Comparaison d'algorithmes de construction de hiérarchies de classes

(Godin, R. & Chau, T.-T. (2000). Comparaison d'algorithmes de construction de hiérarchies de classes. *L'objet*, **5**(3), 321-338.)

Robert Godin, Thuy-Tien Chau

Département d'Informatique
Université du Québec à Montréal
C.P.8888, Succursale Centre Ville
Montréal, Québec
Canada, H3C 3P8
godin.robert@uqam.ca

RÉSUMÉ. *Plusieurs algorithmes ont été proposés pour la construction de hiérarchies de classes à partir de la spécification de leurs propriétés. Entre autres, les algorithmes proposés dans [Dicky 1994] et [Godin 1995a] préservent la structure de sous-hiérarchie de Galois de la relation entre les classes et leurs propriétés. De plus ces algorithmes peuvent incorporer de nouvelles classes à une hiérarchie existante. Le résultat est une hiérarchie qui garantit la factorisation maximale des propriétés et la conformité avec la relation de spécialisation entre les classes. Les deux algorithmes ont été implémentés dans un environnement commun afin de comparer leurs performances. Les résultats d'expériences sont rapportés et analysés.*

ABSTRACT. *Many algorithms are proposed for building class hierarchies from the specification of their properties. Among those, the algorithms proposed in [Dicky 1994] and [Godin 1995a] preserve the Galois sub-hierarchy of the relationship between the classes and their properties. Furthermore, the algorithms are incremental and can therefore incorporate a new class within an existing hierarchy. The result is a hierarchy that guarantees the maximal factorization of the properties and conformity to the specialization relationship between classes. Both algorithms are implemented in a common environment and their performance is compared. The results of experiments are reported and analyzed.*

MOTS-CLÉS : *sous-hiérarchie de Galois, algorithme, conception de hiérarchie de classes*

KEY WORDS: *Galois sub-hierarchy, algorithm, class hierarchy design.*

1. Introduction

La conception et la maintenance de la hiérarchie des classes est une des activités les plus difficiles et les plus importantes de l'approche objet [Booch 1994]. Plusieurs chercheurs ont proposé l'utilisation d'algorithmes afin de supporter ces activités [Casais 1991, Dicky 1994, Dicky 1996, Dvorak 1994, Godin 1993, Godin 1998, Lieberherr 1991, Moore 1996]. Parmi ceux-ci, plusieurs se sont intéressés à l'utilisation de structures basées sur la notion de treillis de Galois. Entre autres, les algorithmes proposés dans [Dicky 1994] et [Godin 1995a] produisent de façon incrémentale des hiérarchies qui correspondent à la notion de sous-hiérarchie de Galois. Le fait d'être une sous-hiérarchie de Galois garantit la factorisation maximale des propriétés [Dicky 1994] et la conformité à la relation de spécialisation [Godin 1993]. Ces deux caractéristiques sont généralement vues comme des qualités importantes des hiérarchies de classes [Casais 1991, Dicky 1996, Johnson 1988, Lalonde 1989, Lieberherr 1991, Liskov 1988]. En étant basée sur un cadre théorique bien défini, la hiérarchie produite est donc indépendante de spécificités algorithmiques, ou de paramètres ajustables ou de l'ordre d'arrivée des classes contrairement à beaucoup des travaux sur les méthodes incrémentales de classification conceptuelle [Fisher 1991, Gennari 1990] et de construction de hiérarchies de classes [Casais 1991, Dvorak 1994, Lieberherr 1991]. L'utilisateur aura tendance à avoir plus confiance dans le résultat d'un algorithme si celui-ci repose sur une conceptualisation claire du résultat attendu.

Dans cet article, nous comparons les deux algorithmes de génération incrémentale de la sous-hiérarchie de Galois. La section 2 fait un rappel des notions de base. La section 3 décrit les principales caractéristiques des algorithmes et compare leurs performances.

2. Définitions

Cette section fait un rappel des définitions de base d'un treillis de Galois et d'une sous-hiérarchie de Galois.

2.1. *Treillis de Galois*

La définition de treillis de Galois est présentée à partir d'un exemple portant sur la hiérarchie d'interface d'un ensemble de classes. Plus de détails sur le sujet peuvent être trouvés dans [Barbut 1970, Davey 1992, Wille 1982].

Selon la terminologie de [Wille 1992], un contexte formel est un triplet (G, M, I) où G et M sont deux ensembles finis et I est une relation binaire entre G et M, i.e. $I \subseteq G \times M$. La notation gIm est utilisée pour représenter que $(g,m) \in I$. Étant donnés $A \subseteq G$ et $B \subseteq M$, définissons :

$A' = \{m \in M \mid (\forall g \in A)\ gIm\}$ et

$B' = \{g \in G \mid (\forall m \in B)\ gIm\}$.

Un *concept* du contexte (G, M, I) est défini comme un couple (A, B) où :
$A \subseteq G$, $B \subseteq M$, $A'=B$ et $B'=A$.
Les concepts sont partiellement ordonnés par :
$(A_1, B_1) \leq (A_2, B_2)$ si $A_1 \subseteq A_2$ (ce qui est équivalent à $B_2 \subseteq B_1$).

L'ensemble de tous les concepts du contexte (G, M, I) avec la relation $\leq$ est un treillis appelé *treillis de concept*s (ou treillis de Galois [Barbut 1970]) du contexte et est noté ici $CL(G, M, I)$.

L'ensemble G est habituellement appelé ensemble d'objets et M ensemble d'attributs. Pour l'application qui nous concerne, G représente un ensemble de *classes* et M représente un ensemble de *propriétés* de ces classes. Les propriétés utilisées peuvent être de diverse nature incluant les attributs (variables d'instances), les sélecteurs de messages et les méthodes elles-mêmes. La figure 1 montre un exemple de contexte représenté par une matrice booléenne. Dans l'exemple, seuls les noms des sélecteurs de messages apparaissent comme propriétés des classes. En général, d'autres types de propriétés peuvent être considérés. L'exemple de la figure 1 est extrait de [Cook 1992] et représente un sous-ensemble des interfaces pour un sous-ensemble des classes *Collection* de la librairie de ObjectWorks Smalltalk. Un sélecteur de message est en relation avec une classe par la relation I s'il fait partie de l'interface de la classe, i.e. l'ensemble des messages auxquels la classe peut répondre. La syntaxe exacte des sélecteurs n'est pas respectée dans l'exemple. Les noms donnés sont formés par la concaténation des mots-clés des arguments de chaque sélecteur.

