16	4.42	7539	5	0.08	128	1.02
Java-1.22	4339	18941	17	6.19	26841	14	1.82	7900	1.42	**20**	5.48	23765	**15**	1.11	4824	1.25
Java-1.30	5438	23772	**20**	8.73	47489	17	4.36	23717	2.00	**22**	6.11	33208	14	1.74	9436	1.40
IDL	67	320	9	4.82	323	1	0.04	3	1.01	9	4.78	320	0	0.00	0	1.00
Unidraw	614	2468	10	4.04	2480	2	0.02	12	1.00	10	4.02	2469	1	0.00	1	1.00
Lov-obj-ed	436	3707	24	14.12	6155	20	5.61	2448	1.66	24	12.33	5378	17	3.83	1671	1.45
Geode	1318	18442	50	25.12	33102	44	11.12	14660	1.79	50	21.00	27674	32	7.00	9232	1.50
Eiffel	1999	17559	39	16.73	33452	31	7.95	15893	1.91	39	11.93	23841	21	3.14	6282	1.36
Laure	293	2392	16	8.45	2475	6	0.28	83	1.03	16	8.19	2400	2	0.03	8	1.00
Flavors	67	328	12	6.22	417	7	1.33	89	1.27	12	5.31	356	2	0.42	28	1.09
Self	1802	55639	41	31.69	57103	31	0.81	1464	1.03	41	31.47	56717	15	0.60	1078	1.02
TOTAL (1/1)	18907	154721	50	11.73	221840	44	3.55	67119	1.43	50	9.91	187422	32	1.73	32701	1.21

FIG. B.8 – Statistiques de coloration pour l'heuristique (1-1)

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	cœur			total			cœur			total		
Java-b	91	91.	91	164	164.	164	12	14.	16	12	14.	16
Java-1.18	313	345.	387	671	706.	759	81	90.	102	109	143.	178
Java-1.22	2876	2876.	2876	7900	7900.	7900	1063	1364.	1619	3590	4394.	5182
Java-1.30	7712	7713.	7714	23715	23716.	23717	2357	2631.	2837	6710	7364.	7904
Unidraw	12	12.	12	12	12.	12	1	2.	2	1	2.	2
Lov-obj-ed	1751	1752.	1755	2511	2512.	2515	1096	1188.	1385	1504	1626.	1830
Geode	11034	11048.	11112	14476	14490.	14554	6106	7172.	8521	7678	9222.	10875
Eiffel	7623	7950.	8252	14992	15859.	16622	3412	3863.	4322	5905	7195.	8104
Laure	61	61.	61	83	83.	83	8	8.	8	8	8.	8
Self	621	621.	621	1464	1464.	1464	334	338.	342	1070	1077.	1083
Cecil	665	665.	665	1451	1451.	1451	174	198.	225	279	334.	399
Total (1/2)	32775	33151.	33562	67455	68374.	69257	14644	16868.	19379	26866	31377.	35581
Java-b	83	83.	83	147	147.	147	20	20.	20	20	20.	20
Java-1.18	315	362.	407	635	713.	778	84	98.	111	141	172.	256
Java-1.22	2267	2271.	2284	4434	4439.	4452	720	831.	936	1225	1469.	1707
Java-1.30	1927	1954.	2002	5289	5983.	6550	747	1088.	1420	1932	2895.	3606
Unidraw	12	12.	12	12	12.	12	1	1.	2	1	1.	2
Lov-obj-ed	1470	1655.	1893	2069	2314.	2641	899	1078.	1200	1147	1570.	1805
Geode	10859	11482.	12289	12500	13225.	14116	7142	7587.	8291	8121	8691.	9651
Eiffel	4286	4451.	4625	6262	6562.	6788	2495	3274.	4067	3645	5175.	6910
Laure	75	75.	75	208	208.	208	10	10.	10	17	17.	17
Self	656	656.	656	4000	4000.	4000	414	419.	422	3492	3497.	3500
Cecil	438	438.	438	624	624.	624	129	183.	229	147	238.	324
Total (2/2)	22404	23455.	24780	36196	38243.	40332	12661	14589.	16708	19888	23746.	27798
Java-a	13	13.	14	13	13.	14	0	0.	0	0	0.	0
Java-b	76	93.	126	105	143.	201	9	15.	23	9	16.	40
Java-1.18	267	347.	632	449	657.	1732	69	100.	166	84	140.	305
Java-1.22	1600	2415.	5186	3860	6640.	14706	554	923.	2364	1145	2426.	5458
Java-1.30	2139	2975.	6027	5092	8523.	17943	793	1199.	2803	1707	3153.	6461
Unidraw	12	13.	14	12	13.	14	1	2.	2	1	2.	2
Lov-obj-ed	1354	1637.	2193	1845	2354.	3291	734	991.	1345	970	1388.	2070
Geode	10779	13079.	15334	12810	15808.	18709	6055	8984.	12457	7011	10641.	14662
Eiffel	4974	5705.	6644	7262	9417.	11715	2384	2981.	3780	3170	4534.	6711
Laure	60	71.	102	87	155.	341	7	12.	28	7	15.	63
Self	675	759.	891	3513	4091.	4235	350	468.	616	610	3442.	5341
Cecil	406	482.	676	553	829.	1882	115	159.	366	133	217.	728
Total (0/2)	22358	27590.	37842	35604	48644.	74786	11071	15832.	23950	14847	25973.	41841

