Mémoire en vue de la soutenance
d'une habilitation à diriger des recherches

Classification de classes dans les approches à objets

algorithmes et applications

Marianne HUCHARD

4 avril 2003

Table des matières

Remerciements

Mes remerciements iront tout d'abord aux rapporteurs de ce mémoire, Yves Caseau, Pierre Cointe et Jean-François Perrot pour la confiance qu'ils m'ont accordée et pour la justesse de leurs remarques et de leurs suggestions.

Jean-Claude Royer était déjà présent dans le jury de ma thèse de doctorat : qu'il considère dans cette habilitation à diriger des recherches la suite d'une aventure ...

Amedeo Napoli a veillé comme un jardinier sérieux et constructif sur nombre de mes réalisations : articles, organisation de workshops, etc. Qu'il retrouve dans ce document beaucoup d'idées qu'il a semées, et qu'il prépare des tuteurs pour ce qui a poussé de travers !

Merci à Roland Ducournau, son stylo rouge et ses hautes considérations sur Aristote, la classification, les objets, la covariance, C++, les styles Latex et la vie pratique (changer les pneus dans le parking du laboratoire, comment caler les armoires et monter les fauteuils, ne pas coller les documents contre les écrans, etc.). Je n'ai pas toujours suivi ses conseils, mais il en est toujours resté quelque chose ...

Merci à Michel Habib, un directeur de thèse qui ne s'oublie pas !

Bien sûr, je n'oublie pas toutes les personnes avec lesquelles j'ai pu collaborer étroitement ces derniers temps et qui ont contribué à mes réalisations :

- Yolande Ahronovitz, avec qui j'ai partagé tant de cours et qui tient le filet dans toutes les manœuvres administratives acrobatiques (sans exception),
- Gilles Ardourel, qui a maintenant un agenda (enfin je crois ...),
- Anne Berry, qui a relu et corrigé le chapitre 3 et donné de bons conseils sur la vie, en général ...
- Michel Dao, qui, depuis 6 ans, a l'air de trouver que ce que nous faisons est amusant et coordonne avec un grand flegme les activités de recherche des agités que nous sommes,
- Hervé Dicky, unique dans son genre,
- Christophe Dony, voyons comment on va programmer tout çà,
- Hervé Leblanc, le poète du clavier,
- Thérèse Libourel, dix ans de travaux ensemble et beaucoup d'autres choses en commun ...
- Cyril Roume, qui implémente plus vite que son ombre,
- Hernan Astudillo, Robert Godin, Petko Valtchev et Houari Sahraoui toujours prompts à l'organisation d'échanges ou de workshops,
- Olivier Gout, Luc Bernard-Gazagnes et Jean Villerd, dans la catégorie jeunes espoirs,

Chapitre 1

Introduction

Les classifications (hiérarchies d'héritage ou de spécialisation) constituent le cœur des systèmes à objets, dont elles structurent la connaissance et supportent une part des raisonnements et du calcul. Elles reflètent l'organisation des concepts du domaine représenté ou classifient les artéfacts logiciels utiles à l'expression d'un problème et de sa solution. Elles organisent des entités de différentes natures comme les classes, les interfaces ou les prototypes, véhiculent des sémantiques variées, notamment les notions de spécialisation/généralisation, de sous-typage et d'héritage de propriétés, enfin prennent des formes diverses telles que des arborescences, des graphes sans circuit ou des treillis. Leur définition tout comme leur utilisation sont à l'origine de nombreuses questions : choix de conception, traitement de l'évolution, codage en mémoire, résolution de l'héritage des propriétés, etc.

Le présent mémoire est centré sur le problème de l'aide à la construction de hiérarchies de classes dans une sémantique de spécialisation/généralisation. Les études qu'il rapporte constituent une pièce essentielle de mes travaux de recherche qui traitent également d'autres aspects des hiérarchies (résolution de conflits par linéarisation, décomposition de graphes), ainsi que des mécanismes de contrôle d'accès statique dans les langages de programmation à objets et de la modélisation des exceptions. Le croquis de la figure 1.1 présente la part temporelle respective des travaux.

Les besoins

Les ateliers de génie logiciel actuels proposent des solutions de plus en plus approfondies pour appuyer les différentes tâches que suppose le développement de logiciels de grande envergure : création assistée des modèles, soutien dans la production de modèles conformes à certains standards, contrôles de cohérence inter et intra modèles, définition de profils utilisateur, méta-modélisation, ré-ingénierie, utilisation de patrons de conception ou de programmation, évaluation de la qualité d'un projet.

L'élaboration et les besoins d'évolution de hiérarchies de grande taille nécessitent des outils spécifiques qui devraient trouver leur place dans les ateliers de développement évolués. Différentes activités peuvent ainsi être soutenues. La conception proprement dite passe par la recherche de classes à différents niveaux d'abstraction et par leur organisation. Cette recherche est souvent guidée par l'image que le concepteur se fait des concepts du domaine mais sous la restriction qu'apportent les besoins spécifiques d'une application. Il est alors utile de suggérer des abstractions plus fines, de dénoncer des redondances dans les définitions ou encore de faire apparaître des relations de spécialisation non identifiées. Des transformations sont souvent nécessaires entre le modèle d'analyse et son codage dans le langage d'implémentation. On peut imaginer proposer une assistance automatique à ces transformations, qui assure une approche systématique et rationnelle des transformations. L'évaluation du résultat peut être faite par le seul concepteur, mais de nombreux aspects peuvent lui échapper dès que la taille de la hiérarchie devient importante. A ce moment-là l'utilisation de métriques, traquant incohérences et redondances peut être d'un grand secours.

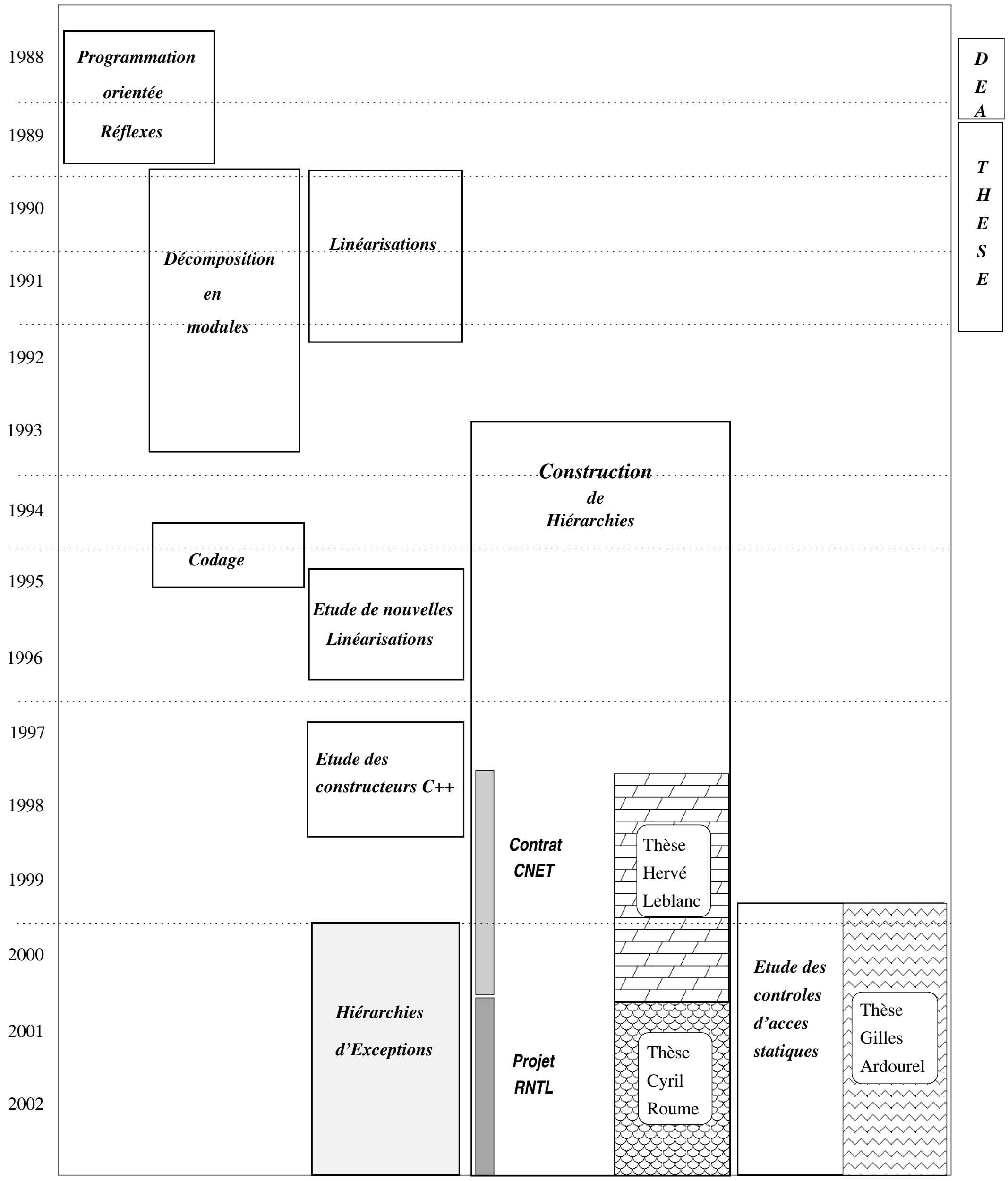


FIG. 1.1 – Croquis des travaux de recherche

Approche du problème

Nous considérons la relation entre les classes comme une relation de spécialisation/généralisation. Elle devrait ainsi reposer naturellement sur l'inclusion des extensions (ensembles d'éléments des classes), mais dans le cadre du génie logiciel, ces éléments ne sont pas exprimés dans les phases initiales de conception. Nous reste donc l'intension (ensemble des caractéristiques structurelles et comportementales d'une classe) pour officier.

Considérer les classes comme des ensembles de caractéristiques nous renvoie aisément à l'utilisation des treillis de Galois (autrement nommés treillis de concepts) pour la structuration, comme le montre bien l'article fondateur de R. Godin et H. Mili sur les relations entre hiérarchies de classes et treillis de Galois. Cette structure mathématique assure, une fois interprétée dans notre problématique, des propriétés fortes aux hiérarchies construites notamment l'absence de redondance de caractéristique, l'identité de l'inclusion entre intensions et de l'ordre entre classes, le respect de la spécialisation naturelle entre propriétés, enfin la minimalité du nombre de classes. La description des données de départ est cependant relativement délicate si l'on veut obtenir des résultats utilisables en pratique. Il y a donc deux aspects mêlés dans les travaux, d'une part la manipulation de la structure mathématique a suscité des études algorithmiques, d'autre part la mise en œuvre a nécessité la compréhension et la prise en compte de divers aspects pratiques des langages de modélisation et de programmation par objets. Un point doit être clair : jamais nous ne prétendrons donner de solution « miracle », la perspective suivie est plutôt de faire des propositions, basées sur une approche rigoureuse, à un concepteur qui seul décidera de leur emploi.

Plan du mémoire

L'objectif du manuscrit est de présenter synthétiquement et dans un cadre unifié un point de vue sur le problème de l'automatisation de la construction de hiérarchies de classes : comment le problème peut être défini, quelles propositions nous avons pu faire, comment elles se situent dans le domaine, et les perspectives de travail qui s'offrent actuellement. C'est dire que nous souhaitons développer ici plutôt une démarche que les résultats techniques qui se trouvent par ailleurs dans diverses publications.

Le chapitre 2 définit le problème comme la recherche d'une méthode automatique de construction de classification. Il précise en particulier le langage de description des données. Le cadre théorique qui permet de former une solution à notre problème est présenté au chapitre 3. Une synthèse des recherches effectuées présente, au chapitre 4, les contributions que nous avons pu faire sur le sujet. Nos travaux sont replacés dans le domaine lors d'une présentation succincte de celui-ci (chapitre 5). Nous tirons une brève conclusion et rassemblons quelques perspectives au chapitre 6.

Chapitre 2

Un problème de classification

Préambule

Les réflexions sur la classification ont été menées avec R. Ducournau, T. Libourel et A. Napoli. Nous scrutons d'un œil intéressé les hiérarchies d'exceptions de Java avec Y. Ahronovitz. Le premier méta-modèle général a été défini dans le cadre du projet ADC en collaboration avec N. Chevalier, M. Dao, C. Dony, H. Leblanc et T. Libourel. Il a été implémenté en Java par les soins de T. Brunel. Le deuxième est le fruit d'une étape du projet MACAO à laquelle ont participé M. Dao, T. Libourel et C. Roume. C. Roume en a transféré la substance dans Objecteering.

2.1 Introduction

Dans cette partie, nous revenons sur la notion de classification, comme l'une des sources d'inspiration des approches par objets en informatique. Nous inscrivons ainsi nos propositions dans le cadre de la construction automatique de classifications comme une forme de classification de classes (partie 2.2). Nous discutons de la nature de la classification de classes en génie logiciel dans la partie 2.3. L'ayant rapprochée d'un problème de regroupement conceptuel, nous décrivons successivement les critères qui peuvent présider à ce regroupement (partie 2.4), les choix faits en matière de langage de description (partie 2.5), la structure attendue (partie 2.6) et quelques éléments concernant les procédures de classification (partie 2.7). Nous concluons dans la partie 2.8.

2.2 La classification dans les approches à objets

La classification est un aspect essentiel de l'intelligence humaine qui a joué un rôle fondamental dans de nombreuses avancées scientifiques, notamment en botanique et en biologie. Cette capacité que nous avons de réunir des objets ou des idées dans des « classes » et d'organiser ces classes entre elles structure notre connaissance, facilite la mémorisation et fonde en partie compréhension et raisonnement inductif. La recherche d'une définition ou d'une description synthétique des classes produit de nouveaux concepts par découverte de caractéristiques communes et par abstration, ce qui constitue l'une des richesses intellectuelles que l'on peut tirer de la classification. Comme le présente la synthèse de [Bou96], les travaux sur la construction automatique de classifications ont été menés à l'origine dans les deux domaines principaux de l'analyse de données [Did71, Ben73] et de l'intelligence artificielle (apprentissage symbolique non supervisé) [MS83, Fis87, FP91]. Actuellement ils

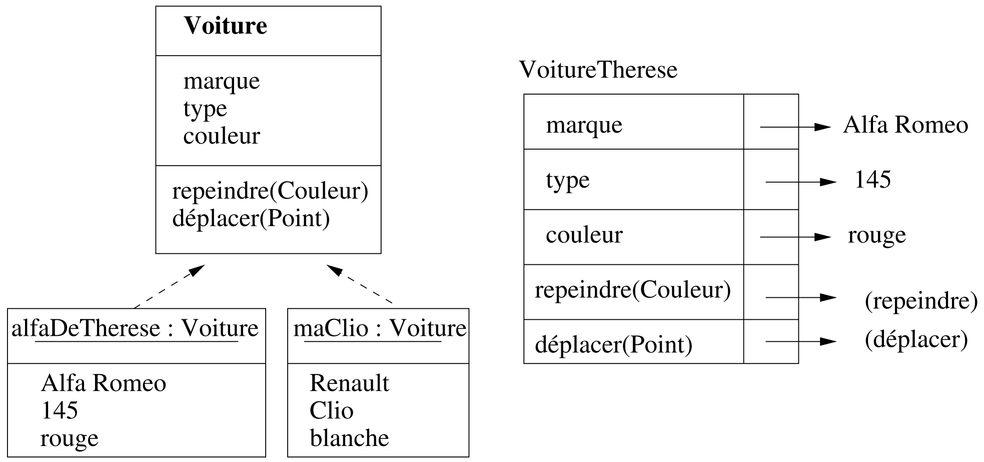


FIG. 2.1 – Classes vs. prototypes

sont également présents dans des domaines apparemment aussi différents que la fouille de données [SN99, PBTL99] ou la reconstruction phylogénétique [Ber97].

Le modèle objet On peut considérer les approches à objets comme l'introduction systématique de la classification dans les représentations informatiques et principalement dans les langages de programmation. La définition d'un langage de programmation à objets a été très tôt déclarée incomplète sans la composante apportée par l'héritage [Weg90]. Les approches à objets sont en effet basées sur l'idée que l'on « simule » le monde réel, entraînant la recherche d'une certaine correspondance entre les entités du monde réel et leur représentation informatique [Per98].

Cependant on ne peut discuter de classification dans les objets sans noter que deux modèles essentiels, issus des réflexions menées en sciences cognitives s'y disputent le terrain (même si l'un est majoritaire) :

- le modèle à classes est basé sur l'idée qu'un concept se décrit à l'aide d'une extension (ensemble d'exemples concrets) et d'une intension (ensemble de caractéristiques); les objets sont considérés selon deux statuts, instance ou classe, les modèles les plus élaborés systématisant ce principe en proposant des méta-classes; Figure 2.1 (à gauche) est donné un exemple de classe, `Voiture`, avec son intension constituée de ses attributs (`marque, type, couleur`) et des opérations que l'on peut lui appliquer (`repeindre, déplacer`). Deux instances sont aussi montrées, `alfaDeTherese` et `maClio`, qui typiquement ne contiennent qu'une valuation des attributs. La relation de spécialisation (ou d'héritage) permet de dériver une classe d'une autre, tandis que la relation d'instanciation relie une instance à une classe.
- le modèle à prototypes [DMB98, NTM99] est basé sur l'idée qu'un concept se décrit à l'aide d'exemples typiques (Figure 2.1, la voiture de Thérèse est une voiture typique), ce qui correspondrait mieux au raisonnement humain [Lie86]; classer un individu consisterait à le rapprocher plus ou moins d'un prototype [Bou96]. Les prototypes sont obtenus par création de toute pièce (comme montré sur la figure), copie (clonage), ou copie différentielle (extension). Ce modèle qui semble plus simple en première approximation se complique dans ses implémentations, où cohabitent finalement deux sortes de prototypes (concrets ou représentants moyens) et où un objet du monde représenté peut se trouver décrit par plusieurs prototypes (problème d'identité).