	Set	Bag	Dictionary	Linked List	Array
isEmpty	1	1	1	1	1
size	1	1	1	1	1
includes	1	1	1	1	1
add	1	1	1	1	0
remove	1	1	0	1	0
minus	1	0	1	0	0
addWithOccurrences	0	1	0	0	0
at	0	0	1	1	1
atput	0	0	1	0	1
atAllPut	0	0	0	0	1
first	0	0	0	1	1
last	0	0	0	1	1
addFirst	0	0	0	1	0
addLast	0	0	0	1	0
keys	0	0	1	0	0
values	0	0	1	0	0

Figure 1. *Représentation matricielle du contexte.*

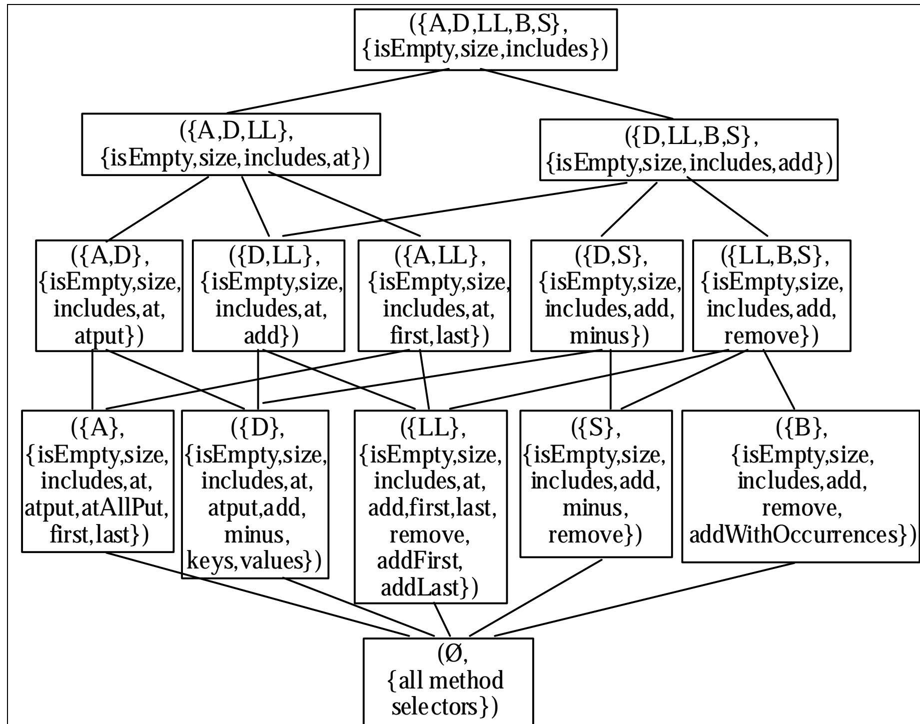

Figure 2. *Treillis de Galois du contexte de la figure 1.*

La figure 2 montre le treillis de Galois correspondant. Seule la première lettre de chacun des noms de classe apparaît dans la figure. L'ordre partiel entre les concepts est utilisé pour générer le graphe de la manière suivante : il y a un arc (C_1, C_2) si $C_1 < C_2$ et s'il n'y a aucun autre concept C_3 dans le treillis tel que $C_1 < C_3 < C_2$. C_1 est appelé *parent* de C_2 et C_2 *enfant* de C_1. Le graphe est habituellement appelé diagramme de Hasse de la relation d'ordre. L'orientation des arcs est implicite (ici vers le haut).

Dans cet exemple, le treillis correspond à la hiérarchie d'interface de l'ensemble des classes. Chacun des concepts du treillis correspond à une classe abstraite qui représente l'interface commune aux classes dans l'extension du concept. Le principe général peut être utilisé pour d'autres types de spécification de classes incluant la distinction entre les méthodes et leur interface, et les relations de spécialisations entre les méthodes. La prise en compte de ces relations est traitée dans [Dicky 1996, Godin 1993, Godin 1998]. De plus, dans [Dicky 1996], on montre comment certains cas de spécialisations entre méthodes peuvent être identifiés automatiquement à partir du code source.

Le treillis peut donc être utilisé comme point de départ à la conception d'une hiérarchie sans redondance qui respecte la relation de spécialisation entre les classes. Chaque concept du treillis est un candidat potentiel de classe. Ainsi, le treillis de Galois peut être considéré comme un idéal à atteindre un peu comme les formes normales en conception de bases de données. En pratique comme pour la

normalisation, ce découpage peut devenir excessif. Diverses alternatives d'élagage du treillis sont à considérer. Une alternative intéressante est la sous-hiérarchie de Galois présentée à la sous-section 2.3.

2.2. *Treillis d'héritage*

Le treillis de Galois tel que représenté à la section précédente n'est pas adéquat pour les hiérarchies de classes à cause de la redondance de cette représentation. Pour un concept $C = (A, B)$, A est présent dans tous les ancêtres de C et, symétriquement, B apparaît dans tous ses descendants. Pour la conception des hiérarchies de classes, cette duplication doit être éliminée car le mécanisme d'héritage permet de retrouver l'information. Pour un concept $C = (A, B)$, notons par AN les éléments non redondants de A, et BN les éléments non redondants de B.

Un *treillis d'héritage* de Galois est représenté par l'ensemble des couples (AN, BN). La figure 3 est le treillis d'héritage correspondant au treillis de la Figure 1. Du point de vue de la conception des classes, l'idée consiste à considérer les concepts comme des classes et le graphe comme la relation d'héritage. BN constitue donc l'ensemble des propriétés à déclarer dans la classe. Comme chaque propriété n'apparaît qu'une seule fois, une factorisation maximale des propriétés communes des classes initiales est obtenue. Les concepts pour lesquels AN est vide correspondent à de nouvelles classes abstraites qui mettent en facteur les propriétés communes à un sous-ensemble des classes utilisées en entrée. Le résultat est une hiérarchie où chaque interface n'est déclarée qu'à un seul endroit. Ces principes peuvent être étendus à des spécifications où la surcharge des sélecteurs est prise en compte. On obtiendra ainsi une hiérarchie où chacune des interfaces et chacune des méthodes est déclarée à un et un seul endroit dans la hiérarchie.