FIG. B.9 – Statistiques de nombres de trous (minimum, moyenne et maximum) suivant une variation aléatoire dans le cadre du schéma d'algorithme, avec l'heuristique de choix de la couleur 2 : de haut en bas, avec l'heuristique du nombre de conflits future pour le max, avec l'heuristique de la meilleure couleur pour le max ou sans heuristique. En bas, sans heuristiques.

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	cœur			total			cœur			total		
Java-b	83	97.	121	128	169.	213	11	17.	27	11	22.	46
Java-1.18	313	362.	483	669	805.	1215	87	114.	157	116	185.	268
Java-1.22	2741	3741.	6214	7500	10694.	18603	1086	1672.	2839	3334	4910.	7854
Java-1.30	3675	6187.	7834	10731	18825.	24191	1586	2442.	3544	4881	7517.	10020
Unidraw	12	12.	12	12	12.	12	1	2.	3	1	2.	3
Lov-obj-ed	1688	1730.	1751	2356	2459.	2522	940	1192.	1402	1272	1625.	1853
Geode	11051	11226.	11497	14363	14569.	14894	6110	7383.	8319	7797	9309.	10725
Eiffel	6976	7697.	8215	13041	15076.	16432	3057	3871.	4341	5593	7123.	8259
Laure	61	62.	64	79	92.	108	8	9.	12	8	11.	18
Self	621	704.	771	1274	2753.	3640	334	390.	458	875	1894.	3296
Cecil	665	665.	665	1451	1451.	1451	171	195.	230	272	325.	395
Total (1/1)	27902	32498.	37643	51620	66921.	83297	13391	17287.	21332	24160	32925.	42737
Java-b	91	91.	91	164	164.	164	12	18.	27	12	26.	43
Java-1.18	325	325.	325	675	675.	675	82	104.	127	119	181.	219
Java-1.22	2722	2723.	2734	7444	7453.	7654	1052	1248.	1405	2880	3789.	4820
Java-1.30	7713	7714.	7718	23708	23709.	23717	2339	2764.	3290	7463	8548.	9927
Unidraw	12	12.	12	12	12.	12	1	1.	2	1	1.	2
Lov-obj-ed	1784	1784.	1784	2636	2636.	2636	938	1037.	1338	1229	1406.	1973
Geode	10793	10795.	10797	14088	14090.	14092	5997	7340.	8673	7689	9323.	11117
Eiffel	7501	7503.	7506	14652	14655.	14659	3409	3831.	4456	5989	7017.	7983
Laure	63	63.	63	95	95.	95	8	10.	12	8	11.	15
Self	676	676.	676	1507	1507.	1507	364	367.	371	1094	1127.	1168
Cecil	636	638.	640	1306	1308.	1310	140	194.	258	211	314.	424
Total (2/1)	32332	32339.	32362	66303	66320.	66537	14342	16914.	19959	26695	31744.	37691
Java-a	13	13.	14	13	13.	14	0	0.	0	0	0.	0
Java-b	76	94.	123	107	145.	200	9	18.	39	9	19.	52
Java-1.18	275	356.	528	404	669.	1386	67	107.	210	85	147.	369
Java-1.22	1566	2416.	4679	3657	6662.	13077	563	975.	2385	1072	2490.	5799
Java-1.30	2072	3137.	6441	5291	8994.	19195	746	1357.	2781	1671	3431.	7570
Unidraw	12	12.	14	12	12.	14	1	2.	3	1	2.	3
Lov-obj-ed	1318	1598.	2138	1887	2288.	3196	711	1005.	1410	896	1408.	2142
Geode	10426	12941.	14741	12512	15617.	18575	5868	8991.	11818	6931	10645.	14452
Eiffel	4841	5735.	6884	6822	9522.	12757	2329	2998.	3864	3170	4533.	6512
Laure	59	72.	101	73	153.	314	7	14.	34	7	19.	65
Self	668	758.	929	2872	4071.	4273	380	476.	597	1392	3673.	5364
Cecil	405	479.	676	549	807.	1688	115	174.	303	131	236.	532
Total (0/1)	21734	27614.	37271	34202	48958.	74692	10796	16115.	23444	15365	26604.	42860