Nous développerons principalement dans ce mémoire le modèle majoritaire en génie

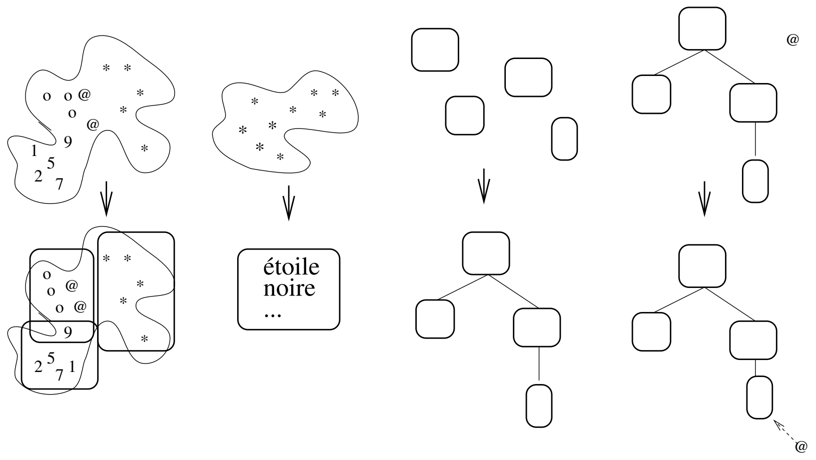


FIG. 2.2 – Schémas de classification

logiciel, qui est le modèle à classes, mais ferons quelques allusions ici ou là au modèle à prototypes où des propositions de construction automatique, proches de celles faites pour les classes, ont été avancées [Moo95, Bar00].

Enfin, le modèle objet « pur », majoritaire au sein des langages de programmation et qui est principalement basé sur des entités décrites par des propriétés, ne met pas en valeur la notion de relation. A l'inverse, les formalismes d'analyse/conception dédiés au génie logiciel à objets dont UML [BRJ99] constitue une synthèse, s'appuient sur une représentation où les relations ont une place essentielle. Dans nos travaux, nous avons beaucoup travaillé sur le formalisme objet pur, que nous étudierons principalement. Nous nous sommes récemment orientés vers la classification en présence de relations (sur la base des diagrammes UML) en proposant des techniques qui étendent sans les contredire les techniques utilisées pour traiter le formalisme objet pur (section 4.7).

Aspects de la classification Nous retiendrons ici que la classification est principalement le processus et le résultat [DHLN99] :

- de la découverte de classes, par deux procédés
  - le premier est la division en classes d'un ensemble d'entités, qui produira des « classes décrites en extension »,
  - le deuxième est la description d'un ensemble d'entités par des propriétés caractéristiques, menant à des « classes décrites en intension »,
- de l'organisation des classes les unes par rapport aux autres, ce que l'on peut appeler « classification de classes »,
- du placement d'une entité relativement aux classes existantes, ce que l'on peut appeler « classification d'instances ».

Les schémas de la figure 2.2 illustrent ces différents aspects (calcul de classes extensionnelles, recherche de classes intensionnelles, organisation de classes et rangement d'instance). L'analyse de données s'intéresse plutôt à la découverte de classes extensionnelles, tandis que les approches à objets, ou l'apprentissage (entre autres, le regroupement conceptuel), recherchent majoritairement des classes intensionnelles.

Une autre distinction importante repose sur le « moment » de la classification :

- classification *a posteriori*. La classification de l'analyse de données [DLPT82] ou la classification conceptuelle en apprentissage [GLF89, Lan96] proposent des classes après analyse d'un ensemble d'individus (ou exemples). Le résultat permet ensuite de classer de nouveaux individus (appliquant ce que nous avons appelé la classification d'instances).

– classification *a priori*. Les approches à objets privilégient une création des classes par un concepteur à partir d'une analyse experte d'un domaine. Le propre des pratiques traditionnelles en génie logiciel et bases de données est de partir d'une spécification textuelle informelle dont on extrait par itérations successives une description et une organisation des classes dans un formalisme graphique tel qu'OMT [RBP+91] ou UML [Rat99] ou dans un langage de programmation. Les individus (ou instances) sont créés ensuite, en respectant le schéma imposé par la classe. L'instance porte relativement peu d'informations, souvent seulement un ensemble de valeurs pour les attributs définis par la classe. En représentation des connaissances, dans les systèmes à prototypes ou plus récemment dans le domaine des données semi-structurées, cette approche connaît une variante dans laquelle les individus peuvent être créés indépendamment de classes, puis (dans les systèmes à classes) classés ensuite [Neb90, MRU90, BGMM99, CG00, Nap00].

Les approches à objets déclinent à peu près toutes les formes de classification. S'il n'y a pas à proprement parler de construction de *classes extensionnelles*, ce qui n'est pas dans la logique des choses, puisque les systèmes à objets cherchent à utiliser l'intension des objets, les instances sont cependant stockées et on a quelquefois un moyen de mémoriser le « bouquet » d'instances d'une classe et d'y faire référence, mais pas toujours. Le système à prototypes KEVO par exemple, conserve en mémoire des familles de prototypes clonés les uns à partir des autres [Tai93].

Plus généralement sont construites des *classes intensionnelles*. Dans le cas des systèmes à prototypes, ce sont des représentants moyens, ou « traits » en SELF [US87], qui servent par exemple de réservoir de propriétés. Dans le cas des systèmes à classes, on construit (automatiquement ou non) des classes intensionnelles.

La *classification de classes* se fait au travers d'une relation que l'on appelle suivant le contexte « généralisation/spécialisation » (en analyse/conception) ou « héritage » (en programmation et dans ce dernier cas, nous verrons qu'elle peut recouvrir plusieurs sémantiques (spécialisation, partage d'implémentation, sous-typage). Dans le cas des prototypes, les relations « parent » entre traits peuvent être considérées comme une forme de classification.

En ce qui concerne la *classification d'instances*, on en trouve une forme excessivement dégradée en bases de données et en programmation, qui consiste à créer les instances à partir des classes (instanciation), c'est donc le programmeur qui prend la décision de représenter un objet du domaine par une instance de telle ou telle classe et le choix qu'il effectue peut ne pas être le meilleur. Il peut créer une représentation de la voiture de Thérèse comme instance d'une classe *Véhicule* alors qu'il existe une classe *Voiture* plus appropriée. Plus intéressante est la classification automatique ou dirigée d'instance qui se pratique en représentation de connaissances par objets, et qui sert de base à des raisonnements. La classe est dans le cas de la classification d'instances *descriptive* si on suppose que ses propriétés sont des conditions nécessaires à l'appartenance de l'instance à la classe, ou *définitionnelle*, dans ce cas ce sont des propriétés nécessaires et suffisantes. Les langages de programmation imposent généralement une contrainte sur le rattachement des instances aux classes : une instance n'a qu'une classe « de rattachement » (l'ensemble des classes auxquelles l'instance appartient admet un unique plus petit élément). On parle dans ce cas de mono-instanciation. En représentation de connaissances par objets, la multi-instanciation est plus communément admise.

Un aperçu sur les objectifs Quatre objectifs majeurs s'imposent pour la classification dans le cadre des objets.

- L'objectif de la phase d'analyse est essentiellement de produire *un modèle de connaissances* du domaine. Ce qui différencie, en génie logiciel, l'approche objet de l'approche précédente qui était profondément fonctionnelle, c'est que l'on met justement l'accent sur des types abstraits de données sur lesquels se construira le logiciel plutôt que seulement sur les fonctionnalités demandées au logiciel à un instant donné. Il est reconnu que les données du domaine, vues d'une manière relativement abstraite sont certainement plus « stables ». Par exemple, si on souhaite construire un logiciel destiné à l'apprentissage des mathématiques, la notion de polygone est certainement plus durable dans la vie de ce logiciel que le fait d'avoir une interface graphique de telle ou telle sorte, ou de savoir si les leçons sont organisées de telle ou telle manière.
- La classification produite est codée dans un programme (aspect production de logiciel), qui s'exécute (aspect opérationnel). En particulier la classification — en l'occurrence la hiérarchie d'héritage — guide le mécanisme d'appel de méthode.
- En représentation de connaissances par objets, la classification d'instance permet une forme de raisonnement : décréter l'appartenance de l'instance à une classe permet de déduire des caractéristiques applicables à l'instance (par exemple des valeurs d'attributs).
- Enfin, la classification sert à stocker de manière organisée des constituants logiciels, et facilite leur identification et leur réutilisation.

Place de la classification de classes en génie logiciel Nous avons décrit les points essentiels de la classification dans le domaine des objets. Dans le contexte du génie logiciel sur lequel nous nous focaliserons à présent, l'éclairage apporté par la classification dans diverses tâches courantes du cycle de vie est important et met en évidence nombre d'outils dont pourrait bénéficier le développement. Il est ainsi intéressant de mettre en perspectives les grandes lignes d'un développement standard et le rôle qu'y joue la classification.

Les représentations des entités de base varient au cours du cycle de vie du logiciel (Figure 2.3). Les langages de représentation sont supposés assurer que les expressions du langage permettent de décrire le domaine de manière cohérente. On peut exprimer cette cohérence au moyen d'une sémantique naturelle [DHLN99]. Les phases d'analyse et de conception sont marquées par l'utilisation de textes et de formalismes graphiques traduisant à la fois les différents concepts en relation et la dynamique du système. On dessine donc des diagrammes que l'on munit d'une sémantique souvent partielle, exprimée par exemple dans un langage tel qu'OCL (Object Constraint Language) [Rat97]. Ces diagrammes servent en phase d'analyse à discuter avec les clients ou les experts du domaine, et à établir le modèle du domaine et de l'application que l'on réalise. En phase de conception, ils servent plutôt, entre informaticiens, à discuter et à expliciter les choix d'implémentation. En phase d'implémentation ou de réalisation, les entités sont des morceaux de programme, des parties du schéma de la base de données ou de connaissances. La structure du langage de programmation est modulaire et respecte relativement la structure des classes. En phase d'exploitaton, certaines entités se sont parfois transformées en structures de données (les classes) codées par des octets (dans les fichiers exécutables en génie logiciel), et on manipule plutôt des instances (dans les langages peu ou pas réflexifs), qui collaborent par des échanges de messages ou sont happées par un mécanisme de filtrage. Dans les phases d'évolution ou de rétro-conception, toutes les constructions peuvent être remises en question.

On peut replacer les différentes formes de classification dans ce contexte de développement (figure 2.4). Seuls les traits les plus marquants de chaque phase sont présentés.

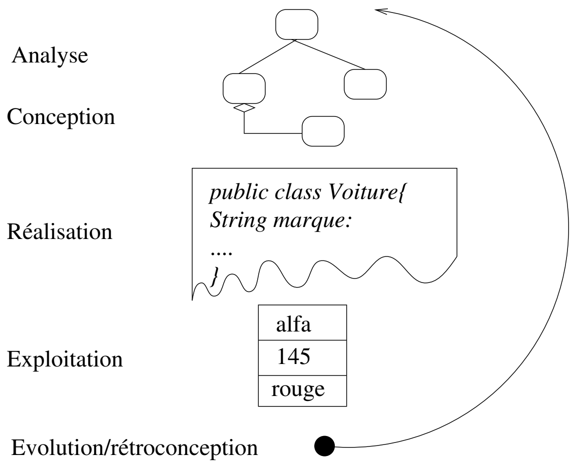


FIG. 2.3 – Cycle de vie

L'analyse, comme on l'a vu, se préoccupe surtout de construire un modèle du domaine utile pour la résolution d'un problème, donc se focalise sur la construction de classes intensionnelles et leur organisation. La conception précise cette première classification, puis, si un formalisme à base de classes et d'associations a été utilisé pour l'analyse, se doit d'expliciter la transition vers le modèle objet, généralement sans associations, d'un langage de programmation cible. On trouvera à cette étape quelques problèmes de transformation de classification intéressants, tels que le passage d'une structure de classification non contrainte (par ex. héritage multiple) à une structure plus contrainte (par ex. héritage simple si le langage est Smalltalk ou Java). On trouvera quelques directions pour traiter ce dernier problème dans [RBP+91] et quelques approches automatisables dans [CMR02, Rou02b]. L'implémentation des aspects structurels (excluant l'écriture des corps de méthodes) doit être relativement automatique si la conception a été faite de manière très précise. A l'exploitation, les instances sont créées et utilisées à diverses fins, notamment le stockage des données à traiter et des résultats. Les instances peuvent évoluer et avoir à migrer ce qui pose des problèmes annexes dus à la rigidité du modèle. Lorsque le logiciel évolue légèrement par exemple, la classification des classes est modifiée d'une manière que l'on peut qualifier de locale ou d'incrémentale et cette perturbation peut se répercuter sur les instances. Enfin, on s'attelle parfois à des opérations de rétro-conception qui remettent en cause fondamentalement toutes les classifications effectuées. En génie logiciel, cela peut se produire lorsque l'on change de langage par exemple et que les contraintes sont différentes (le même exemple peut être donné de transformation due au passage de l'héritage multiple à l'héritage simple).

On voit ainsi que la classification des classes apparaît comme l'une des opérations de classification clef dans le développement logiciel, et mérite ainsi une étude approfondie. Nous nous intéressons ici à la construction automatique de cette classification, mais il est d'autres enjeux importants tels que la navigation (recherche de la classe désirée) que nous ne traiterons pas ici.

2.3 Discussion sur la nature de la classification de classes

Une métaphore biologique Une métaphore développée dans [Ast97] rapproche la problématique de la construction de hiérarchies de classes de la constitution de classifications phylogénétiques, qui s'opposent, en biologie, aux classifications phénétiques.

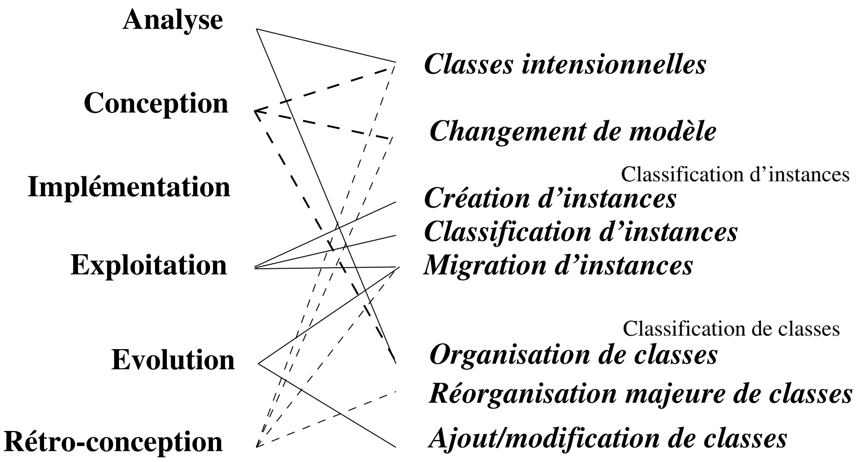


FIG. 2.4 – Classification et cycle de vie

Dans la classification phénétique, les individus sont décrits par leur phénotype et regroupés en classes d'après les similarités de ces mêmes phénotypes. La classification est basée ainsi sur l'apparence et destinée plutôt à l'identification des individus. Ce type d'approche est à l'origine des premières classifications du vivant. Appliquée au génie logiciel, cette métaphore serait, selon l'auteur, particulièrement adaptée lorsque l'enjeu est plutôt de *retrouver* un artefact logiciel, donc elle est bienvenue pour la constitution de hiérarchies d'interfaces.

La classification phylogénétique propose quant à elle une hypothèse sur la manière dont les espèces descendent les unes des autres, elle cherche à montrer les organismes comme le résultat d'un processus. La parcimonie, critère fondamental dans la construction de l'arbre phylogénétique, s'exprimerait en génie logiciel comme la réduction globale (sur l'ensemble de la hiérarchie) du nombre de modifications de caractéristiques dans le passage d'une classe à une autre. Le champ d'application de cette métaphore serait le codage des classes (implémentation) pour *réduire le développement* et l'auteur la conseille dans ce cadre.

On peut discuter ce point de vue. Tout d'abord, si le résultat dans le cadre de la biologie est une hypothèse sur l'évolution des espèces, la transposition en génie logiciel serait la production d'une théorie de l'évolution des concepts (voire simplement de l'évolution de leur représentation) dans un domaine d'application. Or dans une hypothétique modélisation d'un système bancaire et dans son développement ultérieur, est-il plus utile de connaître l'histoire de l'évolution des différents types de comptes bancaires (métaphore phylogénétique) ou d'en avoir une organisation rationnelle plus appropriée à leur identification et à leur utilisation (métaphore phénétique) ? Même dans le cadre de l'implémentation, les développeurs doivent avoir une idée claire et organisée des classes que ce soit pour les modifier ou pour les étendre par de nouvelles classes [Tor02].

Un deuxième problème est posé par les langages de programmation actuellement les plus utilisés. Les liens de spécialisation (dans les langages de modélisation) ou d'héritage (dans les langages de programmation) ne supportent pas bien la sémantique de la descendance généalogique entre espèces. Si l'on considère l'extension d'une classe, les langages présument que les instances d'une classe sont aussi instances de ses super-classes, ce qui n'est pas le cas dans une interprétation d'un lien généalogique telle que propose de l'utiliser H. Astudillo[1],

[1] Ce qui est déjà un écart par rapport à la phylogénie, car dans un arbre phylogénétique, les sommets représentent des groupes incluant une espèce ancêtre et ses descendants ; l'arbre représente plutôt qui est proche de qui et non pas qui descend de qui [Leb98].

la classe des oiseaux descend probablement de la classe des archeopterix mais un oiseau n'appartient pas à la classe des archeopterix (qui est éteinte). Pour ce qui est des intensions, les langages n'acceptent pas souvent la disparition d'un attribut (un attribut déclaré dans une classe apparaît dans la structure mémoire des instances de cette classe et de ses sous-classes, il ne peut être enlevé). Il faut prévoir un codage *ad hoc* pour ce type d'attribut. Une méthode qui est héritée mais n'est plus supportée demandera dans la plupart des langages un traitement peu satisfaisant par la levée d'une exception particulière.

Classification automatique de classes et regroupement conceptuel Nous pensons donc que, pour rester dans une métaphore biologique, la classification des classes la plus adaptée au génie logiciel est de nature phénétique. Dans ce même contexte, nous comprenons la classification comme un problème de regroupement conceptuel, ou encore un apprentissage symbolique qui cherche à former des concepts décrits en intension à partir de l'observation d'un domaine, dans le but de disposer d'une représentation efficace [Bou96].

Cette classification conceptuelle présente quelques spécificités. Contrairement aux approches usuelles en classification conceptuelle, ici, les entités qui servent de point de départ sont déjà des *classes*, éventuellement des classes *déguisées* en instances comme dans [LBSL90]. Mais comme nous l'avons vu les « vraies » instances sont, en génie logiciel, fort maigres en information. Ces premières classes sont un substrat dont on va faire émerger de nouveaux concepts, ces derniers étant eux-même réinterprétés comme les classes de la classification résultat. Un autre point est que l'idée de *similarité* est gommée au profit d'une nécessité d'*exactitude* de la représentation : deux instances d'une même classe disposent exactement des propriétés de la classe, seule la valuation, contrainte par le typage, peut varier. Enfin, les données se présentent sous une telle forme que l'on va plutôt rechercher des propriétés communes que raisonner sur des valeurs partagées [LBSL91].