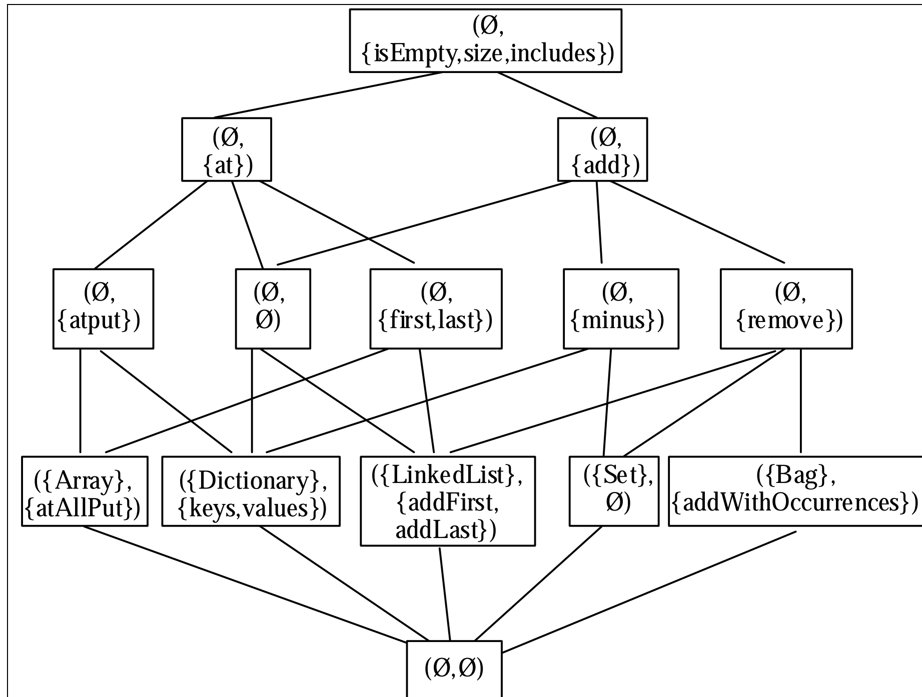

Figure 3. *Treillis d'héritage.*

2.3. *Sous-hiérarchie de Galois*

Pour le problème de conception des hiérarchies de classes, il peut être très avantageux de considérer un sous-ensemble du treillis de Galois. En particulier, les nœuds vides ($AN = \varnothing$ et $BN = \varnothing$) peuvent être éliminés du treillis d'héritage sans perdre d'information. Le résultat est appelé sous-hiérarchie de Galois [Dicky 1994]. La complexité de la sous-hiérarchie peut être de beaucoup inférieure à celle du treillis entier. La différence de taille entre les deux structures dépend de la nature de la relation *I*. Des expériences effectuées sur des hiérarchies de classes [Godin 1998] ont démontré que la différence peut devenir très importante, en particulier, lorsque la relation *I* est très dense et que le nombre de combinaisons potentielles devient large.

Ainsi, la sous-hiérarchie de Galois peut, dans certains cas, être beaucoup plus compacte que le treillis mais préserver les caractéristiques de factorisation maximale et de conformité à la relation de spécialisation. L'inconvénient est que la structure n'est pas nécessairement un treillis et que certaines abstractions potentiellement utiles sont éliminées. Le fait de préserver la structure de treillis simplifie, entre autres, certains aspects de l'inférence de type pour un compilateur

[Caseau 1993]. L'élimination de nœuds vides peut dans certains cas réduire la réutilisabilité de la hiérarchie et même en augmenter la complexité.

La figure 4 montre la sous-hiérarchie correspondant à l'exemple de la figure 3. On peut remarquer que deux concepts vides ont été éliminés. L'élimination du nœud le plus bas est sans conséquence dans le cas des hiérarchies de classes. Par contre, le nœud correspondant à l'abstraction « collection extensible et indexée », extensible puisqu'il hérite de *add*, indexée puisqu'il hérite de *at* peut être utile à conserver. Du point de vue réutilisabilité, c'est une abstraction conceptuellement significative. Du point de vue de la complexité de la hiérarchie, cependant, l'élimination de ce nœud est bénéfique car il n'est pas nécessaire d'ajouter de nouvelles relations d'héritage pour compenser la perte du nœud. Ceci est du au fait que, dans le treillis, les propriétés héritées par les classes qui héritent du nœud vide, soit les classes *Dictionary* et *LinkedList*, sont aussi héritées par un autre chemin (héritage répété). Par exemple, dans le treillis de la figure 3, la classe *Dictionary* hérite du sélecteur *at* par deux chemins dont un des deux passe par un nœud vide. La propriété est toujours héritée correctement dans la sous-hiérarchie de la figure 4 malgré l'élimination du nœud vide.

Il y a des cas par contre où l'élimination d'un nœud vide conduit à une hiérarchie beaucoup plus complexe. Par exemple, si dans la hiérarchie correspondant au treillis, *n* classes héritent toutes d'une seule classe vide qui elle-même hérite de *m* classes, l'élimination de la classe vide nécessite l'ajout de *nm* relations d'héritage. Le résultat est une hiérarchie beaucoup plus complexe en nombre de relations d'héritage : *nm* par rapport à *n+m*.

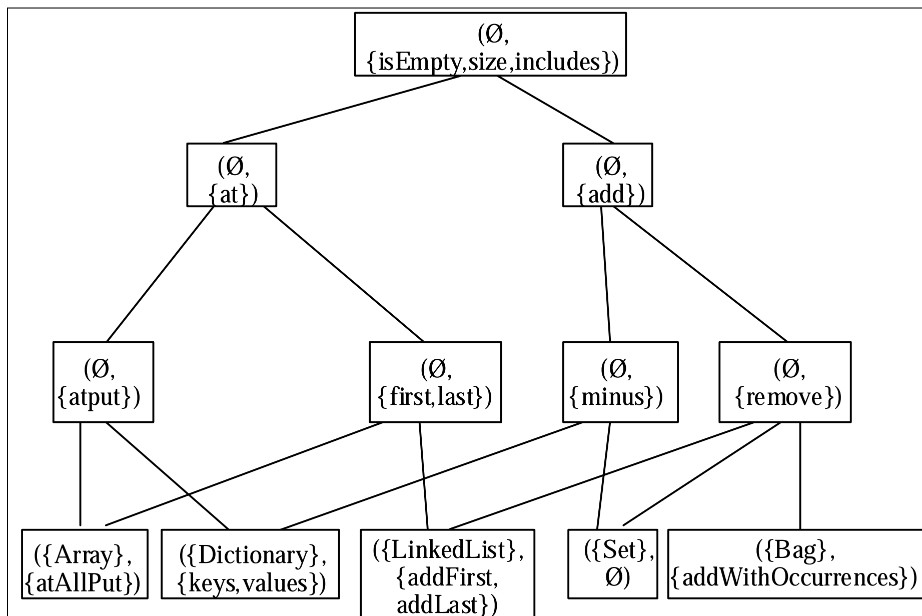

Figure 4. *Sous-hiérarchie de Galois pour l'exemple de la figure 1.*

3. Algorithmes et expériences

Deux algorithmes ont été proposés pour générer la sous-hiérarchie de Galois de façon incrémentale [Dicky 1994, Godin 1995a]. Avec ces algorithmes, on peut insérer une nouvelle classe dans une hiérarchie existante en produisant si nécessaire de nouvelles classes de factorisation et en modifiant la topologie du graphe d'héritage en conséquence. La sous-section suivante décrit l'algorithme ISGOOD en faisant ressortir les différences majeures avec ARES et la sous-section 3.2 compare la performance des deux algorithmes.