FIG. B.10 – Statistiques de nombres de trous (minimum, moyenne et maximum) suivant une variation aléatoire dans le cadre du schéma d'algorithme, avec l'heuristique de choix de la couleur 1 : de haut en bas, avec l'heuristique du nombre de conflits future pour le max, avec l'heuristique de la meilleure couleur pour le max ou sans heuristique. En bas, sans heuristiques.

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	cœur			total			cœur			total		
Java-b	83	97.	121	131	171.	239	11	18.	40	11	25.	59
Java-1.18	313	378.	483	662	832.	1232	91	119.	220	116	198.	385
Java-1.22	2752	4078.	6262	7628	11759.	18665	1022	1755.	2986	3324	5051.	7864
Java-1.30	3685	6273.	8452	10757	19177.	26149	1592	2613.	3811	4859	7683.	10400
Unidraw	12	12.	12	12	12.	12	1	2.	3	1	2.	3
Lov-obj-ed	1688	1965.	2267	2356	2773.	3219	1008	1299.	1583	1237	1789.	2095
Geode	11040	11397.	12082	14309	14777.	15642	5943	7710.	8948	7614	9714.	11215
Eiffel	6994	7986.	8616	13049	15830.	17296	3282	4103.	4871	5809	7590.	9156
Laure	61	62.	64	79	92.	108	8	9.	12	8	11.	18
Self	621	700.	771	1283	2660.	3640	334	398.	463	862	2025.	3302
Cecil	665	665.	665	1451	1451.	1451	144	230.	308	214	400.	556
Total (1/0)	27930	33630.	39811	51733	69550.	87669	13436	18256.	23245	24055	34487.	45053
Java-b	83	83.	83	147	147.	147	20	22.	25	20	22.	25
Java-1.18	317	359.	417	641	715.	841	84	113.	140	121	185.	262
Java-1.22	2167	2269.	2418	4054	4554.	4804	705	1003.	1341	1414	2085.	2776
Java-1.30	1928	2045.	2280	5117	6202.	7323	836	1094.	1517	1773	2529.	3722
Unidraw	12	12.	12	12	12.	12	1	2.	3	1	2.	3
Lov-obj-ed	1414	1621.	1811	2001	2336.	2703	899	1181.	1358	1150	1654.	1977
Geode	10369	11283.	12222	12063	13064.	14363	5076	9386.	11879	5945	10898.	14085
Eiffel	4426	4784.	5235	6375	7007.	7762	2670	3408.	4374	3962	5412.	7614
Laure	75	75.	75	208	208.	208	10	15.	25	17	23.	41
Self	656	656.	656	4000	4000.	4000	404	422.	506	2849	3250.	5001
Cecil	438	449.	459	624	649.	673	120	203.	292	146	264.	709
Total (2/0)	21901	23652.	25684	35258	38909.	42852	10825	16849.	21460	17398	26324.	36215
Java-a	13	13.	14	13	13.	14	0	0.	0	0	0.	0
Java-b	78	95.	141	105	145.	240	10	18.	48	10	19.	54
Java-1.18	277	356.	671	394	681.	2007	73	108.	171	88	150.	289
Java-1.22	1632	2433.	5474	3878	6701.	15393	524	956.	2315	1191	2501.	5509
Java-1.30	2263	3079.	6893	5362	8878.	20391	832	1311.	2879	1718	3413.	7333
Unidraw	12	13.	14	12	13.	14	1	2.	3	1	2.	3
Lov-obj-ed	1358	1662.	2148	1899	2397.	3199	774	1029.	1345	1065	1444.	1958
Geode	10712	13119.	14960	13068	15834.	18847	5818	8938.	12169	6898	10622.	14129
Eiffel	4698	5794.	6785	6812	9644.	13122	2352	3083.	3965	3250	4731.	6875
Laure	60	73.	102	82	154.	299	7	15.	33	7	19.	75
Self	677	761.	962	1765	4068.	4306	362	470.	624	1199	3466.	5396
Cecil	392	479.	713	530	809.	2099	113	179.	343	136	246.	663
Total (0/0)	22175	27881.	38880	33923	49341.	79934	10866	16107.	23895	15563	26612.	42284