Il est important de préciser que, contrairement à la proposition faite dans [Cap93], la classification de classes qui nous intéresse n'est pas un simple placement relatif des classes les unes par rapport aux autres : nous cherchons à mettre en évidence de nouvelles classes qui vont correspondre à de nouvelles abstractions pertinentes.

Pour reprendre la présentation synthétique proposée dans [Bou96], voici les éléments à déterminer dans la définition d'une méthode de construction automatique de classification :
- *critères* quel est l'objectif de la classification ? Comment se définit-elle ?
- *langage de description* objets et concepts (qui sont donc ici des classes) sont-ils représentés par des couples attribut/valeur, par des graphes, etc.
- *choix d'une structure* arbre, pyramide, ordre partiel, etc.
- *procédures de construction* algorithmique mise en œuvre

Nous reprenons ces divers élements dans les sections suivantes.

2.4 Critères pour la classification de classes

Objectifs généraux du génie logiciel Clairement, les objectifs affichés par le génie logiciel ne sont ni le raisonnement ni la prédiction. Ils se concentrent sur la production de logiciels avec des exigences de :
- fiabilité,
- performance,
- coût de développement (d'où l'importance de la réutilisabilité),
- maintenance (corrections et extensions).

Mais ces objectifs sont très généraux et il importe de déterminer plus concrètement quelles caractéristiques la classification des classes choisie doit posséder pour les satisfaire.

Choix d'un point de vue général sur les hiérarchies de classes Trois points de vue majeurs et contradictoires s'opposent lorsque l'on considère des hiérarchies de classes. Notre première préoccupation doit donc être de déterminer lequel est le plus favorable pour répondre à nos objectifs généraux.

– *Economie de développement.* La hiérarchie d'héritage est un réservoir d'information. Les liens d'héritage sont utilisés essentiellement pour réduire la quantité de structure ou de comportement à décrire. L'exemple typique est le sous-classement de `Vecteur` par `Pile`, par exemple réalisé dans l'API Java [AG98] : ces deux classes possèdent la même structure et toutes les opérations admises pour une pile existent déjà pour une liste. Un inconvénient de ce procédé est que les opérations applicables sur les listes mais non destinées aux piles leur sont malgré tout applicables. Certains langages proposent des astuces pour limiter cette application telles que le masquage du lien d'héritage grâce à la directive `private` en C++, le `reject` en Eiffel ou encore le signalement d'exception.
– *Sous-typage ou substituabilité.* Le terme de « sous-typage » étant employé de diverses manières, nous lui avons adjoint ici celui de substituabilité pour réduire l'ambiguïté. Dans cette vision, un type T_2 est sous-type d'un type T_1 si une instance de T_2 est toujours substituable à une instance de T_1 sans provoquer d'erreurs de types. Cela implique en particulier des contraintes sur les méthodes de T_1 redéfinies dans T_2, en l'occurrence la covariance du type de retour et la contravariance des arguments. C'est parfait pour les piles et les vecteurs pour lesquelles il n'y a pas de relation de sous-typage. Mais une conséquence est que l'on ne peut plus représenter `Carnivore` (supposée posséder une méthode `mange(AlimentCarné)`) comme une sous-classe de `Animal` (supposée posséder une méthode `mange(Aliment)`), avec `AlimentCarné` sous-classe de `Aliment`. De manière générale, ce modèle est parfait pour les types et la réduction d'erreurs lors de l'exécution des logiciels, mais trop restrictif dans la modélisation de certains domaines d'applications [Duc02].
– *Spécialisation.* La hiérarchie est implicitement une classification par spécialisation des concepts du domaine d'application ou des artefacts logiciels utilisés. Cette spécialisation se définit par le fait que les objets couverts par un concept sont couverts par les concepts plus généraux (inclusion des extensions). La description d'un concept est spécialisée dans les concepts plus spécifiques par exemple par raffinement des caractéristiques déjà existantes ou ajout de nouvelles caractéristiques (inclusion ou subsomption des intensions).

Malgré leur contradiction ces différents points de vue sont souvent mêlés dans le développement. D'une part ils ne sont pas toujours conflictuels, par exemple spécialisation et sous-typage impliquent souvent un partage de structure et de comportement tandis que certaines formes de spécialisation respectent le sous-typage. D'autre part ce mélange des points de vue dans l'esprit des développeurs est provoqué en partie par l'approche objet qui promeut l'idée d'un *continuum* entre les différentes phases de developpement, idée soutenue par la proximité des concepts dans les langages utilisés dans les différentes phases. Ainsi dans l'utilisation de langages de modélisation tels qu'UML, tout comme dans celui des ateliers de développement, on ne distingue pas toujours bien ce qui relève essentiellement de l'implémentation et ce qui est de l'ordre de l'analyse. Or un langage de programmation impose toujours des contraintes sur les modèles échafaudés lors d'une analyse. Ainsi, si lors d'une modélisation, une méthode en spécialise une autre en ajoutant une nouvelle exception, l'implémentation avec le langage Java, qui respecte la règle de substituabilité en ce qui concerne la déclaration d'exceptions, imposera une stratégie qui réconcilie sous-typage et spécialisation. Par exemple la nouvelle exception deviendra une sous-classe d'une exception plus générale qui sera introduite dans la signature de la méthode spécialisée [AH01]. En se forçant à respecter le sous-typage on aura dans ce cas amélioré également la spécialisation dans le modèle.

Spécialisation et sous-typage sont sans doute ce qui amène les classifications les plus intéressantes du point de vue de la compréhension [Tor02] et il nous semble que produire un code compréhensible est à la base de la majeure partie des qualités attendues d'un logiciel. Mais, comme il a été évoqué ci-dessus, le sous-typage est trop strict et souvent antagoniste avec la modélisation « naturelle » que demande la phase d'analyse. Reste la spécialisation, qui sera donc ici notre idée directrice.

Critères internes attendus De nombreux critères sont cités dans des documents traitant de *refactoring* [OJ93, FBB+99], de métrologie [CK94, LK94] ou donnant des conseils généraux de construction [Cas95].

Ainsi on peut dégager ce qui concerne :
- la qualité de la modularité (taille et cohésion des classes, réduction du couplage),
- la qualité de l'encapsulation (masquage de l'implémentation),
- le polymorphisme (élimination des vérifications de type, stratégies de nommage),
- la qualité des liens de classification, qu'il faut distinguer clairement des autres associations (aggrégation, composition, etc.),
- le degré d'abstraction atteint (factorisation).

Nous ne courrons pas après tous ces objectifs, mais nous pensons que le point primordial est d'assurer une amélioration de la factorisation et des relations de spécialisation/généralisation dans la classification produite. Nous préciserons ces critères en décrivant le type de structure de classification que nous recherchons.

2.5 Un langage de description

Diversité des langages sources On peut définir une hiérarchie de classes comme un ensemble de classes muni d'un ordre partiel (de spécialisation), et une fonction qui associe à chaque classe l'ensemble des propriétés (attributs et méthodes) qu'elle déclare. Mais autour de cette vue simpliste se déclinent de nombreuses variations, même si l'on se restreint aux modèles standards, c'est-à-dire écartant des modèles tels que les mixins [BC90], des points de vue [CG90] ou des traits [UCCH91]. Nous ne tenterons pas de faire un état exhaustif des différences mais présenterons quelques faits marquants.

L'ordre partiel peut être réduit à une arborescence (héritage simple comme Java ou Smalltalk), ou non contraint (héritage multiple comme en C++, en Eiffel ou pour les interfaces Java).

Les propriétés ne sont pas indépendantes les unes des autres puisqu'une méthode par exemple peut être une redéfinition d'une autre et doit pour cela respecter un certain nombre de règles qui peuvent varier suivant le langage (invariance des types des paramètres et de retour en Java, covariance en Eiffel, invariance des types des paramètres en C++).

La sémantique opérationnelle associée à cette structure, qui décrit en particulier pour une instance donnée quel est son ensemble d'attributs ou comment se résout un envoi de message n'est pas mieux fixée. Dans le cas de l'héritage multiple, les travaux sur la résolution des conflits [BCK89, CG90, Bak91, DHHM94, BCH+96] n'ont pas fait émergé de modèle fort. Il est assez notable de voir dans un langage comme C++ la cohabitation d'une résolution des conflits explicite pour les méthodes et les attributs et d'une technique de linéarisation pour la résolution de l'appel des constructeurs [Huc00].

Diverses expériences et tâtonnements dans des champs d'étude spécifiques dont Smalltalk, C++ et Java [DDHL96, DHL97a, HL00] et le désir de travailler à un niveau plus conceptuel nous ont convaincus de l'intérêt de proposer un modèle général pour les hiérarchies de classes et de s'en servir comme base pour l'automatisation, quitte à étudier l'ajustement *a posteriori* aux singularités d'un langage ou d'un autre. Autant dire qu'UML, au moins tel qu'il est défini à ce jour, ne nous a pas apporté les clefs escomptées, car il est finalement trop calqué sur C++ et Java et ne dispose que d'une sémantique partielle.

Proposer un méta-modèle Un premier méta-modèle a été proposé dans [Leb00], mais il a dû être par la suite repensé et amendé dans [DHLR01b]. L'idée directrice de nos travaux est de décrire les classes par des ensembles de propriétés, puis de considérer cette description pour produire une meilleure classification. Ceci explique l'importance que nous attachons à la définition et à la description des propriétés. Cette vision s'oppose à des approches telles que [LA02, ST02], qui, dans un contexte différent, considèrent que la classe est essentiellement décrite par son nom. Or nommer c'est déjà classer comme le dit bien [Jac70]. Notre approche montre ainsi ses limites pour traiter certaines hiérarchies où le nom supporte l'essentiel de l'information, comme les hiérarchies de classes exception du langage Java où une étude linguistique du nom semble plus appropriée. Dans de telles classes en effet, les propriétés sont essentiellement un ou deux constructeurs et une méthode retournant un message décrivant l'erreur, ce qui est peu pour raisonner. Par contre le nom de la classe exception est formé assez clairement de plusieurs mots révélant sa sémantique : quoi de plus clair en effet que `ArrayIndexOutOfBoundsException` ou `ClassCastException` (toutes les exceptions ne sont pas aussi aisées à comprendre cependant). Une manière de réconcilier les deux serait peut-être de considérer le nom comme une propriété, mais cela interdit de réunir deux classes ayant même ensemble de propriétés et des noms différents, lorsque ces noms ne peuvent être considérés comme des synonymes.

Notion de co-référent et de référent Mais revenons aux propriétés. Le seul niveau syntaxique est trompeur et peu intéressant car nous avons besoin de considérer que deux portions de code de syntaxe différente font référence à une même notion. Par exemple, les deux déclarations d'attributs suivantes dans leurs classes respectives C_1 et C_2, quoique syntaxiquement différentes, représentent une même caractéristique pour les instances de C_1 et C_2, qui est de disposer d'un nom.

Exemple 2.1 (Objets qui ont un nom)
class C_1*{... nom :String ...}*
class C_2*{... Name :char[] ...}*

Inversement, des entités syntaxiques identiques peuvent fort bien référencer des notions différentes.

Exemple 2.2 (Objets qui ont un nom)
class C_1 *{... m(){code1} ...}*
class C_{11} *extends* C_1 *{... m(){super.m() ; i++} ...}*
class C_2*{... m(){code2} ...}*
class C_{22} *extends* C_2 *{... m(){super.m() ; i++} ...}*

Si les codes *code*1 et *code*2 sont foncièrement différents, il faut pouvoir considérer que les deux redéfinitions effectuées respectivement dans C_{11} et C_{22}, quoique de syntaxe identique, ne correspondent pas à un comportement similaire pour les instances des classes en question.

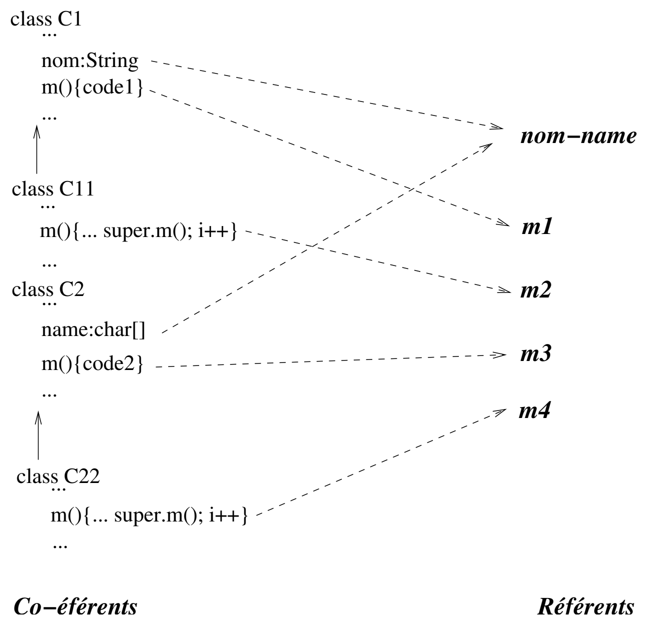


FIG. 2.5 – Co-référents et référents

Nous avons donc besoin de différencier d'une part ce qui est écrit syntaxiquement dans une classe et d'autre part la caractéristique abstraite qui a été ainsi codée (disposer d'un nom, avoir tel comportement, etc). Pour cela nous introduisons le modèle suivant. Ce qui est écrit syntaxiquement dans le code d'une classe ou dans un diagramme UML est appelé *co-référent* tandis que la caractéristique abstraite est appelée *référent*. La figure 2.5 présente ces deux niveaux pour l'exemple ci-dessus. Les co-référents sont faciles à nommer par leur forme syntaxique. Les référents doivent être nommés eux aussi, et le mieux est sans doute de calculer un symbole les représentant.

Notion de propriété générique Une autre dimension à prendre en compte est l'existence de relations entre ce que nous appelerons maintenant les co-référents, comme par exemple la relation qui relie deux méthodes dont l'une redéfinit l'autre ou encore dont l'une est abstraite et l'autre concrète. Ces relations ont une incidence sur les référents. Nous faisons l'hypothèse, qui est une thèse assez forte, que dans le cadre de hiérarchies de spécialisation, de telles relations peuvent s'abstraire en définissant entre les référents des relations de spécialisation. Nous travaillons donc dans un système où, lorsqu'une hiérarchie est bien formée, deux sortes de spécialisation se répondent : une spécialisation des classes et une spécialisation des référents.

Nous mettons en place cette nouvelle notion de la manière suivante. Les référents peuvent être regroupés d'après leur sémantique en ensembles que nous appelerons des *propriétés génériques*. Par exemple, nous imaginons de regrouper tous les référents qui ont vocation d'affichage, ou de sérialisation, ou tous les référents qui ont vocation d'associer un nom. Attributs et méthodes pourraient être mélangés dans cette opération de regroupement. Pour finir, chaque propriété générique est elle-même munie d'un ordre de spécialisation. La figure 2.6 présente un exemple de co-référents, référents et propriétés génériques qui étend l'exemple précédent. Un nouveau référent d'attribut (`nom-name-init-"zz"`) spécialise le référent déjà connu `nom-name` par l'ajout d'une valeur initiale. Ces deux référents sont regroupés dans la propriété générique `nommage`. Les référents correspondant à la méthode `m` sont également regroupés au sein d'une propriété générique nommée `m` à laquelle

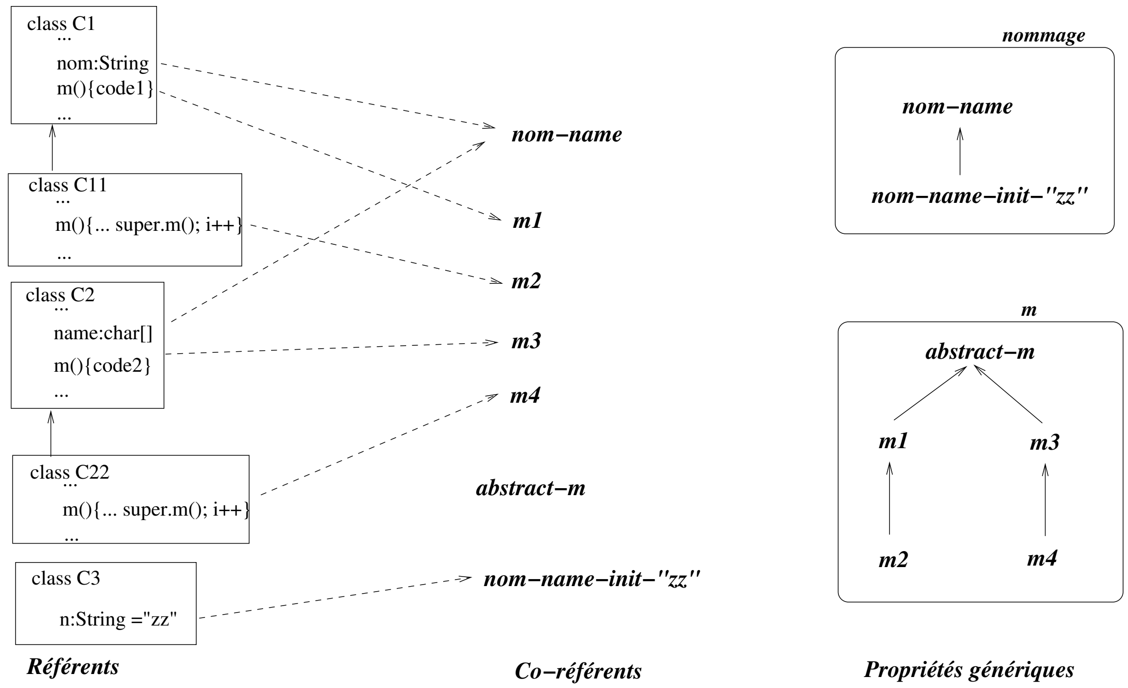


FIG. 2.6 – Propriétés génériques

on a ajouté un référent représentant les formes abstraites de cette méthode. Il n'y a pas de co-référent associé dans la hiérarchie actuelle mais son existence suggère une généralisation possible des classes existantes.

Intension des classes Dans ce modèle, une classe est décrite par une *intension* formée d'un ensemble de référents correspondant aux co-référents qu'elle déclare et dont elle hérite. Le méta-modèle ne met sur cet ensemble (cette intension) qu'une contrainte, que nous spécifions dans la propriété ci-dessous.