3.1. *Algorithmes*

L'algorithme ISGOOD est essentiellement le même algorithme que l'algorithme proposé dans [Godin 1995a]. La structure générée élimine non seulement les concepts vides ($AN = \emptyset$ et $BN = \emptyset$) mais tous les concepts dont l'ensemble des propriétés BN est vide. Il manque donc certains concepts par rapport à la sous-hiérarchie de Galois, i.e. ceux où $AN \neq \emptyset$ et $BN = \emptyset$. Une modification doit donc être apportée afin de forcer le maintien d'une classe *pertinente* même si l'ensemble des propriétés BN de la classe est vide. A cet effet, une propriété supplémentaire factice qui est unique à chacune des classes pertinentes est ajoutée. Ainsi, une classe pertinente a au moins la propriété identifiante qui lui est spécifique et ne sera donc jamais vidée de toutes ses propriétés. Les classes de base utilisées en entrée sont habituellement considérées comme étant pertinentes car ce sont normalement des classes instanciables. On veut donc qu'elles apparaissent explicitement dans la hiérarchie.

Cette approche conduit à une différence mineure avec ARES. En effet, comme chacune des classes en entrée possède une propriété qui lui est spécifique, le concept correspondant dans la sous-hiérarchie produite par ISGOOD doit nécessairement être une feuille. En conséquence, seules les classes feuilles sont instanciables.

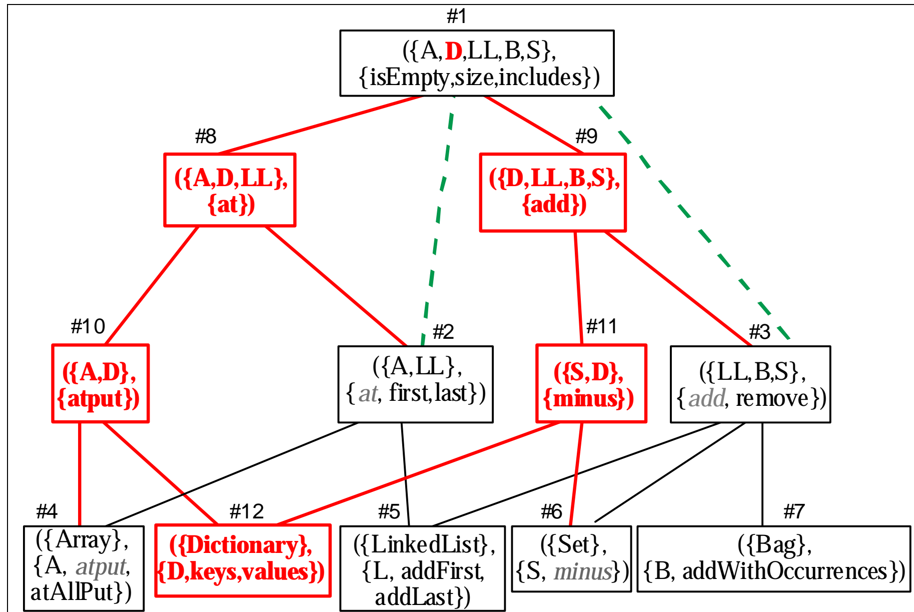

Figure 5. *Modification incrémentale de la sous-hiérarchie de Galois.*

La figure 5 montre les modifications effectuées par ISGOOD pour insérer la nouvelle classe *Dictionary* à la sous-hiérarchie de Galois qui aurait été précédemment produite pour l'ensemble des classes *{Array, Linked List, Bag, Set}*. Dans cet exemple, la propriété identifiante utilisée pour les classes pertinentes est la première lettre du nom de la classe. Ainsi la première lettre du nom de la classe apparaît aussi dans les propriétés (*BN*) des concepts correspondants.

Sauf pour la propriété identifiante, les propriétés sont les mêmes que dans les exemples précédents. La nouvelle classe *Dictionary* possède donc l'ensemble des propriétés *{D, isEmpty, size, includes, add, minus, at, atput, keys, values}* qui correspond à son ensemble de sélecteurs auquel on ajoute la propriété identifiante *D* (première lettre de *Dictionary*).

Dans la figure 5, chacun des concepts correspond au couple (*A*, *BN*) plutôt que (*AN*, *BN*) tel que défini pour la sous-hiérarchie de Galois. En effet, il est nécessaire de calculer l'ensemble des classes (*A*) ou l'ensemble des propriétés (*B*) en extension pour le traitement de la mise à jour du graphe. On peut maintenir cet ensemble explicitement ou le recalculer au besoin. Nous avons choisi de maintenir *A* plutôt que de le calculer, le coût en espace étant raisonnable. On aurait pu choisir de maintenir *B* en extension. C'est cette dernière alternative qui est utilisée dans ARES. Pour alléger la figure, sauf pour les feuilles, seule la première lettre de chacun des noms de classe apparaît dans *A*.

Dans l'algorithme, la nouvelle classe à ajouter en entrée est notée *c* et son ensemble de propriétés est *c*'. L'algorithme spécifie comment mettre à jour la sous-

hiérarchie de Galois. Les concepts de la sous-hiérarchie sont considérés comme des classes du point de vue de la conception de la hiérarchie. Cependant, pour éviter la confusion, le terme concept est utilisé pour faire référence aux éléments de la sous-hiérarchie de Galois dans l'algorithme ISGOOD.

Pour un concept C, $A(C)$ représente son ensemble de classes et $BN(C)$ son ensemble de propriétés. Les lignes 1 à 16 de l'algorithme déterminent les changements à effectuer aux concepts existants et créent de nouveaux concepts si nécessaire. Dans la figure 5, les nouveaux concepts sont représentés en gras. Seuls les concepts ayant au moins une propriété en commun avec la nouvelle classe, *c*, sont visitées. L'accès à ces concepts est effectué à l'aide d'un index *Ind* qui pour chaque propriété, *m*, indique le concept où elle se trouve, *Ind*(*m*). Au départ, l'algorithme produit la partition des propriétés de *c* en parties P_i qui se trouvent dans le même concept C_i. Les nouvelles propriétés qui n'apparaissent dans aucun concept de la hiérarchie sont rassemblées dans *N*. Ensuite ces concepts sont visités un par un dans un ordre quelconque. Dans l'exemple, la partition suivante est produite :

P_1 = {at}:	concept #2
P_2 = {add}:	concept #3
P_3 = {atput}:	concept #4
P_4 = {minus}:	concept # 6
N ={D, keys, values}:	nouvelles propriétés.