FIG. B.11 – Statistiques de nombres de trous (minimum, moyenne et maximum) suivant une variation aléatoire dans le cadre du schéma d'algorithme, avec l'heuristique de choix de la couleur 0 : de haut en bas, avec l'heuristique du nombre de conflits future pour le max, avec l'heuristique de la meilleure couleur pour le max ou sans heuristique. En bas, sans heuristiques.

Système	coloration unidirectionnelle						coloration bidirectionnelle					
	cœur			total			cœur			total		
Java-1.30	19	19.0	19	19	19.0	19	19	19.0	20	19	19.0	20
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.4	27	24	24.4	27
Total (3/3)	278	278.0	278	286	286.0	286	278	278.5	282	286	286.5	290
Java-1.30	19	19.0	19	19	19.0	19	19	19.2	21	19	19.2	21
Lov-obj-ed	24	24.0	24	24	24.0	24	24	25.3	27	24	25.3	27
Total (3/2)	278	278.0	278	286	286.0	286	278	279.5	283	286	287.5	291
Java-1.22	17	17.0	17	17	17.0	17	17	17.0	18	17	17.0	18
Java-1.30	19	19.0	19	19	19.0	19	19	19.3	21	19	19.3	21
Lov-obj-ed	24	24.7	26	24	24.7	26	24	25.0	28	24	25.0	28
Total (2/2)	278	278.7	280	286	286.7	288	278	279.3	285	286	287.3	293
Java-1.22	17	17.0	17	17	17.0	17	17	18.7	21	17	18.7	21
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.2	26	24	24.2	26
Total (1/2)	279	279.0	279	287	287.0	287	281	282.8	287	289	290.8	295
Java-1.22	17	17.0	18	17	17.0	18	17	17.1	20	17	17.1	20
Java-1.30	19	19.0	19	19	19.0	19	19	19.1	21	19	19.1	21
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.1	25	24	24.1	25
Cecil	20	20.0	20	23	23.0	23	20	20.0	21	23	23.0	23
Total (0/2)	278	278.0	279	286	286.0	287	278	278.3	285	286	286.3	292
Java-1.22	17	17.0	17	17	17.0	17	17	17.5	21	17	17.5	21
Java-1.30	19	19.2	20	19	19.2	20	19	19.6	22	19	19.6	22
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.1	26	24	24.1	26
Total (1/1)	278	278.2	279	286	286.2	287	278	279.2	287	286	287.2	295
Java-1.22	17	17.0	17	17	17.0	17	17	18.5	20	17	18.5	20
Java-1.30	20	20.0	20	20	20.0	20	21	21.8	22	21	21.8	22
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.6	25	24	24.6	25
Eiffel	39	39.0	39	39	39.0	39	39	39.0	40	39	39.0	40
Total (2/1)	279	279.0	279	287	287.0	287	280	282.8	286	288	290.8	294
Java-1.22	17	17.0	17	17	17.0	17	17	17.0	19	17	17.0	19
Java-1.30	19	19.0	19	19	19.0	19	19	19.1	21	19	19.1	21
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.0	25	24	24.0	25
Eiffel	39	39.0	39	39	39.0	39	39	39.0	41	39	39.0	41
Total (0/1)	278	278.0	278	286	286.0	286	278	278.1	285	286	286.1	293
Java-1.22	17	17.1	18	17	17.1	18	17	17.6	21	17	17.6	21
Java-1.30	19	19.2	21	19	19.2	21	19	19.6	24	19	19.6	24
Lov-obj-ed	24	24.1	25	24	24.1	25	24	24.1	26	24	24.1	26
Geode	50	50.0	50	50	50.0	50	50	50.0	51	50	50.0	51
Eiffel	39	39.0	39	39	39.0	39	39	39.0	41	39	39.0	41
Total (1/0)	278	278.4	282	286	286.4	290	278	279.4	292	286	287.4	300
Java-1.22	17	17.0	17	17	17.0	17	17	17.0	18	17	17.0	18
Java-1.30	19	19.0	19	19	19.0	19	19	19.2	21	19	19.2	21
Lov-obj-ed	24	24.3	26	24	24.3	26	24	24.8	27	24	24.8	27
Geode	50	50.0	50	50	50.0	50	50	50.1	52	50	50.1	52
Cecil	20	20.0	20	23	23.0	23	20	20.0	21	23	23.0	23
Total (2/0)	278	278.3	280	286	286.3	288	278	279.1	287	286	287.1	294
Java-1.18	16	16.0	16	16	16.0	16	16	16.0	17	16	16.0	17
Java-1.22	17	17.0	17	17	17.0	17	17	17.1	20	17	17.1	20
Java-1.30	19	19.0	20	19	19.0	20	19	19.1	23	19	19.1	23
Lov-obj-ed	24	24.0	24	24	24.0	24	24	24.1	28	24	24.1	28
Eiffel	39	39.0	39	39	39.0	39	39	39.0	40	39	39.0	40
Cecil	20	20.0	20	23	23.0	23	20	20.0	21	23	23.0	23
Total (0/0)	278	278.0	279	286	286.0	287	278	278.4	292	286	286.4	299