PROPRIÉTÉ 2.1 (COMPLÉTUDE DES INTENSIONS) *Lorsqu'un référent apparaît dans l'intension d'une classe, toutes ses généralisations (pour la propriété générique associée) y apparaissent aussi.*

Nous pouvons supposer que les intensions des classes de l'exemple seraient raisonnablement comme ci-dessous.

EXEMPLE 2.3 (INTENSIONS)
Intension(C1) ={nom-name,m1,abstract-m}
Intension(C11)={nom-name,m2,m1,abstract-m}
Intension(C2) ={nom-name,m3,abstract-m}
Intension(C22)={nom-name,m3,m4,abstract-m}
Intension(C3) ={nom-name-init-"zz",nom-name}

Méta-modèle proposé Cette modélisation peut se traduire mathématiquement, mais gagne en clarté à être présentée dans le formalisme UML (voir Fig. 2.7). Pour le décrire nous reprenons les termes de [DHLR01b]. Ce méta-modèle complète celui d'UML v1.3 [Rat99] sans le contredire ; les nouveautés apparaissent dans la partie hachurée. Nous avons introduit la notion de référent, qui représente une caractéristique que plusieurs classes

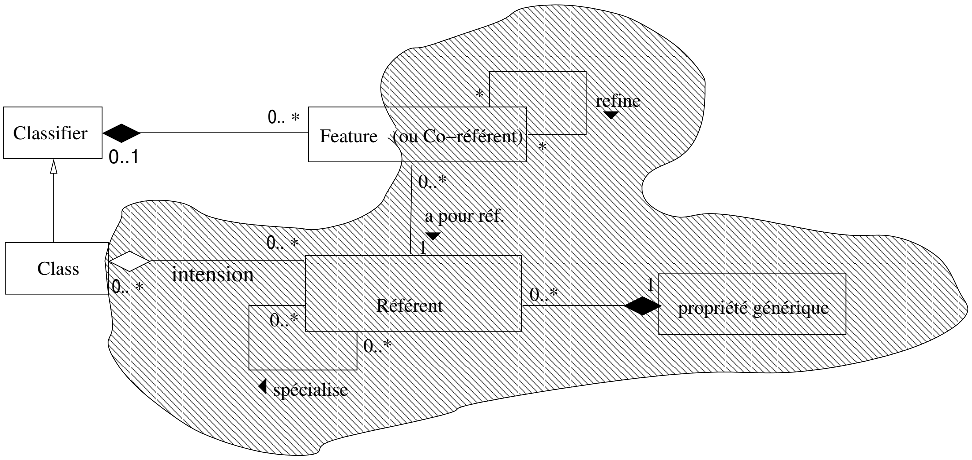


Fig. 2.7 – Méta-modèle

peuvent partager par l'intermédiaire de « features » (dans le vocabulaire UML) ou « co-référents » (dans celui qui est le nôtre à présent). Toute *feature* a exactement un référent, tandis qu'un référent peut être référent de plusieurs features, comme nous l'avons vu dans l'exemple. A priori, un référent peut exister sans co-référent : c'est le cas pour *abstract-m* vis-à-vis de la hiérarchie initiale. Sur l'élément $Feature$ (dont le nom est complété par *Co-référent*), une association $refine$ est ajoutée. Cette association, qui représente le fait qu'un co-référent peut en spécialiser une autre, peut correspondre par exemple à la redéfinition dans les langages de programmation ; en modélisation, il semble qu'elle devrait exister dans UML 2.0 et elle est déjà matérialisée dans un atelier tel qu'Objecteering entre les méthodes. Elle pourrait induire une spécialisation entre référents. Une propriété générique se compose de plusieurs référents, ces derniers étant ordonnés par un ordre partiel de spécialisation (association *spécialise*). Une propriété générique n'a pas forcément toujours de référent. Par exemple, dans un contexte de conception, on peut prévoir l'existence d'une propriété générique pour décrire l'affichage avant de déterminer quels en seront les référents. Enfin l'association *intension* met en relation une classe avec les référents qui constituent son intension.

Mise en pratique Dans notre approche, toutes les cartes doivent être jouées pour obtenir à partir d'une hiérarchie réelle une représentation des propriétés dans ce méta-modèle : règles du langage initial, experts du domaine, analyse du langage naturel, lexiques, heuristiques, etc. De même, après avoir effectué une opération de manipulation, le retour vers un langage concret dépend de règles dont certaines seront spécifiques au langage cible. Entre les deux, nous jouerons sur le méta-modèle. Dans l'état actuel des travaux, nous avons essentiellement étudié la définition d'heuristiques basées sur des règles de bonne conception.

Ainsi, dans le rapport [DHLR01b] nous nous sommes attachés à proposer des heuristiques pour identifier des référents et propriétés génériques dans un diagramme statique UML. Le même type d'étude a été mené pour Java dans nos travaux sur les métriques de généralisation/spécialisation [DHL$^+$01a, DHL$^+$02]. Pour donner un exemple simple automatisé, on peut utiliser en Java la signature d'une méthode (sans la liste d'exceptions) pour déterminer une méthode générique. Ses divers référents sont ensuite identifiés d'après leur code et leur liste d'exceptions. Un référent de méthode abstraite est naturellement spécialisé par un référent correspondant à une forme concrète si la liste d'exceptions le permet.

Un appel à la « super-méthode » peut être un indice de spécialisation, à manipuler bien sûr avec précautions. M. Dao a rapporté dans [DHLR01b] une étude de telles heuristiques sur une méthode de l'application Pétra [dt97b, dt97a].

Dans le contexte de Java, nous avons proposé ainsi une construction de l'application *Intension*. L'intension d'une classe C contient un référent r si l'une des conditions suivantes est vérifiée :

- C déclare un co-référent de r,
- C hérite un co-référent de r non masqué (non redéfini) par un autre co-référent de la même propriété générique,
- r généralise un co-référent que C déclare ou hérite sans qu'il soit masqué.

On imagine combien ce type de construction est dépendant du langage. En C++ par exemple, où les liens d'héritage peuvent être rendus impraticables par une directive de protection (`private`), les propriétés héritées de la classe ainsi inaccessible, si elles ne sont pas utilisées, peuvent ne pas être considérées dans l'intension. Toujours en C++, une méthode même masquée est toujours applicable sur une instance grâce à l'opérateur de portée, ce qui la rend candidate à être dans l'intension. Au contraire en Java, une méthode héritée mais masquée pour une classe C donnée et que C ne peut atteindre par une chaîne d'appels à *super* peut être absente de l'intension de C.

Synthèse et notations Pour conclure, nous proposons de travailler avec une interprétation de la hiérarchie de classes à traiter dans ce méta-modèle qui met en avant la spécialisation des classes et de leurs caractéristiques (les référents). Une classe est décrite dans notre langage par une intension, formée d'un ensemble de symboles (référents) eux-mêmes organisés dans une forêt d'ordres partiels (de spécialisation). Le calcul de ces symboles se fait à partir de la description des classes dans le langage de programmation (comme Java ou C++) ou dans le langage de modélisation (comme UML). C'est une description assez proche des systèmes de description attribut/valeur propres à certaines approches de regroupement conceptuel, où les propriétés génériques correspondraient à des attributs et les référents à des valeurs. Mais elle est riche de la prise en compte de relations de spécialisation entre les valeurs.

Nous proposons pour finir quelques notations utiles pour la suite.

Définition 2.1 (Hiérarchie de classes) *Une hiérarchie de classes est un graphe orienté sans circuit $GH = (\mathcal{C},\mathcal{U})$, où $\mathcal{C}$ est l'ensemble des classes et $\mathcal{U}$ l'ensemble des liens de généralisation/spécialisation.*

Définition 2.2 (Ordre de généralisation/spécialisation) *GH induit un ordre noté $H = (\mathcal{C}, <_H)$. H sera la notation utilisée le plus fréquemment pour parler d'une hiérarchie.*

On note ainsi les ensembles de propriétés :

- $\mathcal{COR}_{ef}$ est l'ensemble des co-référents,
- $\mathcal{R}_{ef}$ est l'ensemble des référents.

Définition 2.3 (Propriété générique) *Une propriété générique est un ordre partiel $P_{gen} = (P, \leq_P)$ avec $P \subseteq \mathcal{R}_{ef}$. On note $\mathcal{P}_{gen}$ l'ensemble des propriétés génériques. Les propriétés génériques forment une partition de $\mathcal{R}_{ef}$. L'union des ordres des propriétés génériques reste un ordre sur $\mathcal{R}_{ef}$, noté $\leq_{\mathcal{P}_{gen}}$.*

Définition 2.4 (Propriétés d'une classe) *Les applications suivantes associent à une classe $C \in \mathcal{C}$ ses différentes descriptions :*

- $declar_{cor} : \mathcal{C} \longrightarrow \mathcal{COR}_{ef}$
  $declar(C)$ associe à C les co-référents qu'elle déclare (dans un langage de programmation ou de modélisation)
- $Intension_r : \mathcal{C} \longrightarrow \mathcal{R}_{ef}$
  $Intension_r(C)$ associe à C l'ensemble de référents qui la décrive
- $propre_r : \mathcal{C} \longrightarrow \mathcal{R}_{ef}$
  $propre_r(C)$ est l'ensemble des référents de C qui n'apparaissent pas dans les super-classes de C
  $propre_r(C) = Intension_r(C) \setminus \{r/r \in Intension_r(C') \text{ et } C' >_H C\}$
- $declar_r : \mathcal{C} \longrightarrow \mathcal{R}_{ef}$
  $declar_r(C)$ est l'ensemble des plus petits référents propres à C.
  $declar_r(C) = min_{\leq_{\mathcal{P}_{gen}}}(propre_r(C))$
- $herit_r : \mathcal{C} \longrightarrow \mathcal{R}_{ef}$
  $herit_r(C)$ est l'ensemble des référents de C qui ne sont pas dans $propre_r(C)$.

2.6 Structure de la classification

La structure est généralement un ordre partiel : cela répond à un besoin naturel de modélisation utilisant des critères multiples. Si le langage cible n'admet pas de hiérarchie d'héritage multiple comme Smalltalk ou Java dans le cas des classes, des procédures de transformation devront être envisagées [CMR02, Rou02b].

Cependant notre problématique générale apporte quelques autres contraintes que nous souhaiterions voir respecter par la classification produite.

Inclusion des intensions et liens de classification Une classe qui en spécialise une autre devrait inclure aussi son intension. Inversement l'inclusion entre intensions détermine la spécialisation.

Définition 2.5 (Accord inclusion/liens) *Soit une hiérarchie $H = (\mathcal{C}, <_H)$. Une hiérarchie vérifie l'accord inclusion/lien si pour toutes classes $C_1, C_2 \in \mathcal{C}$, $C_1 \leq_H C_2$ si et seulement si $Intension_r(C_1) \supseteq Intension_r(C_2)$*

Accord entre les ordres de spécialisation La spécialisation des classes devrait respecter la spécialisation des référents qui les décrivent (et inversement).

Définition 2.6 (Accord liens/pgen) *Soit une hiérarchie $H = (\mathcal{C}, <_H)$. Une hiérarchie vérifie l'accord lien/pgen lorsque pour tous référents $f_1, f_2 \in P$ et toutes classes C_1, C_2 avec $f_1 \in declar_r(C_1)$ et $f_2 \in declar_r(C_2)$ on a $f_1 \leq_P f_2$ si et seulement si $C_1 \leq_H C_2$.*

Factorisation et généralisation La factorisation doit être maximale et ceci se décline selon trois acceptions. D'une part, un référent ne devrait apparaître que dans une seule classe. D'autre part, les référents présents dans la hiérarchie de classes devraient faire apparaître toutes les généralisations que les propriétés génériques leur connaissent. Par exemple, figure 2.6, le référent *abstract-m* manquerait dans une super-classe commune de C_1 et C_2 pour généraliser de manière appropriée la présence de méthodes m concrètes dans ces deux classes. Dans ce cas simple, c'est parfait, mais demander d'avoir toutes les généralisations serait peut-être parfois excessif. On peut donc se restreindre à demander que pour toute paire de référents d'une même propriété générique déclarés dans la hiérarchie, on voit apparaître aussi leur(s) plus petite(s) généralisation(s) commune(s). Ces trois formes de factorisation formalisées et illustrées dans [DHLR01b] dans un cadre plus restreint sont définies ci-dessous.

Définition 2.7 (Factorisation maximale des propriétés (FM)) *Un référent est dans l'intension propre d'une unique classe.*

Définition 2.8 (factorisation maximale la plus spécifique (FMS)) *Soit une propriété générique $P_{gen} = (P, \leq_P)$, et soit une hiérarchie de classes GH, une FMS est une FM qui vérifie de plus que pour $f_1, f_2 \in P$ et $C_1, C_2 \in \mathcal{C}$ avec $f_1 \in declar_r(C_1)$ et $f_2 \in declar_r(C_2)$, tout plus petit majorant de f_1 et f_2 dans $<_P$ est dans un ensemble $propre_r(C)$ où C est une super-classe commune de C_1 et C_2.*

Définition 2.9 (Factorisation Maximale Exhaustive (FME)) *Soit une propriété générique $P_{gen} = (P, \leq_P)$, et soit la hiérarchie de classes GH, une FME est une FM qui vérifie de plus que pour $f_1, f_2 \in P$ et $C_1, C_2 \in \mathcal{C}$ avec $f_1 \in declar_r(C_1)$ et $f_2 \in declar_r(C_2)$, et pour tout majorant f de f_1 et f_2 dans $<_P$ il existe une classe C avec $f \in declar_r(C)$.*

Ces factorisations ne déterminent pas une structure unique, en particulier elles ne mettent pas de contraintes sur le nombre de classes nécessaires pour assurer la factorisation : par exemple lorsque deux classes possèdent (et sont les seules) deux référents communs, ceux-ci peuvent être factorisés dans une ou deux super-classes communes, ces dernières pouvant être reliées ou non par un lien de généralisation/spécialisation, ce qui produit déjà 4 possibilités.

2.7 Procédures automatiques

Modalités des procédures Nous avons vu précédemment la place que peut occuper la classification de classes dans le cycle de vie du logiciel, mais nous n'en avons pas décrit les modalités. Diverses opérations peuvent bénéficier d'une assistance :

- la construction proprement dite (« from scratch »),
- l'évolution par ajout/retrait de classes, ajout/retrait/modification de propriétés et de liens,
- la reconstruction totale remettant tout en cause,
- la fusion de hiérarchies provenant de développements séparés.

Par ailleurs ces opérations communes agissent sur les classes et manipulent en première approximation des co-référents. Les hypothèses que nous avons choisies nous conduisent également à considérer la manipulation des référents et des propriétés génériques. Pour prendre un exemple, ajouter un co-référent à une classe demande également d'identifier ou de construire puis de placer le référent correspondant dans une propriété générique, l'un pouvant avoir une influence sur l'autre [DVH+02]. Il peut donc y avoir définition d'opérations de classification de référents préalablement ou parallèlement à la classification des classes.

Propriétés des procédures Il est utile de définir un certain nombre de propriétés permettant de juger des procédures de construction automatique de classifications. Nous en donnons deux exemples ci-dessous.

Définition 2.10 (Robustesse (approches par modification incrémentale)) *Un ensemble d'algorithmes incrémentaux est robuste si le résultat de l'application d'un ensemble de transformations par ces algorithmes ne dépend pas de l'ordre d'application.*

Par exemple, l'ajout successif de n classes à une hiérarchie par un algorithme incrémental devrait produire le même résultat quel que soit l'ordre d'ajout des classes. Ceci n'est en particulier pas vrai pour les algorithmes de classification conceptuelle incrémentale qui dépendent de l'ordre « d'arrivée » des instances à classer.

DÉFINITION 2.11 (PRÉSERVATION DES INTENSIONS (APPROCHES-RESTRUCTURATION)) *Un algorithme préserve les intensions si à toute classe de la hiérarchie d'origine on peut associer (par une application b) une classe de la hiérarchie résultat possédant la même intension.*

Cette propriété assure que les concepts initiaux ne sont pas perdus.

DÉFINITION 2.12 (PRÉSERVATION DE LA STRUCTURE (APPROCHES-MODIFICATION)) *Un algorithme préserve la structure lorsque les intensions sont préservées et que pour toute paire de classes C_1, C_2 de la hiérarchie d'origine, C_1 sous-classe de C_2 si et seulement si $b(C_1)$ sous-classe de $b(C_2)$*

Cette propriété n'est intéressante que si l'organisation d'origine est déjà correcte.

Choix des procédures Les méthodes proposées dans le cadre de l'analyse de données et de l'apprentissage sont dédiées à la production de classifications (production de *clusters*) qui servent à faire de la prédiction et ne sont pas adaptées à la structuration des données [Bou96]. Elles se basent le plus souvent sur des critères de minimisation de la similarité *inter-cluster* et de maximisation de la similarité *intra-cluster*. En analyse de données, la description des clusters est essentiellement numérique et ne peut être retenue pour notre propos. Si la description retenue pour le cluster, lorsqu'elle peut être comprise comme une intension, est formée par des caractéristiques statistiquement utiles, d'une part elle est incomplète, d'autre part elle n'est pas parfaitement correcte pour tous les exemples regroupés. Un autre problème majeur est que la hiérarchie résultante n'est généralement pas une hiérarchie de spécialisation des intensions (ce n'est pas son but), et un sous-concept peut posséder des caractéristiques qui contredisent celles d'un super-concept comme on le voit parfaitement dans les exemples de [Dec93]. Ces aspects font que nous n'avons pas pensé dans un premier temps les utiliser. L'expérience maintenant acquise montre cependant que lors des opérations de modélisation les classes du génie logiciel passent par des phases où elles sont décrites de manière incomplète, voire légèrement erronées, ou avec des stratégies de description qui ne permettent pas de repérer des caractéristiques pourtant communes. On pourrait donc imaginer que dans ces phases de conception les classes soient considérées avec de tels outils de regroupement, qui permettraient au concepteur de prendre conscience de problèmes d'incomplétude dans la description (deux classes regroupées car elles sont très proches peuvent indiquer que l'une est imparfaitement décrite).

Une autre voie, ouverte dans le cadre des langages à objets par [GM93], se dessine avec l'utilisation de techniques dérivées de la théorie des treillis déjà largement utilisées depuis les années 70. Ces techniques, basées sur la construction du treillis de Galois associé à une relation binaire (ou treillis de concepts), sont explorées dans de multiples domaines où l'on cherche à structurer un ensemble d'objets partageant des caractéristiques communes [GMMM95]. Les propriétés attendues pour la structure de la classification vont dans ce sens aussi puisque nous montrerons que la FMS est obtenue grâce à ce modèle. C'est donc naturellement la structure recherchée, que nous allons définir dans le chapitre suivant, qui va guider la définition des procédures de classification.