ALGORITHME *ISGOOD*

Entrée: c : nouvelle classe;
c' : l'ensemble des propriétés de la nouvelle classe;
index *Ind* : *Ind*(m) pointe vers le concept contenant la propriété m;
H : la hiérarchie à mettre à jour

Sortie: H : la hiérarchie mise à jour
Ind : index mis à jour

BEGIN

1. *NouveauxConcepts*:=Ø; *ConceptsModifiées*:=Ø;
2.

{Partitionner c' en sous-ensembles P_i de propriétés qui sont dans le même concept C_i de H , i.e. même valeur de *Ind*(m) pour toutes les propriétés m de P_i}

3. **POUR** toute propriété p de c'
4.   **SI** il existe une partie P_i avec $Ind(p)= C_i$,
5.   Ajouter p a P_i,
6.   **SINON**
7.   Créer une partie $P_i =\{p\}$
8.   **FINSI**
9. **FINPOUR;**
10. N:= l'ensemble des nouvelles propriétés de c';

{Modifier les concepts existants et créer les nouveaux concepts de factorisation}

11. **POUR** chacune des parties P_i
12.   {le concept correspondant est noté C_i}
13.   **SI** $BN(C_i) \subseteq c'$ {Cas d'un concept modifié}
14.   Ajouter c à $A(C_i)$; Ajouter C_i à *ConceptsModifiés*
15.   **SINON** {Cas d'un nouveau concept}
16.   Créer nouveau concept $C = (A(C_i) \cup \{c\}, P_i)$;
17. $BN(C_i) := BN(C_i) - P_i$; $\forall m \in P_i$: $Ind(m):=C$;
18.   Ajouter arc (C, C_i); Ajouter C à *NouveauxConcepts*
19.   **FINSI**
20. **FINPOUR**;
21. **SI** $N \neq Ø$
22.   Créer nouveau concept $C_0=(\{c\}, N)$; $\forall m \in N$: $Ind(m):=C$;
23.   Ajouter C_0 à *NouveauxConcepts*
24. **FINSI**;

{Remplacer le lien entre le générateur et un parent modifié par un lien entre le nouveau concept et le parent modifié}

25\. **POUR** chaque $C \in$ *NouveauxConcepts excluant* C_0
26\.   **POUR** l'unique enfant C_g *de* C
20\.   **POUR** chaque parent C_p de C_g
27\.   **SI** $C_p \in$ *ConceptsModifiées*
28\.   Supprimer arc (C_p, C_g);

```
29.                     Ajouter arc (Cp, C)
30.             FINSI
25.         FINPOUR
26.     FINPOUR
31. FINPOUR;
32. ConceptsTraitées:=ConceptsModifiées ∪ NouveauxConcepts;
```

{Mettre à jour les arcs entre concepts modifiés et leurs parents qui n'ont pas été modifiés}

```
33. POUR chaque classe C ∈ ConceptsModifiés
34.     POUR chaque parent Cp de C
35.         SI Cp ∉ ConceptsModifiés
36.             Supprimer arc (Cp,C);EP=Ø;ChercherEnfantsPotentiels(EP, C);
37.             POUR chaque Ce ∈ EP
38.                 SI il n'existe pas de D ∈ EP tel que A(D) ⊃ A(Ce)) ET
39.                    il n'existe pas un enfant D de Cp tel que A(D) ⊃ A(Ce))
                    {Pour éviter les arcs de transitivité}
40.                     Ajouter arc (Cp , Ce)
41.                 FINSI
42.             FIN POUR
43.         FINSI
44.     FINPOUR
45. FINPOUR;
```

{Ajouter les liens qui manquent entre les concepts traités}

```
46. POUR chaque C∈ ConceptsTraitées
47.     POUR chaque D ∈ ConceptsTraitées
48.         SI D n'est pas un enfant de C et A(C) ⊃ A(D)
49.             SI il n'existe pas de E ∈ ConceptsTraitées tel que
50.                     A(C) ⊃ A(E) ⊃ A(D)
51.                 Ajouter arc (C,D)
52.             FINSI
53.         FINSI
54.  FINPOUR
55. FINPOUR;
56. H := H ∪ NouveauxConcepts
  FIN

  PROCÉDURE ChercherEnfantsPotentiels(EP,C)
  DÉBUT
      POUR chaque enfant D de C
          SI D ∈ ConceptsTraités
              ChercherEnfantsPotentiels(EP,D)
          SINON
```

Ajouter D à EP
FINSI
FINPOUR
FIN

Un concept C_i est modifié si $BN(C_i) \subseteq c'$ (lignes 13-14). Dans ce cas, c est ajouté à son ensemble $A(C_i)$. C'est le cas du concept #1. Dans le cas contraire, les propriétés communes sont extraites pour en faire un nouveau concept de factorisation (lignes 15-19). Dans la figure 5, les propriétés extraites apparaissent en italique. Les concepts d'où sont extraites les propriétés sont en quelque sorte les *générateurs* des nouveaux concepts. Les classes #8, #9, #10 et #11 sont ainsi créées à partir des générateurs, #2, #3, #4, #6 respectivement. Un nouveau concept devient toujours un parent de son générateur. Un nouveau concept est aussi créé si de nouvelles propriétés sont présentes (lignes 21-24). Par exemple, le concept #12 est créé pour les nouvelles propriétés *{D, keys, values}*. Dans le cas d'une classe pertinente, la propriété identifiante provoque toujours la création d'un nouveau concept.

Le reste de l'algorithme (ligne 25-56) s'occupe de mettre à jour les arcs du graphe. Dans la figure 5, les nouveaux arcs sont en gras et les arcs à éliminer sont en pointillés. Dans les lignes 25-32, le lien entre un concept générateur et un parent qui a été modifié est remplacé par l'arc entre le nouveau concept et le parent modifié. Par exemple, l'arc (#1,#2) est remplacé par (#1,#8) et (#1,#3) par (#1,#9). La section 33-45 supprime les arcs entre un concept modifié et un parent non modifié. Ce parent doit, dans certains cas, être relié à des sous-concepts qui n'ont pas été modifiés. La section 46-56 ajoute les liens qui manquent entre les concepts traités, par l'algorithme naïf qui effectue une vérification exhaustive des combinaisons possibles. Comme le nombre de concepts traités est restreint, cette stratégie est acceptable.

L'utilisation d'une extension linéaire dans ARES a pour but de garantir qu'avant de visiter un concept, tous ses ancêtres seront visités. L'algorithme compare alors les propriétés déclarées dans un concept avec les propriétés de la nouvelle classe afin de déterminer s'il est nécessaire de créer un nouveau concept de factorisation en extrayant du concept existant les propriétés communes. Ceci peut avoir pour effet de vider un concept existant qui devra être éliminé par la suite sauf si la nouvelle classe a été marquée comme étant *pertinente*. Dans ISGOOD, plutôt que de créer un nouveau concept de factorisation et de vider le concept visité lorsque toutes les propriétés déclarées par le concept visité sont incluses dans c', le concept existant est tout simplement modifié (cas des concepts modifiés). L'impact de cette différence de stratégie est cependant assez mineur car le nombre de concepts touchés est encore borné par k.