FIG. B.12 – Statistiques sur le nombre de couleurs (minimum, moyenne et maximum) suivant une variation aléatoire dans le cadre du schéma d'algorithme, dans toutes les variantes d'heuristiques décrites. Seules les hiérarchies qui présentent une variation du nombre de couleurs sont décrites.

Annexe C

Processeurs et pseudo-code de la coloration

Envoi de message

```
load [object + #tableOffset], table
load [table + #selectorColor], method
call method
```

Accès à un attribut

Cas d'un accès non encapsulé, ou d'un accès encapsulé en compilation non efficace :

```
load [object + #tableOffset], table
load [table + #classColor], offset
add object, offset, offset
load [offset + #attributeLocalOffset], attribute
```

Cas d'un attribut encapsulé, en compilation efficace :

```
load [self + #attributeGlobalOffset], attribute
```

Vérification de type

En coloration de classes pure, ou avec une cible statiquement dans le cœur, et test statique :

```
load [object + #tableOffset], table
load [table + #colorTableOffset], table
load [table + #targetColor], classId
comp classId, #targetId
bne #fail
// vérification de type réussie
```

Par double numérotation, pour une cible de la couronne, et test statique :

```
load [object + #n1Offset], classid
comp classid, #targetN1
blt #fail
comp classid, #targetN2
bgt #fail
// vérification de type réussie
```

Le cas d'un test statique sur une cible dont on n'est pas sûr se traitera par inclusion conditionnelle de ces deux fragments.

Enfin, pour un test dynamique, il est nécessaire de composer ces deux fragments et d'extraire les informations nécessaires concernant la cible (`targetN1`=`targetId`, `targetN2` et `targetColor`) des données à dispositions (objets ou tables de méthodes).

Annexe D

Edition de liens

D.1 Compilation directe en langage machine

Pour être applicable en compilation séparée, l'approche globale de la coloration présentée ici nécessite une flexibilité minimale de l'édition de liens :

1. une étape préliminaire doit calculer la coloration proprement dite, à partir des fichiers d'en-tête des classes : cette étape fournit des valeurs pour les différentes « constantes » qui parsèment le code produit par la compilation séparée ;
2. l'édition de liens doit permettre de résoudre ces « constantes » en les remplaçant dans l'exécutable par leur valeur ;
3. l'édition de liens doit aussi permettre des inclusions conditionnelles de code, qui peuvent intervenir dans la compilation de la technique mixte de vérification de types, lorsque l'on ne sait pas si la classe cible appartient ou non au cœur, ainsi que pour la double compilation des accès aux attributs de `self`.

Si l'éditeur de liens ne maîtrise pas le point 3, ou pire, le point 2, il est alors nécessaire d'effectuer un prétraitement avant l'édition de liens proprement dite.

D.2 Compilation dans un langage intermédiaire

L'approche de la coloration peut aussi être adoptée pour une compilation dans un langage intermédiaire comme C.

La compilation du langage source à C est alors purement séparée.

La coloration calcule les constantes qui sont à la base du code C. La résolution de ces constantes et les inclusions conditionnelles ne posent alors plus aucun problème particulier.