2.8 Conclusion

Nous avons montré la diversité des problèmes de classification dans les approches à objets et défini notre problème comme la recherche de méthodes automatiques de construction d'une classification de classes. La variété des langages dans les approches à objets nous a conduit à proposer un modèle général de spécialisation dans lequel les propriétés descriptives des classes, également appelées référents, sont détachées des formes syntaxiques et organisées par spécialisation. Les caractéristiques de notre problème de classification (recherche de factorisation et de généralisation) nous dirigent assez directement dans une voie tracée par R. Godin qui consiste à utiliser une technique dérivée de l'utilisation du treillis de Galois associé à une relation binaire. Nous en donnerons les ingrédients théoriques au chapitre suivant avant de revenir sur notre problème de classification.

Chapitre 3

Quelques notes sur le treillis et la sous-hiérarchie de Galois

Préambule

Ce chapitre rend compte pour une part de mes lectures favorites, celles qui ont formé ma connaissance et ma compréhension de l'analyse formelle de concepts et des treillis de Galois. Pour une autre part, qui commence à partir de la définition de la sous-hiérarchie de Galois, il décrit des discussions ou des travaux effectués en collaboration avec A. Berry, H. Dicky, R. Godin, H. Leblanc, M. Liquiere et L. Nourine. La notion de shg-réduction et les perspectives sur la typologie des relations et la généralisation de la sous-hiérarchie de Galois à un cadre non relationnel sont propres à ce mémoire.

3.1 Introduction

Ce chapitre pose le cadre théorique de nos travaux sur la construction de hiérarchies de classes. Il nous donne également l'occasion de faire la synthèse de diverses approches de la notion de treillis de Galois, parfois réduite à celle de treillis de concepts, ainsi que d'ouvrir de nouvelles perspectives de réflexion.

Deux concepts essentiels sont tout d'abord présentés, les correspondances de Galois (partie 3.2), puis les opérateurs de fermeture (partie 3.3). La partie 3.4 définit le treillis de Galois, qui résulte de l'association des éléments fermés induits par une correspondance de Galois. Le cas particulier des treillis de Galois associé à une relation binaire est ensuite exposé (partie 3.5). De ces treillis se déduit un sous-ordre particulier, la sous-hiérarchie de Galois (partie 3.6) qui sert de base à de nombreux travaux sur la construction de hiérarchies de classes. Outre sa définition, diverses problématiques relatives à la sous-hiérarchie de Galois sont évoquées, dont sa relation avec les éléments irréductibles du treillis. Une notion de réduction de relation, qui étend celle qui est déjà connue pour les treillis de Galois est ainsi définie. Pour clore ce chapitre, (Partie 3.7) nous posons la question de généraliser la notion de sous-hiérarchie de Galois hors du contexte des correspondances de Galois associées à des relations binaires. Nous concluons dans la partie 3.8.

3.2 Correspondance de Galois

Selon Phill Schultz [Sch01], la notion abstraite de *correspondance de Galois* aurait été introduite par G. Birkhoff [Bir40]. Dans une version ultérieure de ses travaux [Bir67], G.

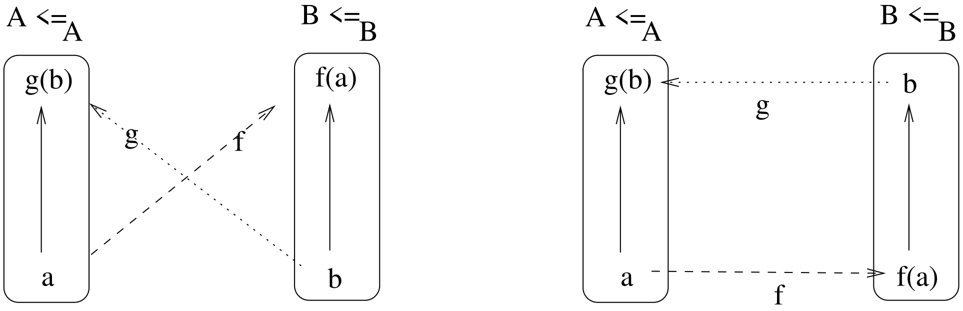


FIG. 3.1 – Correspondance de Galois et paire résiduée

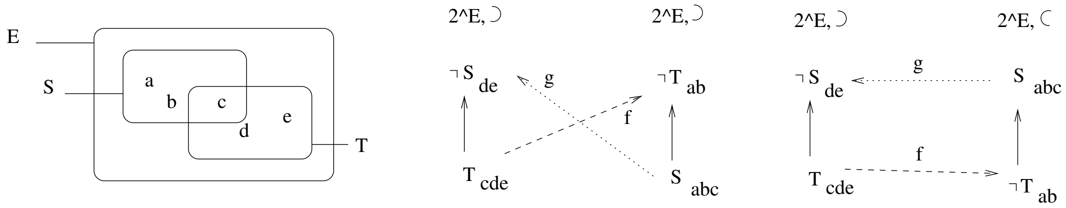


FIG. 3.2 – Correspondance de Galois et paire résiduée pour $(\neg, \neg)$

Birkhoff l'attribue à Oystein Ore [Ore44], qui la décrit lui-même comme une généralisation d'une correspondance établie par la théorie classique de Galois entre sous-groupes et sous-corps. Deux aspects concordent donc : l'apparition de la notion dans les années 1940 et son origine dans la théorie de Galois.

Dans la première acception, les correspondances de Galois se présentent sous une forme contravariante comme des couples d'applications qui mettent en correspondance inverse les éléments de deux ensembles ordonnés (voir Fig. 3.1, gauche). Différentes définitions équivalentes de la forme contravariante ont été proposées.

DÉFINITION 3.1 (CORRESPONDANCE DE GALOIS [BIR67, BM70])
Soient deux ordres partiels $(A, \leq_A)$ et $(B, \leq_B)$. Un couple (f, g) d'applications telles que $f : A \to B$ et $g : B \to A$ est une correspondance de Galois si :
- *f décroissante : pour tout $a, a' \in A$, $a \leq_A a'$ implique $f(a) \geq_A f(a')$*
- *g décroissante : pour tout $b, b' \in B$, $b \leq_B b'$ implique $g(b) \geq_B g(b')$*
- *leurs composées gf et fg sont extensives : pour tout $a \in A$ et $b \in B$, $a \leq_A g(f(a))$ et $b \leq_B f(g(b))$*

Ces correspondances mettent en exergue des dualités dont on peut trouver de nombreux exemples en mathématiques [Bir67, Hil01, Bac01]. Nous ne proposons ici que deux exemples très simples et suffisants pour notre propos.

EXEMPLE 3.1 (COMPLÉMENTAIRE [BAC01]) *Pour un ensemble fini E, notons $\neg$ l'application qui associe à tout sous-ensemble X de E son complémentaire dans E. $(\neg, \neg)$ est une correspondance de Galois entre $(2^E, \supseteq)$ et $(2^E, \supseteq)$ (voir Figure 3.2).*

EXEMPLE 3.2 (ATOMES [BM70]) *Soit un treillis fini T, muni d'un minorant universel, et E l'ensemble de ses atomes (sommets successeurs du minorant universel). Soit $f : 2^E \to T$, définie, pour tout $A \in 2^E$, par $f(A) = \vee\{a/a \in A\}$. f associe donc à un ensemble d'atomes leur borne supérieure. Soit $g : T \to 2^E$, définie, pour tout élément $t \in T$, par $g(t) = \{a/a \in E, a \leq_T t\}$. g associe à un élément du treillis l'ensemble des atomes qui lui*

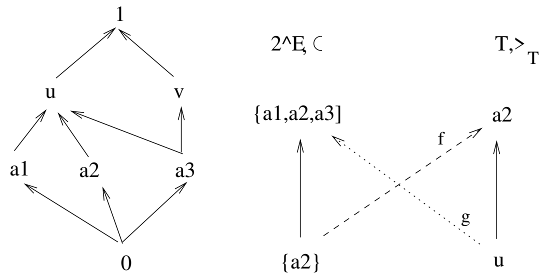


FIG. 3.3 – Correspondance de Galois entre un treillis et ses ensembles d'atomes

sont inférieurs. (f, g) *est une correspondance de Galois entre* $(2^E, \subseteq)$ *et* $(T, \geq_T)$*, illustrée par la figure 3.3.*

Une deuxième définition est attribuée à J. Schmidt par [Bir67] et mentionnée parfois comme « relation de Pickert » [Pol98].

DÉFINITION 3.2 (CORRESPONDANCE DE GALOIS [DP91])
Soient deux ordres partiels $(A, \leq_A)$ *et* $(B, \leq_B)$*. Un couple* (f, g) *d'applications telles que* $f : A \to B$ *et* $g : B \to A$ *est une correspondance de Galois si pour tout* $a \in A$ *et* $b \in B$*,* $a \leq_A g(b)$ *équivaut à* $b \leq_B f(a)$*.*

Une troisième définition (forme duale de celle proposée dans [Bac01]) met en avant le fait que $g(b)$ est le plus grand x dont l'image par f dépasse b.

DÉFINITION 3.3 (CORRESPONDANCE DE GALOIS [BAC01])
Soient deux ordres partiels $(A, \leq_A)$ *et* $(B, \leq_B)$*. Un couple* (f, g) *d'applications telles que* $f : A \to B$ *et* $g : B \to A$ *est une correspondance de Galois si :*
- f *décroissante : pour tout* $a, a' \in A$*,* $a \leq_A a'$ *implique* $f(a) \geq_A f(a')$
- fg *est extensive : pour tout* $b \in B$*,* $b \geq_B f(g(b))$
- *pour tout* $a \in A$ *et* $b \in B$*,* $b \leq_B f(a)$ *implique* $a \leq_B g(b)$

Par ailleurs, les mathématiciens utilisent également une forme duale (covariante) sous le nom de paire résiduée/résiduelle [Pol98]. Il est important de la mentionner car elle est parfois utilisée sous le même nom de *correspondance de Galois* sans plus de précisions.

DÉFINITION 3.4 (PAIRE RÉSIDUÉE/RÉSIDUELLE [BAC01])
Soient deux ordres partiels $(A, \leq_A)$ *et* $(B, \leq_B)$*. Un couple* (f, g) *d'applications telles que* $f : A \to B$ *et* $g : B \to A$ *est une correspondance de Galois si pour tout* $a \in A$ *et* $b \in B$*,* $a \leq_A g(b)$ *équivaut à* $f(a) \leq_B b$*.*

Le rapport entre les deux approches est simple à établir [Pol98] et explique l'abus de langage. Si (f, g) est une forme résiduée/résiduelle pour $(A, \leq_A)$ et $(B, \leq_B)$, c'est aussi une correspondance de Galois pour $(A, \leq_A)$ et $(B, \geq_B)$.

La forme résiduée est plus adaptée à l'appréhension de certaines situations, comme le montre l'exemple ci-dessous.

EXEMPLE 3.3 (PARTIE ENTIÈRE [BAC01]) *La paire d'applications* $(Ceil, Id)$ *est une paire résiduée/résiduelle entre* $(R, \leq)$ *et* $(N, \leq)$ *(ou encore une correspondance de Galois entre* $(R, \leq)$ *et* $(N, \geq)$*). Ceil désigne l'application qui associe à tout élément de* R *sa partie entière supérieure.*

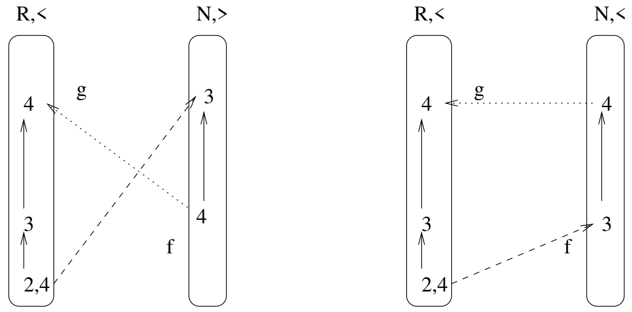


FIG. 3.4 – Correspondance de Galois et paire résiduée pour $(Ceil, Id)$

La figure 3.4 illustre l'intérêt, ne serait-ce que visuel, de la forme résiduée/résiduelle. Pour les mathématiciens, l'intérêt des formes résiduées tient en particulier au fait que la composition de fonctions résiduées est une fonction résiduée. Pour la classification, la résiduation permettrait d'obtenir des dissimilarités tandis que les correspondances de Galois mettraient en évidence des similarités [Pol98].

3.3 Opérateurs de fermeture

DÉFINITION 3.5 (OPÉRATEUR DE FERMETURE [BM70, DP91])
Soit l'ordre partiel $(E, \leq_E)$. Une application $h : E \to E$ est un opérateur de fermeture dans $(E, \leq_E)$ si, pour tous $x, y \in E$, les propriétés suivantes sont satisfaites :
- $x \leq_E y$ *implique* $h(x) \leq_E h(y)$ *(h monotone croissante)*
- $h(x) \geq_E x$ *(h extensive)*
- $h(h(x)) = h(x)$ *(h idempotente)*

EXEMPLE 3.4 (PAVAGE EN COORDONNÉES ENTIÈRES [BM70])
L'application qui associe à tout point (x, y) du plan cartésien le point dont les coordonnées sont les parties entières supérieures de x et y est un opérateur de fermeture sur $Z \times Z$.

DÉFINITION 3.6 (FERMÉ [BM70, DP91])
Soit l'ordre partiel $(E, \leq_E)$. Si l'application $h : E \to E$ est un opérateur de fermeture, un élément x est dit « fermé » lorsque $h(x) = x$.

EXEMPLE 3.5 (PAVAGE EN COORDONNÉES ENTIÈRES (SUITE) [BM70])
Dans l'exemple du pavage en coordonnées entières, les fermés sont naturellement les points de coordonnées entières.

Dans [Bir67] les opérateurs de fermeture « sur un ensemble E » sont définis dans le cas restreint des applications de $(2^E, \subseteq)$ dans lui-même. La plupart des exemples célèbres de fermetures sont en effet de ce type. Ce sera le cas en particulier des fermetures sous-jacentes au treillis de Galois.

EXEMPLE 3.6 (FERMETURE TRANSITIVE [BM70])
L'application qui associe à toute relation binaire R sur un ensemble E sa fermeture transitive est un opérateur de fermeture h de $2^{E\times E} \to 2^{E\times E}$.

Soit $R \in 2^{E\times E}$, $h(R)$ est la relation R' contenant R et telle que pour tous $x, y \in E$, $(x, y) \in R$ et $(y, z) \in R$ implique $(x, z) \in R'$.

Les fermés sont les relations transitives.

Les fermetures sur des treillis ont des propriétés particulières qui seront utiles pour comprendre les constructions ultérieures.

Propriété 3.1 (Stabilité de l'ensemble des fermés pour l'infimum [BM70])
Soit un opérateur de fermeture h sur un treillis T. Notons $x \wedge y$ l'infimum de x et y. Si x et y sont fermés pour h, alors $x \wedge y$ l'est également.

Cette propriété n'est pas vraie de manière générale pour le supremum (noté $x \vee y$). On sait seulement que $x \vee y \leq h(x \vee y)$ par croissance de h. Ainsi l'intersection de deux relations transitives est transitive, tandis que leur union ne l'est généralement pas. Dans l'exemple du pavage, l'infimum comme le supremum sont également fermés, dans ce cas particulier l'ordre est topologique.

La propriété suivante 3.2 montre que cette stabilité pour l'infimum admet une réciproque selon laquelle toute partie stable pour l'infimum et contenant l'élément maximum définit un opérateur de fermeture. Nous retrouverons ce type de réciprocité entre structures et constructions tout au long de notre avancée jusqu'aux sous-hiérarchies de Galois : une construction (ici l'opération de fermeture sur T) induit une structure (une partie de T, formée des éléments fermés, stable pour l'infimum et contenant le maximum). Inversement toute structure du type de la structure induite (partie de T stable pour l'infimum et contenant le maximum) peut être obtenue à partir d'une opération de fermeture sur T. Cet aspect est important car il justifiera en partie l'adéquation du treillis et de la sous-hiérarchie de Galois comme modèles pour les hiérarchies d'héritage.

Propriété 3.2 (Caractérisation des fermetures sur un treillis fini [BM70])

Les fermetures sur un treillis fini T sont en correspondance bijective avec les parties de T stables pour l'infimum et contenant le maximum.

Le fait que le supremum de deux fermés ne soit généralement pas fermé interdit de considérer l'ensemble des fermés associés à une fermeture sur un treillis T comme sous-treillis de T. L'ensemble des fermés va néanmoins pouvoir être muni d'une structure de treillis comme nous le voyons ci-dessous.

Propriété 3.3 (Treillis des fermés)
Soit F l'ensemble des fermés d'un opérateur de fermeture h sur un treillis fini T. Soient les opérations suivantes pour $x, y \in F$:
- $x \wedge_F y = x \wedge y$
- $x \vee_F y = h(x \vee y)$

$(F, \wedge_F, \vee_F)$ est un treillis.

Exemple 3.7 (Fermeture transitive (suite) [BM70])
L'ensemble F des relations binaires transitives sur un ensemble E forme un treillis pour les deux opérations suivantes :
- $R_1 \wedge_F R_2 = R_1 \cap R_2$
- *$R_1 \vee_F R_2$ est la fermeture transitive de $R_1 \cup R_2$*

3.4 Treillis de Galois

Les deux notions que nous avons évoquées ci-dessus, en l'occurrence les correspondances de Galois et les opérateurs de fermeture vont à présent se rejoindre.

Propriété 3.4 (Fermetures de Galois [DP91])
Soit une correspondance de Galois (f, g) *entre deux ensembles ordonnés* $(A, \leq_A)$ *et* $(B, \leq_B)$. *Les composées* $h_A = gf$ *et* $h_B = fg$ *sont des opérateurs de fermeture. Notons* F_{h_A} *les fermés de* h_A *et* F_{h_B} *les fermés de* h_B.

La propriété suivante permet de caractériser les fermés.

Propriété 3.5 (Fermés et images par f et g [DP91])
Soit une correspondance de Galois (f, g) *entre deux ensembles ordonnés* $(A, \leq_A)$ *et* $(B, \leq_B)$, *et notons* $h_A = gf$ *et* $h_B = fg$. *Tout élément* F_{h_A} *est image d'un élément de* B *par* g. *Symétriquement, tout élément de* F_{h_B} *est image d'un élément de* A *par* f.