Dans ARES, au fur et à mesure de la visite de la hiérarchie, on construit l'ensemble des ancêtres du nouveau concept, noté *SH*. Cet ensemble sert à la mise à jour des arcs du graphe. L'ordre de visite des nœuds garantit que les parents d'un concept seront déjà dans l'ensemble *SH* au moment de visiter le concept. La

recherche des parents est donc effectuée au moment de la création du nouveau concept. Il est intéressant de noter que cette stratégie est très semblable à celle proposée dans [Godin 1995b] pour générer le treillis de Galois. Ceci est un avantage par rapport à ISGOOD qui visite les concepts dans un ordre quelconque. Il est donc nécessaire dans ISGOOD d'attendre que tous les concepts soient visités avant d'effectuer le travail de mise à jour des parents d'un nouveau concept. Le résultat est une complexité accrue du fait que l'ensemble des concepts à considérer n'est pas contraint par le contexte. L'impact est cependant assez limité car le nombre de concepts touchés est borné par k.

3.1. *Comparaison du coût des algorithmes*

Le nombre de concepts visités par ISGOOD pour la section des lignes 11-24 de l'algorithme est borné par $|c'|$, le nombre de propriétés de c. On suppose que la recherche dans l'index *Ind* est effectuée en temps constant. En pire cas, chaque propriété de c sera dans un concept différent de la hiérarchie existante. Ceci implique par ailleurs que le nombre de nouveaux concepts générés est aussi borné par $|c'|$. Au total, le nombre de concepts h de la sous-hiérarchie de Galois est donc borné par kn où n est le nombre de classes en entrée utilisées pour produire la hiérarchie et k est une borne supérieure sur le nombre de propriétés par classe. Le test d'inclusion de ligne 5 peut-être réalisé en $\boldsymbol{O}(k)$. On obtient donc un coût $\boldsymbol{O}(k^2)$ pour la section 11-24.

La section 25-32 est $\boldsymbol{O}(k^3)$ car le nombre de nouveaux concepts, le nombre de concepts modifiés et le nombre de parents est borné par k.

La section 46-56 est aussi $\boldsymbol{O}(k^3)$ car le nombre de parents d'un concept est borné par k. En effet, tous les parents d'un concept C doive contenir une propriété pour chacune des classes de $A(C)$ et le nombre de propriétés est borné par k pour chacune d'elles.

Le seul traitement qui dépend du nombre total de concepts h dans la hiérarchie est la section 33-45. Contrairement au reste de l'algorithme, ce traitement n'est pas borné par un polynôme en k, car le nombre d'enfants d'un concept n'est pas borné par k mais par h. En pratique, ce pire cas est cependant assez rare. En effet, comme le nombre de parents d'un concept est borné par k, le nombre moyen d'enfants est aussi borné par k. On peut donc espérer qu'en moyenne, le traitement dépend plutôt de k. Les résultats obtenus dans les expériences sont en accord avec cette hypothèse. Le pire cas est $\boldsymbol{O}(k^4n^2)$. Ce coût est expliqué d'une part par la recherche des enfants des parents de chaque concept modifié dans la procédure *ChercherEnfantsPotentiels*. Le nombre d'enfant est borné par $h \leq kn$. Cette recherche est effectué k^2 fois au maximum. Et il faut ensuite itérer sur ces enfants dans le test de la ligne 38. En moyenne, on peut espérer un coût $\boldsymbol{O}(k^4)$. Donc au total, en moyenne, le coût serait $\boldsymbol{O}(k^4)$.

Contrairement à ISGOOD, ARES effectue le travail en parcourant tous les concepts de la hiérarchie initiale selon une extension linéaire de la relation d'ordre

partiel. Le fait de parcourir toute la hiérarchie implique une complexité de l'ordre de grandeur de la hiérarchie, h. L'analyse détaillée donnée dans [Dicky 1994] montre que le coût de ARES est $\boldsymbol{O}(m + h\ (kw + s))$. Les paramètres sont :

w : largeur de la sous-hiérarchie

m : nombre total d'arcs

s : un facteur borné par le nombre de parents du concept qui contiendra c.

Le nombre d'arcs de la sous-hiérarchie, m, est borné par k^2n car h est borné par kn et le nombre de parents de chaque concept est borné par k. La largeur w est bornée par kn. Le nombre de parents d'un concept, s, est borné par k. On obtient donc un coût $\boldsymbol{O}(k^3n^2)$.

Cette analyse montre un pire cas plus intéressant pour ARES. Cependant, si l'hypothèse tient au sujet du nombre moyen d'enfants visités en pratique par la procédure *ChercherEnfantsPotentiels*, le résultat sera plus avantageux pour ISGOOD. En effet, le coût moyen sera $\boldsymbol{O}(k^4)$ qui ne dépend aucunement du nombre de classes en entrées n. Le coût d'insertion ne devrait donc pas se détériorer au fur et à mesure de la croissance de la hiérarchie.

Afin de comparer les performances réelles des deux algorithmes, nous les avons implémentés dans un environnement commun : ObjectWorks/Smalltalk. Des expériences ont été effectuées sur diverses librairies de classes sur une plate-forme PC munie d'un processeur Pentium 120. La figure 6 montre le temps cumulatif pour construire incrémentalement la sous-hiérarchie de Galois pour un ensemble de 64 classes de la sous-hiérarchie des *Collections* de ObjectWorks/Smalltalk. Chacune des classes est décrite par son protocole, i.e. l'ensemble des sélecteurs de message auxquels la classe répond sans retourner une erreur. Le protocole a été extrait par une analyse automatique. La figure 6 montre que le temps de traitement pour ISGOOD est inférieur à ARES lorsque le nombre de classes dépasse la vingtaine et que la différence augmente avec le nombre de classes traitées. La figure inclut aussi une analyse de régression polynomiale d'ordre 2 par rapport au nombre de classes traitées. La variable x de l'équation de régression représente le nombre de classes en entrée qui ont été utilisées pour produire la hiérarchie (et non pas le nombre de concepts de la hiérarchie). Quoiqu'assez faible, la composante quadratique pour ARES est plus importante que pour ISGOOD, d'où l'accroissement de la différence. Dans le cas de ISGOOD, au delà de la trentaine de classes, le temps de traitement pour ajouter une nouvelle classe est presque constant. On obtient donc un accroissement presque linéaire du temps cumulatif de traitement.

Ceci confirme l'hypothèse faite précédemment dans l'analyse de complexité du cas moyen de ISGOOD au sujet du nombre moyen d'enfants. La composante quadratique de ARES est explicable essentiellement par la nécessité de visiter tous les concepts de la hiérarchie alors que ISGOOD ne visite qu'un ensemble limité de concepts borné par k qui ne dépend pas du nombre n de classes de base traitées.