En revanche, se pose, comme en SMALL EIFFEL, le problème de la recompilation des fichiers C. Cette recompilation est inutile si le code C n'a pas été modifié et que les constantes qu'il contient n'ont pas été modifiées non plus.

Bibliographie

[Agrawal *et al.*, 1989] R. Agrawal, A. Borgida, et H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data, Portland (OR), USA*, ACM SIGMOD Record, 18(2), pages 253–262, 1989.

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

[Bacon et Sweeney, 1996] D.F. Bacon et P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 324–341. ACM Press, 1996.

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

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

[Day *et al.*, 1995] M. Day, R. Gruber, B. Liskov, et A. Myers. Subtypes vs. where clauses. constraining parametric polymorphism. In *Proc. OOPSLA'95*, pages 156–168. ACM Press, 1995.

[Diwan *et al.*, 1996] A. Diwan, J.E. Moss, et K.S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 292–305. ACM Press, 1996.

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

[Driesen *et al.*, 1995] K. Driesen, U. Hölzle, et J. Vitek. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, éditeur W. Olthoff, LNCS 952, pages 253–282. Springer-Verlag, 1995.

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

[Driesen et Hölzle, 1996] K. Driesen et U. Hölzle. The direct cost of virtual function calls in c++. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 306–323. ACM Press, 1996.

[Driesen, 1999] K. Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. Phd thesis in computer science, University of California, Santa Barbara, 1999.

[Ducournau *et al.*, 1992] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, et A. Napoli. L'héritage multiple dans tous ses états. Rapport de Recherche 92-021, L.I.R.M.M., Montpellier, 1992.

[Ducournau *et al.*, 1994] R. Ducournau, M. Habib, M. Huchard, et M.-L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of OOPSLA'94, Portland (OR), USA*, special issue of ACM SIGPLAN Notices, 29(10), pages 164–175, 1994.

[Ducournau, 1991] R. Ducournau. *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique*. Sema Group, Montrouge, 1991.

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

[Ducournau, 2001a] R. Ducournau. La compilation de l'envoi de message dans les langages statiques. Rapport de Recherche 01-014, L.I.R.M.M., Montpellier, 2001.

[Ducournau, 2001b] R. Ducournau. Spécialisation et sous-typage : thème et variations. Rapport de Recherche 01-013, L.I.R.M.M., Montpellier (à paraître dans *L'Objet*), 2001.

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

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading (MA), USA, 1990.

[Gil et Itai, 1998] J. Gil et A. Itai. The complexity of type analysis of object oriented programs. In *Proc. ECOOP'98*, LNCS 1445, pages 601–634. Springer-Verlag, 1998.

[Golumbic, 1980] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.

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

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

[Liskov *et al.*, 1995] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, et A. C. Myers. THETA reference manual. Technical report, MIT, 1995.

[Meyer, 1997] B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

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

[Odersky et Wadler, 1997] M. Odersky et P. Wadler. Pizza into Java : Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Pugh et Weddel, 1990] W. Pugh et G. Weddel. Two-directional record layout for multiple inheritance. In *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, Special issue of ACM SIGPLAN Notices, 25(6), pages 85–91, 1990.

[Queinnec, 1997] Ch. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 1997.

[Raynaud et Thierry, 2001] O. Raynaud et E. Thierry. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. ECOOP'2001*, LNCS 2072, pages 165–180. Springer-Verlag, 2001.

[Schubert *et al.*, 1983] L.K. Schubert, M.A. Papalaskaris, et J. Taugher. Determining type, part, color and time relationships. *Computer*, 16(10) :53–60, 1983.

[Simon *et al.*, 2000] R. Simon, E. Stapf, C. Mingins, et B. Meyer. Eiffel for e-commerce under .NET. *Journal of Object-Oriented Programming*, pages 42–47, october 2000.

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

[Thierry, 2001] E. Thierry. *Sur quelques interactions entre structure de données et algorithmes efficaces*. Thèse d'informatique, Université Montpellier II, 2001.

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

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

[Wang et Smith, 2001] T. Wang et S.F. Smith. Precise constraint-based type inference for java. In *Proc. ECOOP'2001*, LNCS 2072, pages 99–117. Springer-Verlag, 2001.

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

[Zibin et Gil, 2001] Y. Zibin et J. Gil. Efficient subtyping tests with PQ-encoding. In *Proc. OOPSLA'01*, SIGPLAN Notices, 36(10). ACM Press, 2001.