Par application des résultats précédents, si A (resp. B) est un treillis complet, F_{h_A} (resp. F_{h_B}) peut être muni d'une structure de treillis.

Propriété 3.6 (Treillis des fermetures de Galois)
Soit une correspondance de Galois (f, g) *entre deux treillis complets* $(A, \wedge_A, \vee_A)$ *et* $(B, \wedge_B, \vee_B)$.
$\mathcal{L}_A = (F_{h_A}, \wedge_{F_{h_A}}, \vee_{F_{h_A}})$ *est un treillis avec, pour* $x, y \in F_{h_A}$:
- $x \wedge_{F_{h_A}} y = x \wedge_A y$
- $x \vee_{F_{h_A}} y = h_A(x \vee_A y)$

De même, $\mathcal{L}_B = (F_{h_B}, \wedge_{F_{h_B}}, \vee_{F_{h_B}})$ *est un treillis avec, pour* $x, y \in F_{h_b}$:
- $x \wedge_{F_{h_B}} y = x \wedge_B y$
- $x \vee_{F_{h_B}} y = h_B(x \vee_B y)$

Les deux treillis $\mathcal{L}_A$ et $\mathcal{L}_B^d$ (dual de $\mathcal{L}_B$) sont isomorphes. f est un isomorphisme de $\mathcal{L}_A$ dans $\mathcal{L}_B^d$ et, sur cette restriction, $g = f^{-1}$.

Dans le cas plus restreint des ordres, f et g restreintes aux ensembles de fermés F_{h_A} et F_{h_B} sont également des isomorphismes (inverses l'un de l'autre) entre $(F_{h_A}, \subseteq)$ et $(F_{h_B}, \supseteq)$ [DP91].

Pour en revenir au cas des treillis, le produit cartésien $P = F_{h_A} \times F_{h_B}$ peut être muni d'une structure de treillis.

Définition 3.7 (Treillis du produit des ensembles de fermés [BM70])
Soient les opérations définies de la manière suivante pour deux couples (x_1, y_1) *et* (x_2, y_2) *de* $F_{h_A} \times F_{h_B}$:
- $(x_1, y_1) \wedge_P (x_2, y_2) = (x_1 \wedge_{F_{h_A}} x_2, y_1 \vee_{F\ h_B} y_2)$
- $(x_1, y_1) \vee_P (x_2, y_2) = (x_1 \vee_{F\ h_A} x_2, y_1 \wedge_{F_{h_B}} y_2)$

$\mathcal{L} = (F_{h_A} \times F_{h_B}, \wedge_P, \vee_P)$ *est un treillis.*

Enfin, la restriction du précédent treillis aux couples de fermés en correspondance par f (ou de manière équivalente par g) induit un treillis particulier : le treillis de Galois.

Définition 3.8 (Treillis de Galois)
Le treillis de Galois d'une correspondance de Galois (f, g) *est le sous-treillis* $\mathcal{L}_{\mathcal{G}}$ *de* $\mathcal{L}$ *(treillis du produit des fermés tel que décrit def. 3.7) restreint aux couples* (x, y) *tels que* $y = f(x)$ *(ou de manière équivalente* $x = g(y)$*).*

Exemple 3.8 (Atomes (suite))
La Figure 3.5 présente sur l'exemple les diagrammes des fonctions f, g, *les treillis des fermés de* gf, fg *et enfin le treillis de Galois.*

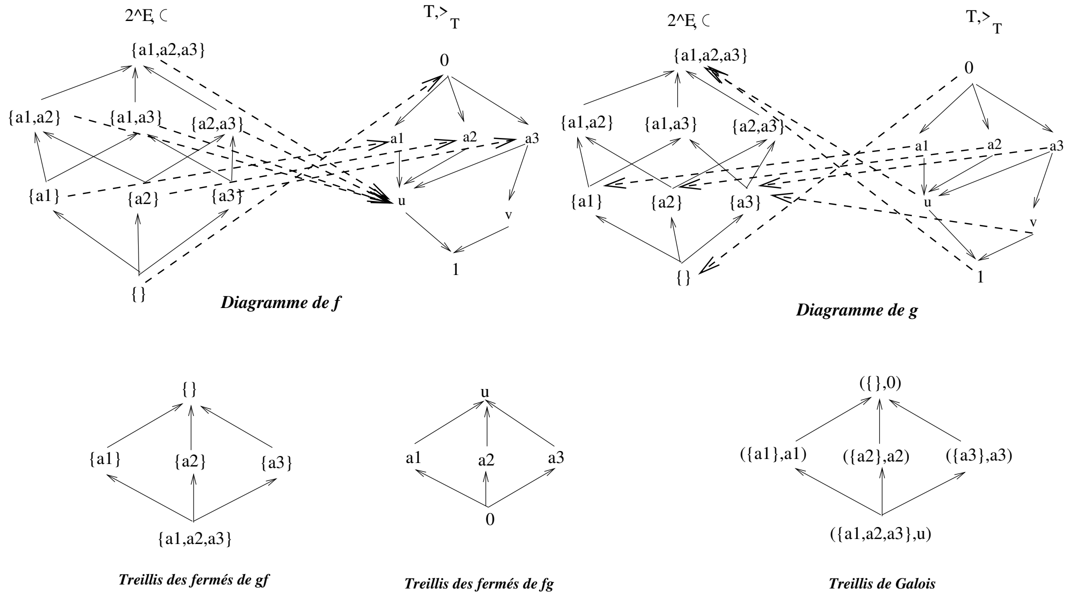


FIG. 3.5 – Treillis de Galois pour l'exemple 3.2 (atomes)

Properties

Classes		a	b	c	d	e	f
	1	x	x				
	2			x			
	3	x		x	x		x
	4	x		x		x	x
	5	x					

FIG. 3.6 – Une Relation R1 entre E=1..5 et F=a..f

3.5 Treillis de Galois d'une relation binaire

Nous présentons à présent un cas particulier de treillis de Galois, construit sur la base d'une relation binaire.

Considérons deux ensembles finis E et F et une relation binaire $R \subseteq E \times F$. Dans le contexte de la construction des hiérarchies de classes, E correspondrait à l'ensemble des classes, tandis que F représenterait l'ensemble des propriétés. La relation R permet d'associer à une classe l'ensemble des propriétés dont elle dispose : $(x, y) \in \mathcal{R}$ pour $x \in E$ et $y \in F$ signifie que x possède y. La Figure 3.6 propose une relation binaire qui servira à illustrer les définitions suivantes.

La construction est fondée sur les deux applications duales f et g définies ci-dessous, qui jouent des rôles symétriques.

DÉFINITION 3.9 (APPLICATIONS CARACTÉRISTIQUES D'UNE RELATION BINAIRE)
$f : \mathcal{P}(E) \rightarrow \mathcal{P}(F)$
$X \longmapsto \{y \in F \mid \forall x \in X, (x, y) \in \mathcal{R}\}$
$g : \mathcal{P}(F) \rightarrow \mathcal{P}(E)$
$Y \longmapsto \{x \in E \mid \forall y \in Y, (x, y) \in \mathcal{R}\}$

Pour $X \subseteq E$, $f(X)$ représente l'ensemble des propriétés partagées par toutes les classes de X. Sur l'exemple, si $X = \{2, 3, 4\}$, $f(X)$ est l'ensemble des propriétés partagées par les classes 2, 3 et 4, c'est-à-dire $\{c\}$. Symétriquement, si $Y \subseteq F$, $g(Y)$ est l'ensemble des classes possédant chacune toutes les propriétés de Y. Ainsi, avec $Y = \{a, c\}$, $g(Y)$ est l'ensemble des classes qui possèdent à la fois a et c, nous avons ainsi $g(Y) = \{3, 4\}$.

PROPRIÉTÉ 3.7 *Soient f et g deux applications caractéristiques d'une relation binaire R (Déf. 3.9). $(f,\ g)$ forme une correspondance de Galois entre $(\mathcal{P}(E), \subseteq)$ et $(\mathcal{P}(F), \subseteq)$.*

Il est aisé de démontrer que f et g sont décroissantes et que leurs composées $h_E = gof$ et $h_F = fog$ sont extensives [BM70].

Pour illustrer l'opération de fermeture effectuée, l'ensemble de propriétés $\{a, c, f\}$ est un fermé pour h_F, tandis que $\{a, c\}$ n'en est pas un. On peut en effet constater que $g(\{a, c\}) = \{3, 4\}$ et que $f(g(\{a, c\})) = f(\{3, 4\}) = \{a, c, f\}$, c'est-à-dire que si l'on s'intéresse aux classes qui possèdent $\{a, c\}$, on remarque que ces classes partagent non seulement a et c mais aussi f.

Comme dans le cas général, et par application des définitions des sections précédentes, les deux opérateurs de fermeture h_E et h_F associés à R par le biais de f et g définissent deux treillis isomorphes, dont les ensembles de sommets sont les fermés. Contrairement à la propriété 3.6, la définition suivante utilise directement le dual du treillis associé aux ensembles de fermés de h_F.

DÉFINITION 3.10 (TREILLIS DES FERMÉS ASSOCIÉS À UNE RELATION BINAIRE)
L'ensemble des fermés de h_E, noté $\mathcal{L}_E$ (resp. $\mathcal{L}_F$ pour h_F), ordonné par l'inclusion ensembliste $\subset$ (resp. par $\supset$) forme un treillis. Pour X_1, X_2 fermés quelconques de h_E (resp. Y_1, Y_2 fermés quelconques de h_F), l'infimum et le supremum sont définis de la manière suivante :

$X_1 \wedge X_2 = (X_1 \cap X_2)$ *(resp.* $Y_1 \wedge Y_2 = h_F(Y_1 \cup Y_2)$*)*
$X_1 \vee X_2 = h_E(X_1 \cup X_2)$ *(resp.* $Y_1 \vee Y_2 = Y_1 \cap Y_2$*)*

La Figure 3.7 présente à gauche le treillis $\mathcal{L}_E$ et au centre le treillis $\mathcal{L}_F$. Il peut être utile de s'attarder sur quelques calculs d'infimum et de supremum. Pour $X_1 = \{1\}$ et $X_2 = \{3, 4\}$, on constate que $X_1 \vee X_2 = h_E(X_1 \cup X_2) = \{1, 3, 4, 5\}$ ce qui montre en particulier que $\{1, 3, 4\}$ n'est pas fermé. Lorsqu'on considère au contraire $X_1 = \{3\}$ et $X_2 = \{4\}$, on trouve que $X_1 \vee X_2 = h_E(X_1 \cup X_2) = \{3, 4\}$ ce qui montre que $\{3, 4\}$ est fermé. Enfin pour $X_1 = \{1, 3, 4, 5\}$ et $X_2 = \{2, 3, 4\}$, le calcul de l'infimum s'effectue sans fermeture avec $X_1 \wedge X_2 = X_1 \cap X_2 = \{3, 4\}$.

Toujours en suivant le schéma général précédent f (ou g) mettent en bijection $\mathcal{L}_E$ et $\mathcal{L}_F$ et cette association entre les deux treillis définit le treillis de Galois associé à une relation binaire.

DÉFINITION 3.11 (TREILLIS DE GALOIS D'UNE RELATION BINAIRE)
Soit une relation binaire R et f et g ses applications caractéristiques associées. Le treillis de Galois associé à R est l'ensemble ordonné noté $(\mathcal{L}, <_{\mathcal{L}})$ défini comme suit. $\mathcal{L}$ est l'ensemble des couples (X, Y), où $X \in \mathcal{L}_E$, $Y \in \mathcal{L}_F$, et $Y = f(X)$ (ou de manière équivalente $X = g(Y)$). Pour deux éléments de L, (X_1, Y_1) et (X_2, Y_2), $(X_1, Y_1) <_{\mathcal{L}} (X_2, Y_2)$ si et seulement si $X_1 \subset X_2$ (ce qui est équivalent à $Y_2 \subset Y_1$). Les opérateurs d'infimum et de supremum consistent à appliquer aux membres gauches et droits des couples les opérations d'infimum et de supremum définies respectivement dans $\mathcal{L}_E$ et $\mathcal{L}_F$.

A ce point, il convient de faire le lien avec la présentation utilisée dans le cadre de l'analyse formelle de concepts (« Formal Concept Analysis » [GW99]), introduite en 1982 par R. Wille. Deux termes fondamentaux sont mis en avant dans cette approche, le contexte

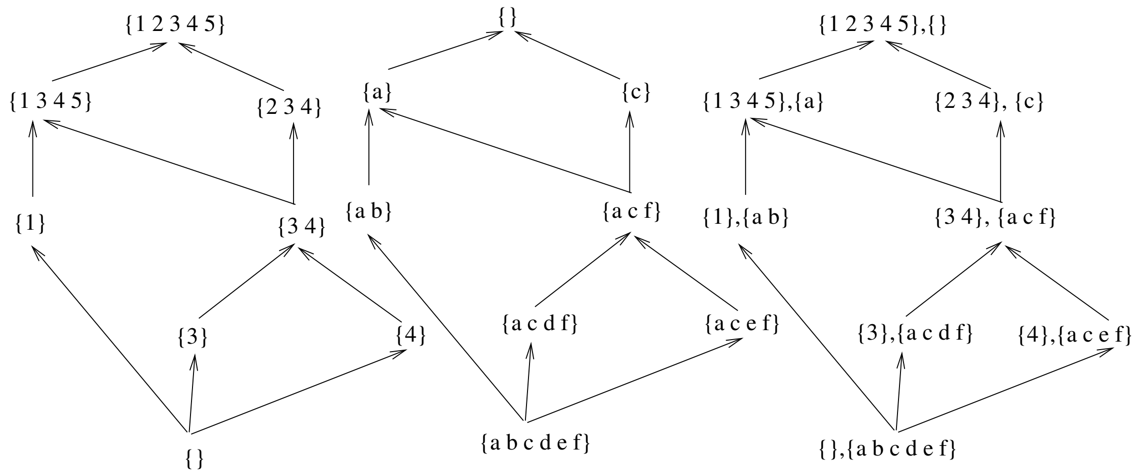


FIG. 3.7 – Treillis $\mathcal{L}_E$, $\mathcal{L}_F$ et $\mathcal{L}$

et les concepts. Cette terminologie est édifiée par référence aux travaux d'Aristote sur l'épistémologie (étude de la connaissance) dans lesquels il développe l'idée que tout concept a une double nature : une *extension* qui correspond à l'ensemble des objets que réunit le concept, et une *intension* constituée par l'ensemble des prédicats (ou propriétés, ou attributs) que vérifient tous les objets.

DÉFINITION 3.12 (CONTEXTE FORMEL) *Un contexte formel est un triplet* (E, F, R)*, où* E *et* F *sont des ensembles finis (représentant respectivement des objets formels et des attributs formels), et* R *est une relation binaire qui associe* E *et* F*.*

Dans notre cas, les objets formels correspondent aux classes et les attributs formels aux propriétés.

DÉFINITION 3.13 (TREILLIS DE CONCEPTS) *Soit* $(\mathcal{L}, <_{\mathcal{L}})$ *le treillis de Galois associé à un contexte formel* (E, F, R)*, ce treillis est aussi appelé le treillis de concepts du contexte formel.*

Un « concept formel » est un élément (X, Y) *de* $\mathcal{L}$*, dont* X *constitue l'extension et* Y *l'intension.*

Pour conclure cette section, il faut noter que les treillis de Galois ne sont pas des treillis (finis) particuliers. Plus précisément, à tout treillis fini on peut associer une infinité de relations (irréductibles au sens de la définition 3.20) dont il est le treillis de Galois [BM70].

Ces relations peuvent être construites en utilisant certains éléments d'un treillis, appelés les éléments « (inf- et sup-) irréductibles » , qui jouent un rôle bien particulier pour la compréhension de la structure des treillis et pour leur codage [AKBLN89, Cas93, HN94, RT01]. Ces éléments ne peuvent en effet s'exprimer comme l'infimum (resp. le supremum) de plusieurs autres éléments strictement plus grands (resp. plus petits).

DÉFINITION 3.14 (INF-IRRÉDUCTIBLE [BM70, DP91]) *Soit* $(T, \leq_T)$ *un treillis fini. Un élément* $v \in T$ *est un élément inf-irréductible si les deux propriétés suivantes sont vérifiées :*

- $v \neq 1$*, où* 1 *est l'élément maximal de* T *s'il existe,*
- $v = a \wedge b$ *implique* $v = a$ *ou* $v = b$*.*

DÉFINITION 3.15 (SUP-IRRÉDUCTIBLE [BM70, DP91]) *Soit* $(T, \leq_T)$ *un treillis fini. Un élément* $v \in T$ *est un élément sup-irréductible si les deux propriétés suivantes sont vérifiées :*

– *$v \neq 0$, où 0 est l'élément minimal de T s'il existe,*
– *$v = a \vee b$ implique $v = a$ ou $v = b$.*

On note usuellement $\mathcal{J}$ l'ensemble des sup-irréductibles (ou *join-irreducible*), $\mathcal{M}$ l'ensemble des inf-irréductibles (ou *meet-irreducible*), et $\mathcal{I}rr$ l'ensemble de tous les éléments irréductibles. La propriété suivante permet de se représenter graphiquement ces éléments dans la relation de couverture de l'ordre.

PROPRIÉTÉ 3.8 ([GW99]) *v est inf-irréductible, également noté $\wedge$-irréductible, (resp. sup-irréductible ou encore $\vee$-irréductible) si et seulement s'il est couvert par (resp. couvre) exactement un élément de T.*

Nous formalisons ainsi la proposition de [BM70] pour trouver une relation binaire dont un treillis donné T est treillis de Galois. Les éléments de $\mathcal{J}$ et de $\mathcal{M}$ sont étiquetés de manière bijective dans E et F respectivement. Deux ensembles sont ensuite associés à chaque sommet x de T. Le premier est l'ensemble des étiquettes des éléments de $\mathcal{J}$ qui précèdent x (au sens large) dans l'ordre, tandis que le deuxième est constitué des étiquettes de tous les éléments de $\mathcal{M}$ qui suivent x dans l'ordre. Chaque sommet de T est enfin considéré comme un pavé maximal d'une relation sur $E \times F$.