On peut observer que malgré cette différence, le temps de traitement pour les deux algorithmes est très raisonnable. Le temps nécessaire pour ajouter une nouvelle classe est inférieur à une seconde par classe en moyenne malgré le fait que

le nombre moyen de propriétés soit assez grand dans cette librairie, soit de plus d'une centaine par classe en moyenne. D'autres expériences portant sur d'autres librairies ont donné des résultats semblables. La figure 7 montre le résultat d'une expérience similaire sur un autre jeu d'essai d'un domaine différent. Les objets correspondent à des éléments de données d'un dictionnaire de données pour une application bancaire et les attributs à des mots-clés descriptifs. Le nombre d'objets est beaucoup plus grand mais le nombre d'attributs est plus petit que pour l'expérience précédente. On peut observer un comportement relatif semblable. Le temps de traitement obtenu dans les expériences permet de conclure à la faisabilité d'un outil interactif d'aide à la conception basé sur ces algorithmes dans le cas de hiérarchies de grande taille.

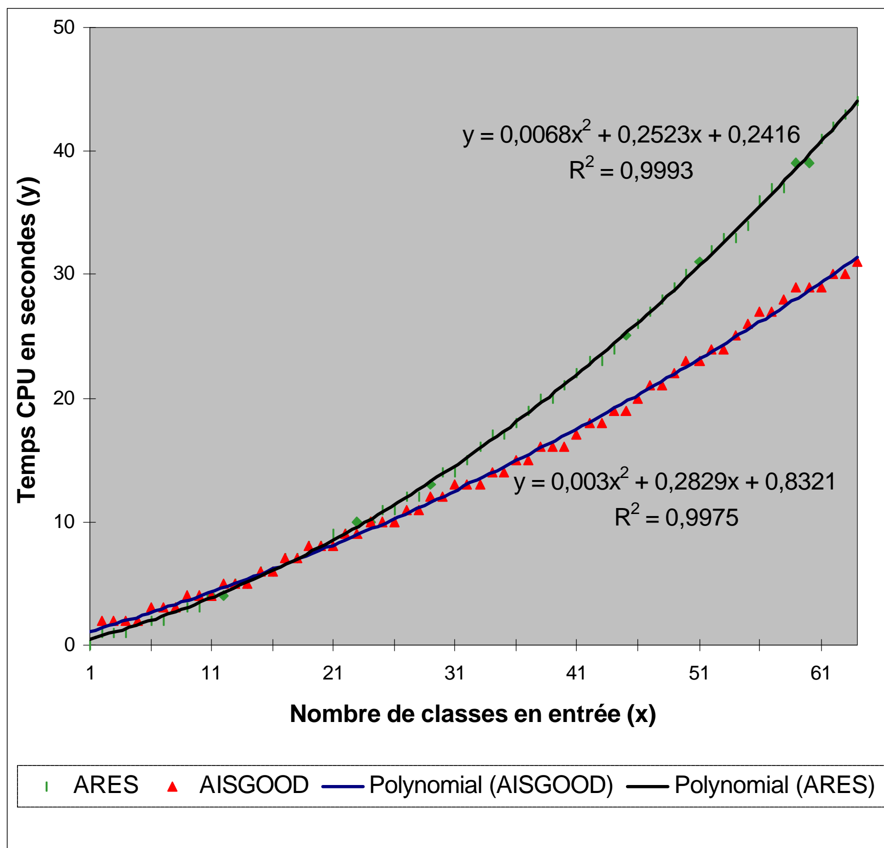

Figure 6. *Modification incrémentale de la sous-hiérarchie de Galois pour les classes Collections.*

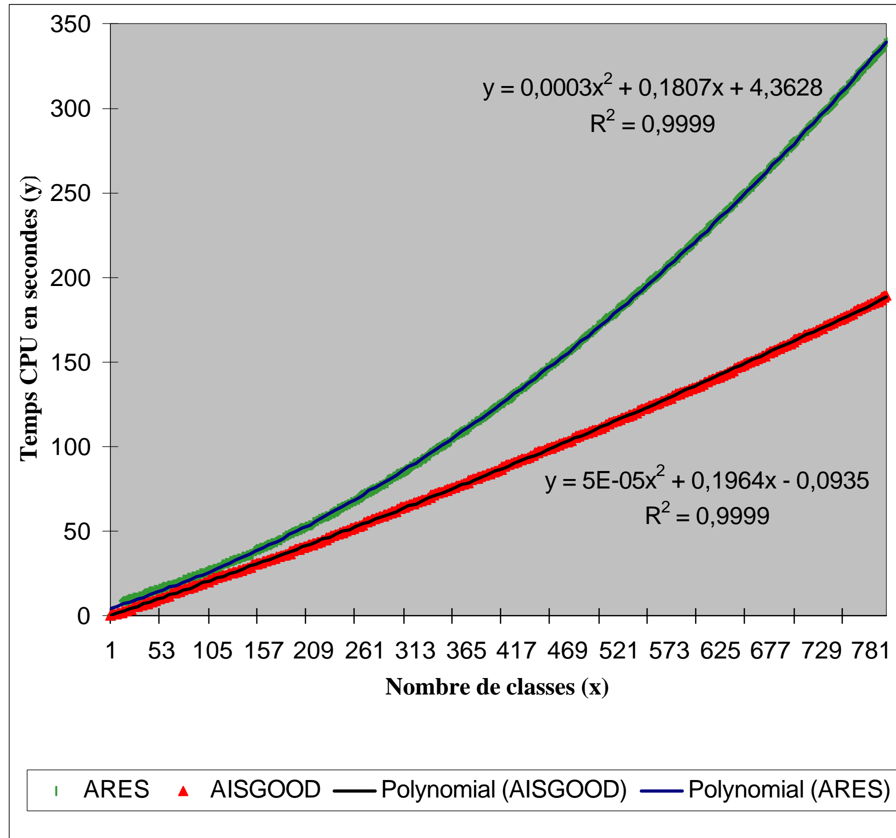

Figure 7. *Modification incrémentale de la sous-hiérarchie de Galois pour l'application des éléments de données.*

4. Conclusion

Nous avons comparé les algorithmes ARES [Dicky 1994] et ISGOOD [Godin 1995a] de construction de hiérarchies de classes basées sur la notion de sous-hiérarchie de Galois. Ces algorithmes peuvent incorporer de façon incrémentale de nouvelles classes à une hiérarchie existante. Le résultat est une hiérarchie qui garantie la factorisation maximale des propriétés et la conformité avec la relation de spécialisation.