PROPRIÉTÉ 3.9 (D'APRÈS [BM70]) *Soit un treillis fini T. On peut trouver une relation R dont T soit le treillis de Galois associé par la construction suivante :*
– *Soient E et F deux ensembles finis d'étiquettes, avec $|E| = |\mathcal{J}|$ et $|F| = |\mathcal{M}|$,*
– *Soient deux fonctions bijectives d'étiquetage $j : \mathcal{J} \longrightarrow E$ et $m : \mathcal{M} \longrightarrow F$*
– *Ces fonctions s'étendent de la manière suivante :*
 $j_e : T \longrightarrow 2^E$, $j_e(x) = \{j(y) | y \in \mathcal{J} \text{ et } y \leq_T x\}$
 $m_e : T \longrightarrow 2^F$, $m_e(x) = \{j(y) | y \in \mathcal{M} \text{ et } y \geq_T x\}$
– *$\{j_e(x), m_e(x) | x \in T\}$ peut s'interpréter comme l'ensemble des pavés maximaux qui déterminent une relation $R \subseteq E \times F$.*

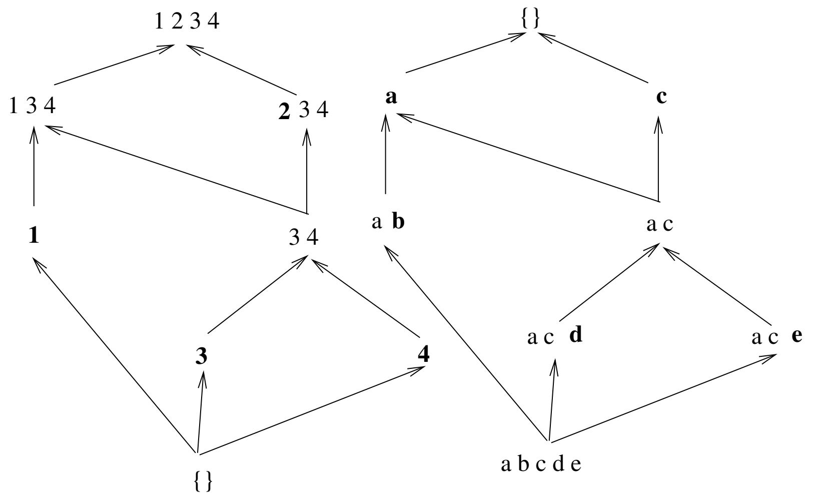
La figure 3.8 donne un aperçu de la construction de la relation binaire dont un treillis est treillis de Galois. En gras apparaissent tout d'abord les étiquettes des éléments sup-irréductibles (à gauche) puis celles des éléments inf-irréductibles (au centre). La relation déduite de l'assemblage des étiquetages se trouve à gauche de la figure.

Pour faire un lien, mais sans rentrer trop en détails, avec la problématique du codage, j_e constitue un codage d'ordre de T c'est-à-dire respecte l'ordre $\leq_T$. Ce codage d'ordre préserve de plus l'infimum.

3.6 Sous-hiérarchie de Galois d'une relation binaire

Différentes simplifications du treillis de Galois associé à une relation binaire, dont les plus simples ont été proposées dans [GMMM95], ont été introduites pour des raisons de complexité (algorithmique ou conceptuelle). Deux aspects sont étudiés dans cette référence : la simplification des étiquettes et la réduction du nombre de concepts.

Une simplification ne peut se faire sans un objectif annoncé. De la référence [GMMM95] qui traite de multiples applications du treillis de Galois qui n'ont pas toutes les mêmes besoins, nous examinons ici les simplifications qui concernent le mieux, de notre point de vue, notre problématique. Le treillis de Galois est le support dont on doit extraire une hiérarchie de classes représentatives d'un ensemble de classes « initiales ». Les classes initiales doivent donc se retrouver dans la structure finale, avec une « sémantique » préservée. De



Application j_e — *Application j_e* — *Relation déduite*

	a	b	c	d	e
1	x	x			
2			x		
3	x		x	x	
4	x		x		x

FIG. 3.8 – Construction d'une relation binaire pour un treillis

manière symétrique, ce qui décrit les classes, c'est-à-dire essentiellement les attributs et les méthodes, doit se trouver « déclaré » quelque part dans la structure. Nous reviendrons plus en détails sur ces différents points.

La réduction du nombre de concepts peut se justifier par le fait que le treillis de Galois associé à une relation binaire $R \subseteq E \times F$ a dans le pire des cas un nombre de concepts de l'ordre de $2^{min(|E|,|F|)}$.

La simplification des concepts eux-mêmes (nous parlerons parfois des « étiquettes » des nœuds du treillis) est très naturelle. Elle se base sur l'observation que les étiquettes portent une information redondante avec l'ordre dans le treillis [GM93]. Une propriété (resp. une classe) qui apparaît dans un élément c du treillis apparaît dans tous les éléments plus petits que c (resp. plus grands). La simplification effectuée considère alors que les propriétés et les classes n'ont besoin d'apparaître que dans l'élément qui les introduit.

3.6.1 Sous-Hiérarchie de Galois : Définition par élimination

La notion de sous-hiérarchie de Galois est à l'origine ([GM93]) définie par une réduction du nombre de concepts qui s'appuie sur la simplification des étiquettes. Un concept disparaît lorsque la simplification des étiquettes le vide de sa substance.

Nous utilisons les notations R, f, g, $\mathcal{L}$, etc. de la section précédente sans les rappeler à chaque occasion.

DÉFINITION 3.16 *Soit (X, Y) un concept de $\mathcal{L}$, le concept associé par simplification des propriétés est noté (X, Y_s) avec $Y_s = Y \setminus herit(Y)$ où $herit(Y) = \{y \in Y \ / \ \exists Y' \in \mathcal{L}_F, y \in Y' \text{ et } Y' \subset Y\}$.*

Les propriétés enlevées de cette manière sont en effet héritées de concepts plus grands que (X, Y) dans le treillis (d'après l'ordre $<_{\mathcal{L}}$). Cette simplification d'un concept s'applique à son intension prise indépendamment. Ainsi, soit un fermé de propriétés $Y \in \mathcal{L}_F$, la simplification mène à $Y_s = Y \setminus \mathrm{herit}(Y)$.

Définition 3.17 *Soit* (X, Y) *un concept de* $\mathcal{L}$*, sa forme simplifiée selon les classes est* (X_s, Y) *avec* $X_s = X \setminus sousclasses(X)$*, où* $sousclasses(X) = \{x \in X \ / \ \exists X_2 \in \mathcal{L}_E, x \in X_2$ *et* $X_2 \subset X\}$.

Une classe qui disparaît de X de cette manière est forcément présente dans l'extension d'un concept plus petit que (X, Y) (pour l'ordre $<_{\mathcal{L}}$).

Une propriété déjà énoncée dans [BM70, DP91] (voir Prop. 3.5) et démontrée, comme les suivantes, dans [HDL00] indique la provenance des fermés de E : ce sont tous des images par g de parties de F.

Propriété 3.10 *Pour tout* $Y \subseteq F$*,* $g(Y)$ *est un fermé pour* h_E.

Une propriété symétrique existe bien entendu pour les fermés de E.

Propriété 3.11 *Quel que soit* $X \subseteq E$*,* $f(X)$ *est un fermé pour* h_F.

Ces propriétés, appliquées aux singletons, sont exploitées ci-dessous dans les propriétés 3.12 et 3.13 (sous une forme qui étend [HDL00]) pour déterminer la position des concepts qui introduisent les éléments de E et de F.

Propriété 3.12 *Si* (X, Y_s) *résulte de la simplification des propriétés du concept* (X, Y)*, alors pour tout* $y \in F$*, les trois propositions suivantes sont équivalentes :*
(i) $y \in Y_s$,
(ii) $g(\{y\}) = X$,
(iii) (X, Y) *est le plus grand concept dans le treillis où* y *apparaît.*
y *est dit « déclaré » par le concept* (X, Y).

Preuve. Pour compléter les preuves de [HDL00], il convient d'ajouter la preuve que (iii) implique (ii). Ceci se déduit aisément de la définition 3.3 de correspondance de Galois appliquée à notre cas. En posant $a = X_0$ et $b = \{y\}$, cette définition nous dit en effet que pour tout fermé $X_0 \subseteq E$, $\{y\} \subseteq f(X_0)$ implique $X_0 \subseteq g(\{y\})$. $g(\{y\})$ est lui-même fermé (Prop. 3.10) et $\{y\} \subseteq f(g(\{y\}))$ (fg extensive). On en déduit aisément que $max_{\subseteq}\{X_0 | \{y\} \subseteq f(X_0)\} = g(\{y\})$, c'est-à-dire que l'intension du plus grand concept contenant y est $g(\{y\})$, ou encore $X = g(\{y\})$. □

Propriété 3.13 *Si* (X_s, Y) *résulte de la simplification des classes du concept* (X, Y)*, alors pour tout* $x \in E$*, les trois propositions suivantes sont équivalentes :*
(i) $x \in X_s$
(ii) $f(\{x\}) = Y$
(iii) (X, Y) *est le plus petit concept du treillis où* x *apparaît.*
x *est dit « définie » par le concept* (X, Y).

La preuve que (*iii*) implique (*i*), qui reste à donner pour compléter la référence [HDL00], s'établit de manière symétrique à ce qui a été proposé pour la propriété 3.12.

Certains fermés simplifiés peuvent se trouver vides, ce qui indique qu'il ne sont utiles à l'introduction d'aucun élément, classe ou propriété suivant le cas. Ou encore, comme le dit à l'inverse la propriété suivante, tous leurs éléments sont introduits par d'autres concepts.

Propriété 3.14 *Soit le concept* (X, Y)*, si* $X_s = \emptyset$ *et* $Y_s = \emptyset$*, on a :*
- $\forall x \in X$*, il existe un concept* $(X_1, Y_1) \in \mathcal{L} \ / \ (X_1, Y_1) <_{\mathcal{L}} (X, Y)$ *et* $x \in X_1$,
- $\forall y \in Y$*, il existe un concept* $(X_1, Y_1) \in \mathcal{L} \ / \ (X, Y) <_{\mathcal{L}} (X_1, Y_1)$ *et* $y \in Y_1$.

	a	b	c	d	e	f
1	x	x	x			
2			x			
3	x		x	x		x
4	x		x		x	x
5	x	x				

Une relation binaire R2 sur E = {1, 2 ,3, 4, 5} et F = {a, b, c, d, e, f}

Fig. 3.9 – Relation R2

La réduction du nombre de concepts du treillis se fait alors en supprimant les sommets dont l'extension et l'intension simplifiées sont vides.

Définition 3.18 *La sous-hiérarchie de Galois $\mathcal{S}$ est le sous-ordre du treillis de Galois $\mathcal{L}$ induit par les concepts simplifiés (X_s, Y_s) tels que $X_s \neq \emptyset$ et $Y_s \neq \emptyset$.*

Les figures 3.9 et 3.10 présentent respectivement une relation binaire $R2$, puis les structure associées suivantes : treillis de Galois, treillis où les concepts ont été simplifiés, enfin sous-hiérarchie de Galois.

La version simplifiée du treillis de Galois, tout comme la sous-hiérarchie de Galois, peuvent être interprétées comme des hiérarchies de classes usuelles. Chaque concept est interprété comme une classe de la hiérarchie résultat. L'extension simplifiée contient les classes initiales définies par le concept. Quand elle est vide, le concept s'interprètera comme une nouvelle classe de factorisation. Quand elle est réduite à une classe, cette dernière peut donner son nom à la classe du résultat. L'intension simplifiée contient les propriétés déclarées par la classe résultat. Quand elle est vide, la classe résultat hérite toutes ses propriétés de ses super-classes. La sous-hiérarchie de Galois préserve la partie la plus informative du treillis : tous les concepts qui définissent au moins une classe ou déclarent au moins une propriété. En ce sens c'est la plus petite structure qui permette une factorisation maximale des propriétés, respecte les relations d'inclusion entre les intensions et rende compte de toutes les classes et de toutes les propriétés d'origine. Le nombre de concepts est réduit de manière significative, puisque d'exponentiel dans le pire des cas il est à présent borné par $|E| + |F|$.

3.6.2 Sous-Hiérarchie de Galois : Définition constructive

Aussi bien pour des raisons algorithmiques que de compréhension, il est utile de considérer la sous-hiérarchie de Galois non pas seulement comme une partie du treillis (par réduction) mais au contraire comme le résultat de la construction de certains fermés particuliers.

Cela donne la définition suivante, dont nous avons démontré qu'elle était équivalente à la précédente dans [HDL00]. Elle est à la base de l'algorithme Ceres [Leb00].

Définition 3.19 ([HDL00]) *Soit une relation binaire $R \subseteq E \times F$, f et g les correspondances de Galois associées.*

Soit $O_C = \{(g(f(\{e\})), f(\{e\}))|e \in E\}$ et $O_P = \{(g(\{a\}), f(g(\{a\})))|a \in F\}$.

La sous-hiérarchie de Galois $\mathcal{S}$ associée à R est l'ensemble des concepts simplifiés de $O_C \cup O_P$, muni de l'ordre suivant. Soient (X_1, Y_1) et (X_2, Y_2) deux concepts de $O_C \cup O_P$

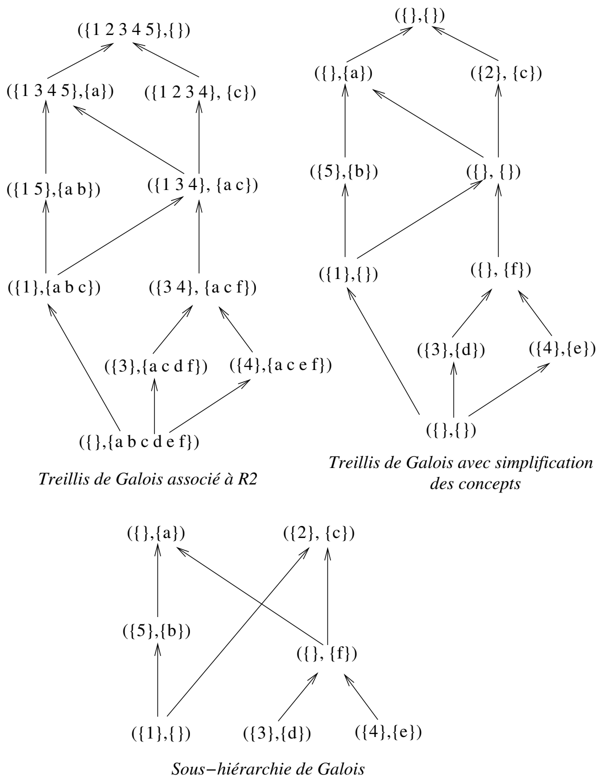


FIG. 3.10 – Simplification des concepts selon les classes (à gauche), les propriétés (au centre), et sous-hiérarchie de Galois (en bas)

(non simplifiés), $(X_{1_s}, Y_{1_s}) <_{\mathcal{S}} (X_{2_s}, Y_{2_s})$ *si et seulement si* $X_1 \subset X_2$ *(ce qui équivaut à* $Y_2 \subset Y_1$*).*

Les éléments de O_C sont construits dans notre cadre de travail à partir de la fermeture des singletons de classes, tandis que les éléments de O_P sont issus de la fermeture des singletons de propriétés. De tels éléments sont appelés « (formal) object concepts » et « (formal) attribute concepts » dans [GW99]. Dans cette référence, on ne s'intéresse cependant pas au sous-ordre induit par ces concepts particuliers. Ils sont introduits essentiellement pour l'identification du contexte formel dans le treillis.

3.6.3 Sous-hiérarchie de Galois et réduction

Eléments irréductibles

Si l'on examine les éléments irréductibles (voir Section 3.5) d'un peu plus près, on peut remarquer qu'ils appartiennent forcément à la sous-hiérarchie de Galois.

PROPRIÉTÉ 3.15 *Les éléments inf-irréductibles et sup-irréductibles du treillis de Galois associés à une relation binaire appartiennent également à la sous-hiérarchie de Galois correspondante. Nous exprimons ceci formellement par* $\mathcal{I}rr \subseteq O_C \cup O_P$

Preuve. Soit (X, Y) un concept du treillis de Galois associé à une relation binaire. Si (X, Y) est un élément inf-irréductible, et $(X, Y) \neq 1$, soit (X^*, Y^*) l'unique élément qui le couvre. Par définition $Y^* \subset Y$, donc le concept (X, Y) déclare forcément un élément : il existe $y \in F$, avec $y \in Y$ et $y \notin Y^*$. (X, Y) étant le sommet le plus élevé qui possède y, on a

$g(\{y\}) = Y$ (prop. 3.12). D'après la définition 3.19, (X, Y) est donc aussi un sommet de la sous-hiérarchie de Galois. □

Nous voyons au travers de la preuve que, de manière plus précise, les inf-irréductibles appartiennent forcément à O_P ($\mathcal{M} \subseteq O_P$) tandis que les sup-irréductibles appartiennent à O_C ($\mathcal{J} \subseteq O_C$).

Une question vient alors immédiatement à l'esprit, qui consiste à chercher à caractériser la sous-hiérarchie de Galois à l'aide de ces éléments irréductibles. La première réponse apportée est qu'il y a évidemment dans la sous-hiérarchie de Galois des éléments autres que les éléments irréductibles. Il suffit pour s'en convaincre de se reporter à l'exemple donné à l'occasion de la figure 3.7 où le concept $(\{3,4\}, \{a,c,f\})$, qui appartiendrait à la sous-hiérarchie de Galois (sous sa forme simplifiée$(\{\}, \{f\})$) n'est pas un sommet irréductible. Cependant cette première intuition est vraie pour certaines relations, déjà identifiées dans la littérature pour un autre propos.

Relation réduite

Définition 3.20 (Relation réduite [BM70]) *Soit une relation binaire $R \subset E \times F$. R est réduite si on ne peut y trouver :*

- *ni élément $x \in E$ tel qu'il existe une partie $X \subseteq E \setminus \{x\}$ avec $f(X) = f(\{x\})$,*
- *ni élément $y \in F$ tel qu'il existe une partie $Y \subseteq F \setminus \{y\}$ avec $g(Y) = g(\{y\})$.*

Visuellement, dans une relation réduite, aucune fonction caractéristique d'une ligne ($f(\{x\}$ pour $x \in E$) n'est l'intersection d'autres fonctions caractéristiques d'autres lignes ($f(X) = \cap_{a \in X} f(a)$). La même propriété est vérifiée par les colonnes. La relation R1 (figure 3.6) n'est pas réduite, puisque $g(\{f\}) = g(\{a, c\})$.

Propriété 3.16 (Relation réduite et sous-hiérarchie de Galois(1)) *Si une relation binaire $R \subseteq E \times F$ est réduite, la sous-hiérarchie de Galois associée à R a pour sommets les éléments irréductibles du treillis de Galois, ainsi que 0 s'il existe $x \in E$ avec $f(\{x\}) = F$ et 1 s'il existe $y \in F$ avec $g(\{y\}) = E$.*

Pour démontrer cette propriété nous utilisons la propriété (plus forte) suivante. Les relations binaires utilisées $R \subseteq E \times F$ sont supposées telles que tout élément de E (resp. de F) est en relation avec au moins un élément de F (resp. de E).