L'analyse de la complexité de l'insertion d'une nouvelle classe dans une hiérarchie existante favorise ARES dont le coût est $\boldsymbol{O}(k^3n^2)$ contre $\boldsymbol{O}(k^4n^2)$ pour ISGOOD, n étant le nombre de classes utilisées en entrée pour produire la hiérarchie et k étant le nombre maximal de propriétés par classe.

Cependant, dans le cas moyen, le nombre d'enfants d'une classe devrait plutôt être borné par k. Sous cette hypothèse, le résultat est une complexité moyenne qui est $\boldsymbol{O}(k^4)$ pour ISGOOD. Dans les expériences effectuées, le temps de traitement moyen pour ISGOOD est inférieur à ARES lorsque le nombre de classes dépasse un certain seuil et la différence va en grandissant avec le nombre de classes traitées. Au delà d'un certain point, le temps d'insertion d'une nouvelle classe est presque constant pour ISGOOD. Pour ARES, le coût inclut une composante linéaire en n non négligeable. La composante linéaire pour ARES vient principalement de la nécessité de visiter toutes les classes de la hiérarchie existante alors que ISGOOD ne visite qu'un ensemble limité de classes dont la taille est bornée par le nombre de propriétés k de la nouvelle classe à insérer.

Le temps de traitement raisonnable obtenu dans les expériences permet de conclure à la faisabilité d'un outil interactif d'aide à la conception basé sur ces algorithmes dans le cas de hiérarchies de grande taille.

5. Bibliographie

[Barbut 1970] Barbut, M., Monjardet, B., *Ordre et Classification. Algèbre et Combinatoire, Tome II*, Hachette, 1970.

[Booch 1994] Booch, G., *Object-Oriented Analysis and Design*, Benjamin Cummings, 1994.

[Casais 1991] Casais, E., « Managing Evolution in Object Oriented Environments: An Algorithmic Approach », Thèse de doctorat, Geneva, 1991.

[Caseau 1993] Caseau, Y., « Efficient Handling of Multiple Inheritance Hierarchies », *Actes de ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, Washington, DC, 1993, ACM Press, p. 271-287.

[Cook 1992] Cook, W. R., « Interfaces and Specifications for the Smalltalk-80 Collection Classes », *Actes de Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, B.C., Canada, 1992, ACM Press, p. 1-15.

[Davey 1992] Davey, B. A., Priestley, H. A., *Introduction to Lattices and Order*, Cambridge University Press, 1992.

[Dicky 1994] Dicky, H., Dony, C., Huchard, M., Libourel, T., « Un algorithme d'insertion avec restructuration dans les hiérarchies de classes », *Actes de Langages et Modèles à Objets*, Grenoble, 1994, p.

[Dicky 1996] Dicky, H., Dony, C., Huchard, M., Libourel, T., « On Automatic Class Insertion with Overloading », *Actes de ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, CA, USA, 1996, ACM SIGPLAN Notices, p. 251-267.

[Dvorak 1994] Dvorak, J., « Conceptual Entropy and Its Effect on Class Hierarchies », *IEEE Computer* , vol. 27, n° 6, 1994, p. 59-63.

[Fisher 1991] Fisher, D. H., Pazzani, M. J., Computational Models of Concept Learning, in *Concept Formation: Knowledge and Experience in Unsupervised Learning,* D. H. Fisher, M. J. Pazzani, P. Langley (Eds.), Morgan Kaufmann, p. 3-44, 1991.

[Gennari 1990] Gennari, J. H., Langley, P., Fisher, D., Models of Incremental Concept Formation, in *Machine Learning: Paradigms and Methods,* J. Carbonell (Eds.), MIT Press, p. 11-62, 1990.

[Godin 1993] Godin, R., Mili, H., « Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices », *Actes de ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, Washington, DC, 1993, ACM Press, p. 394-410.

[Godin 1998] Godin, R., Mili, H., Mineau, G. W., Missaoui, R., Arfi, A., Chau, T.-T., « Design of Class Hierarchies based on Concept (Galois) Lattices », *Theory and Application of Object Systems* , vol. 4, n° 2, 1998, p. 117-134.

[Godin 1995a] Godin, R., Mineau, G. W., Missaoui, R., « Incremental structuring of knowledge bases », *Actes de International Knowledge Retrieval, Use, and Storage for Efficiency Symposium (KRUSE'95)*, Santa Cruz, 1995a, Springer-Verlag's Lecture Notes in Artificial Intelligence, p. 179-198.

[Godin 1995b] Godin, R., Missaoui, R., Alaoui, H., « Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices », *Computational Intelligence* , vol. 11, n° 2, 1995b, p. 246-267.

[Johnson 1988] Johnson, R., Foote, B., « Designing Reusable Classes », *Journal of Object-Oriented Programming* , vol. June/July, 1988, p. 22-35.

[Lalonde 1989] Lalonde, W. R., « Designing Families of Data Types Using Exemplars », *ACM Trans. on Prog. Languages and Systems* , vol. 11, n° 2, 1989, p. 212-248.

[Lieberherr 1991] Lieberherr, K. J., Bergstein, P., Silva-Lepe, I., « From Objects to Classes: Algorithms for Optimal Object-Oriented Design », *Journal of Software Engineering* , vol. 6, n° 4, 1991, p. 205-228.

[Liskov 1988] Liskov, B., « Data abstraction and hierarchy », *ACM SIGPLAN Notices* , vol. 23, n° 5, 1988, p. 17-34.

[Moore 1996] Moore, I., « Automatic Inheritance Hierarchy Restructuring and Method Refactoring », *Actes de ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, CA, USA, 1996, ACM SIGPLAN Notices, p. 235-250.

[Wille 1982] Wille, R., Restructuring Lattice Theory: an Approach Based on Hierarchies of Concepts, in *Ordered Sets,* I. Rival (Eds.), Reidel, p. 445-470, 1982.

[Wille 1992] Wille, R., « Concept Lattices and Conceptual Knowledge Systems », *Computers & Mathematics with. Applications* , vol. 23, n° 6-9, 1992, p. 493-515.

Robert Godin *est professeur au département d'informatique de l'université du Québec à Montréal (UQAM) depuis 1983. Il a reçu un Ph.D.. de l'Université de Montréal en 1986. Ses travaux de recherche touchent aux domaines suivants : bases de données (prospection de données), repérage de l'information, génie du logiciel, intelligence artificielle (classification conceptuelle). Ses travaux sont subventionnés par le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG) depuis 1987. Il a participé à plusieurs grands projets de R&D. Il compte à son actif une quarantaine de publications.*

Thuy-Tien Chau *a obtenu une maîtrise en informatique de gestion à l'Université du Québec à Montréal en 1997. Elle est présentement architecte réseautique chez ISM/Services Mondiaux IBM. Elle possède 12 années d'expérience dans les environnements réseautiques et les logiciels de télécommunications.*