Propriété 3.17 (Relation réduite et sous-hiérarchie de Galois(2)) *Si une relation binaire R est réduite, $O_C = \mathcal{J} \cup \{0$ s'il existe $x \in E$ avec $f(\{x\}) = F\}$ et $O_P = \mathcal{M} \cup \{1$ s'il existe $y \in F$ avec $g(\{y\}) = E\}$.*

Preuve. Si R est réduite, il ne peut exister $e \in E$ et une partie $X \subseteq E \setminus \{e\}$ avec $f(X) = f(\{e\})$. Soit un élément de O_C, c'est-à-dire un concept de la forme $(g(f(\{e\})), f(\{e\}))$, avec $e \in E$. Soit cet élément de O_C est 0, s'il existe $x \in E$ avec $f(\{x\}) = F$. Soit ce n'est pas 0, et dans ce cas il ne peut être le supremum de plusieurs concepts (X_i, Y_i), $1 \leq i \leq n$, strictement inférieurs, sinon par définition $f(\{e\}) = \cap_{1 \leq i \leq n} Y_i = \cap_{1 \leq i \leq n} f(X_i)$, ce qui aboutirait à une contradiction en posant $X = \cup_{1 \leq i \leq n} X_i$. Un élément de O_C est donc forcément 0 ou un sup-irréductible : $O_C \subseteq \mathcal{J} \cup \{0$ s'il existe $x \in E$ avec $f(\{x\}) = F\}$. De manière similaire, on peut démontrer que les éléments de O_P sont des inf-irréductibles ou 1 : $O_P \subseteq \mathcal{M} \cup \{1$ s'il existe $y \in F$ avec $g(\{y\}) = E\}$. On conclut par la propriété 3.15. □

Les réciproques des deux propriétés précédentes sont fausses. En ce qui concerne la propriété 3.16, lorsque la sous-hiérarchie de Galois est réduite aux éléments irréductibles du treillis (plus 0 ou 1 éventuellement), rien ne garantit que la relation soit réduite. En

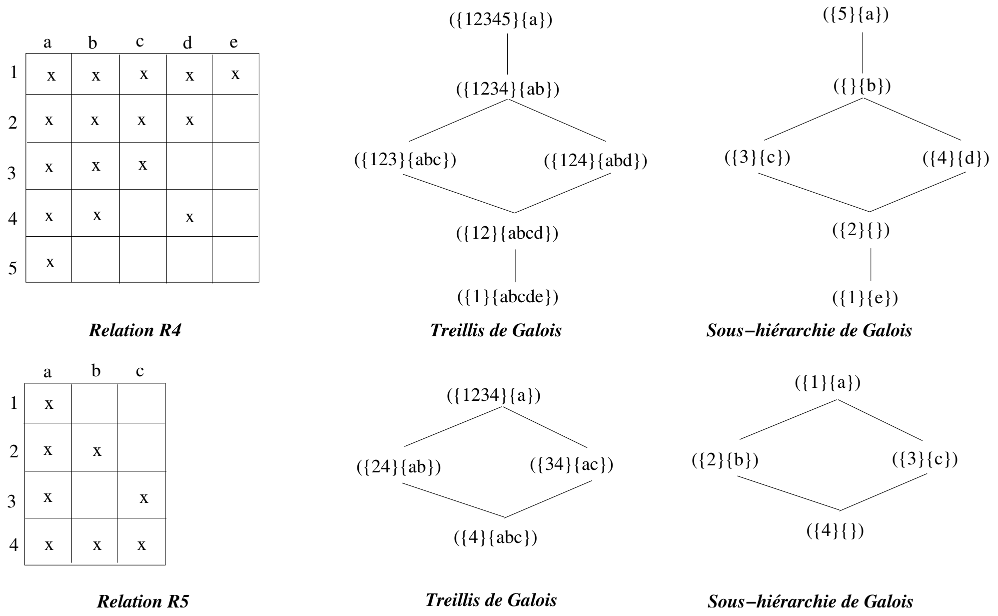


FIG. 3.11 – Présence des éléments 0 et 1 dans la sous-hiérarchie de Galois

effet un sup-irréductible peut être à la fois l'infimum de plusieurs classes (ne pas être un inf-irréductible) et déclarer une propriété (introduire un élément de F). Pour ce qui est de la réciproque de la propriété 3.17, deux propriétés y_1 et y_2 peuvent être déclarées par le même sommet, et donc $g(y_1) = g(y_2)$ contredit la définition de la réduction. On trouve une réciproque en restreignant encore la forme des concepts.

PROPRIÉTÉ 3.18 (RELATION RÉDUITE ET SOUS-HIÉRARCHIE DE GALOIS (3)) *Une relation $R \subseteq E \times F$ est réduite si et seulement si un concept ne définit pas plus d'un élément de E et ne déclare pas plus d'un élément de F et dans le treillis de Galois tout élément de O_C est un sup-irréductible (ou 0 s'il définit un élément de E) et tout élément de O_P est un inf-irréductible (ou 1 s'il déclare un élément de F).*

Dans les sous-hiérarchies de Galois des relations $R1$ et $R2$, les éléments minimum et maximum du treillis ont disparu. Ce n'est pas toujours le cas comme le mentionnent les propriétés précédentes et nous allons illustrer ce fait. La figure 3.11 présente quelques exemples montrant que les éléments 0 et 1 peuvent être présents, qu'ils soient irréductibles ou non, dans la sous-hiérarchie de Galois. Dans la relation $R4$ et le treillis $T(R4)$ associé, $0_{T(R4)}$ et $1_{T(R4)}$ sont des éléments irréductibles et seraient conservés dans la sous-hiérarchie de Galois rien qu'à ce titre. Dans la relation $R5$ et le treillis $T(R5)$, $0_{T(R5)}$ et $1_{T(R5)}$ ne sont irréductibles ni l'un ni l'autre, cependant ils appartiennent tous deux à la sous-hiérarchie de Galois : $0_{T(R5)} \in O_C$ car il existe une classe qui possède toutes les propriétés et $1_{T(R5)} \in O_P$ car il existe une propriété que possèdent toutes les classes.

On a obtenu ainsi, sous une autre formulation, ce que les auteurs de [GW99] utilisent comme définition alternative de la réduction de relation (au problème du 0 et du 1 près).

DÉFINITION 3.21 (RELATION CLARIFIÉE [GW99]) *Une relation clarifiée est une relation dans laquelle pour tous $a, b \in E$ (resp. $a, b \in F$), $f(\{a\}) = f(\{b\})$ (resp. $g(\{a\}) = g(\{b\})$) implique $a = b$.*

DÉFINITION 3.22 (RELATION W-RÉDUITE [GW99]) *Une relation W-réduite est une relation clarifiée et telle que dans le treillis de Galois associé tout objet-concept (élément de*

O_C) est un sup-irréductible et tout attribut-concept (élément de O_P) est un inf-irréductible.

Relation shg-réduite

Un intérêt de la réduction présentée ci-dessus, outre qu'elle nous a offert un cadre pour caractériser la sous-hiérarchie de Galois par les irréductibles, est que le treillis associé à la relation réduite reste isomorphe au treillis de Galois de la relation initiale, seuls les concepts (intension et extension) sont réduits. Dans l'exemple suivi (figure 3.6), l'élément f n'était ainsi pas utile pour la construction du concept $(\{3, 4\}, \{a, c, f\})$ et pouvait donc disparaître. Nous savons malheureusement que cette réduction ne préserve pas la sous-hiérarchie de Galois, puisque le retrait de l'élément f ci-dessus justement ferait disparaître un élément de O_P (et dans l'application aux hiérarchies de classes une classe importante, qui servait à déclarer au moins cette fameuse propriété f). Il est donc intéressant de voir comment cela s'étend à la sous-hiérarchie de Galois, c'est-à-dire si la relation peut être simplifiée tout en gardant une sous-hiérarchie de Galois isomorphe. Nous apportons une réponse à cette question avec la notion de réduction suivante qui réduit plus faiblement que la réduction de [BM70, GW99].

Définition 3.23 (relation shg-réduite) *Une relation binaire $R \subseteq E \times F$ est shg-réduite si :*
- *elle est clarifiée,*
- *il n'existe pas d'élément $y \in F$ qui soit déclaré par un sommet de $\mathcal{J} \setminus \mathcal{M}$,*
- *il n'existe pas d'élément $x \in E$ qui soit défini par un sommet de $\mathcal{M} \setminus \mathcal{J}$,*
- *un sommet qui n'est pas irréductible déclare un élément $y \in F$ ou (exclusivement) définit un élément $x \in E$.*

Dans l'exemple des figures 3.6 et 3.7, f est déclaré dans le treillis par un sommet qui n'est pas un sup-irréductible, donc la shg-réduction ne s'applique pas, f est conservée (on a vu qu'elle aurait été supprimée par la w-réduction). Dans l'exemple des figures 3.9 et 3.10, f est déclaré dans le treillis par un sommet qui est un inf-irréductible mais pas un sup-irréductible, donc f est également conservée (elle l'aurait été par la w-réduction).

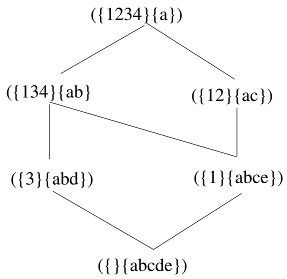
Nous donnons un autre exemple de réduction grâce à la figure 3.12. Dans ce dernier cas, nous constatons que la propriété e est déclarée par un sommet (le concept $(\{1\}\{abce\})$) qui, dans le treillis, est sup-irréductible, mais pas inf-irréductible. Un regard montre que e n'est pas nécessaire pour obtenir le sommet $(\{1\}\{e\})$ dans la sous-hiérarchie de Galois puisque ce sommet est créé par la présence de l'objet 1. e peut donc être supprimée dans la relation, la sous-hiérarchie de Galois obtenue après cette suppression restant isomorphe. De manière symétrique, 4, qui est définie par un sommet (le concept $(\{134\}\{ab\})$) qui est inf-irréductible, mais pas sup-irréductible, peut disparaître.

Cette réduction nous semble avoir au moins trois intérêts. D'une part, elle permet de mieux comprendre une structure « minimale » génératrice d'une sous-hiérarchie de Galois, et en particulier nous servira dans la section suivante à trouver, pour un ordre partiel quelconque $\mathcal{P}$, une relation binaire dont la sous-hiérarchie de Galois associée est isomorphe à $\mathcal{P}$.

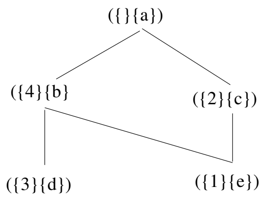
D'autre part toute réduction s'interprète de manière « conceptuelle » selon les applications traitées. Dans le cas de la construction de hiérarchies de classes, par exemple, la clarification de la relation permet de mettre en évidence des ensembles d'attributs déclarés conjointement qui n'ont besoin que d'un représentant pour la compréhension de la hiérarchie. Symétriquement, des classes définies ensemble peuvent être véritablement redondantes d'un point de vue génie logiciel. Une propriété telle que e accompagnerait-elle

	a	b	c	d	e
1	x	x	x		x
2	x		x		
3	x	x		x	
4	x	x			

relation R3



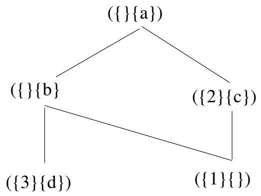
Treillis de Galois



Sous−hiérarchie de Galois

	a	b	c	d
1	x	x	x	
2	x		x	
3	x	x		x

relation R3 shg−réduite



Sous−hiérarchie de Galois réduite

FIG. 3.12 –

« inutilement » la définition de la classe 1 qu'il suffit de comprendre comme la conjonction des classes 4 et 2 ? La classe 4 serait-elle inutile car les propriétés a et b, communes à 1 et 3 forment déjà un concept (donc une classe) ? Ces interprétations sont bien évidemment à prendre avec délicatesse suivant le contexte particulier dans lequel on travaille, mais montrent que la réduction n'a pas qu'un intérêt théorique.

Enfin la shg-réduction peut aider dans la recherche d'algorithmes efficaces pour la construction et le codage de la sous-hiérarchie de Galois, tout comme la réduction l'est pour le treillis [GW99]. En effet, un algorithme comme CERES procède par un calcul conjoint des concepts et de l'ordre qui les relie. Dans ce calcul, l'un des éléments de la complexité est le nombre de classes et le nombre de propriétés manipulées. On imagine aisément qu'une réduction diminuerait donc la complexité de l'algorithme CERES. Reste à calculer cette réduction à un coût (au moins en pratique) inférieur au gain obtenu dans CERES.

Vers le calcul de la shg-réduction

Une méthode peut être construite par extension de celle qui a été proposée dans [GW99] pour la réduction d'un contexte clarifié. Cette méthode ne construit bien entendu pas le treillis de Galois mais est capable de déterminer si un élément de O_C ou de O_P est irréductible.

DÉFINITION 3.24 (RELATIONS FLÈCHES [GW99]) *Soit une relation binaire clarifiée $R \subseteq E \times F$, $o \in E$ et $p \in F$. Trois relations flèches peuvent être définies pour f et g applications caractéristiques de R :*

- $o \leftarrow p$ *ssi* $(o,p) \notin R$ *et* $f(o) \subset f(a)$ *implique* $(a,p) \in R$
- $o \rightarrow p$ *ssi* $(o,p) \notin R$ *et* $g(p) \subset g(q)$ *implique* $(o,q) \in R$
- $o \leftrightarrow p$ *ssi* $o \leftarrow p$ *et* $o \rightarrow p$

$o \leftarrow p$ lorsque o est un objet de plus grande intension qui n'a pas la propriété p, tandis que $o \rightarrow p$ si p est une propriété de plus grande extension qui n'est pas possédée par o.

	a	b	c	d	e
1	x	x	x		x
2	x		x		
3	x	x		x	
4	x	x			

relation R3

	a	b	c	d	e
1	x	x	x	$\leftrightarrow$	x
2	x	$\leftrightarrow$	x		$\leftarrow$
3	x	x	$\leftrightarrow$	x	$\leftarrow$
4	x	x	$\rightarrow$	$\rightarrow$	

relation R3 flèches

	a	b	c	d
1	x	x	x	
2	x		x	
3	x	x		x

relation R3 shg−réduite

FIG. 3.13 – Relations flèches pour $R3$

La figure 3.13 expose les relations flèches correspondant à la relation $R3$. Nous détaillons ci-dessous quelques éléments du calcul.

- $1 \leftarrow d$: $f(\{1\}) = \{abce\}$ et $\{abce\}$ est la plus grande intension ne contenant pas d.
- $1 \rightarrow d$: $g(\{d\}) = \{3\}$ qui est la plus grande extension ne contenant pas 1.
- $2 \leftarrow b$: $f(\{2\}) = \{ac\}$, la seule autre intension contenant a et c est $f(\{1\})$ qui contient b.
- On n'a pas $2 \leftarrow d$: en effet $f(1) = \{abce\}$ est une intension plus grande que $f(\{2\}) = \{ac\}$ et elle ne contient pas d.
- $2 \leftarrow e$: $f(\{2\}) = \{ac\}$, la seule autre intension contenant a et c est $f(\{1\})$ qui contient e.

La propriété fondamentale de ces relations flèches est de permettre d'identifier directement dans la relation les éléments irréductibles.

PROPRIÉTÉ 3.19 (IRRÉDUCTIBLES ET RELATIONS FLÈCHES [GW99]) *Soit une relation binaire clarifiée (finie) $R \subseteq E \times F$, et soient $o \in E$ et $p \in F$.*

- *le concept $(g(f(\{o\})), f(\{o\})) \in O_C$ est sup-irréductible si et seulement si $\exists p \in F$ avec $o \leftrightarrow p$*
- *le concept $(g(\{p\}), f(g(\{p\}))) \in O_P$ est inf-irréductible si et seulement si $\exists o \in E$ avec $o \leftrightarrow p$*

L'utilisation de cette propriété pour obtenir une relation w-réduite est directe, puisqu'il suffit d'enlever toutes les lignes et toutes les colonnes qui ne contiennent pas de relation $\leftrightarrow$.

Pour obtenir une relation shg-réduite, il faut être un peu plus subtil. Le premier point à considérer est qu'il ne faut pas enlever ni 0_T s'il définit une classe (il existe alors une ligne "pleine"), ni 1_T s'il déclare une propriété (il existe alors une colonne "pleine").

Le deuxième aspect relève de la définition 3.23. Une ligne (c'est-à-dire un objet o de E) ne doit être enlevée que si le concept correspondant est un élément de $\mathcal{M} \setminus \mathcal{J}$, c'est-à-dire un inf-irréductible qui n'est pas sup-irréductible. Dans le cas contraire, elle doit être gardée car elle est génératrice d'un élément de la sous-hiérarchie de Galois. Le concept $C_o = (g(f(\{o\})), f(\{o\}))$, engendré à partir de o, n'est pas sup-irréductible si la ligne ne contient pas de relation $\leftrightarrow$. Pour déterminer si C_o est par ailleurs un inf-irréductible, il suffit de vérifier s'il existe une propriété $p \in f(\{o\})$ qui est à la fois déclarée par C_o ($g(\{p\}) = g(f(\{o\}))$) et munie d'une relation $\leftrightarrow$ dans sa colonne. Dans l'exemple de la relation $R3$, la ligne de l'objet 4 n'a pas de relation $\leftrightarrow$, ce qui la rend potentiellement effaçable. Le concept engendré par 4, c'est-à-dire $(\{134\}, \{ab\})$, n'est en effet pas sup-irréductible. Reste à constater s'il est ou non inf-irréductible. Calculant le concept engendré par b, on trouve également qu'il s'agit de $(\{134\}, \{ab\})$. Or la relation $\leftrightarrow$ présente dans la

	a	b	c	d	e	f
1	x	x				
2			x			
3	x		x	x		x
4	x		x		x	x
5	x					

Relation R1

	a	b	c	d	e	f
1	x	x	$\leftrightarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$
2	$\leftrightarrow$		x			
3	x	$\leftrightarrow$	x	x	$\leftrightarrow$	x
4	x	$\leftrightarrow$	x	$\leftrightarrow$	x	x
5	x	$\rightarrow$	$\rightarrow$			

Relation R1 flèche

	a	b	c	d	e	f
1	x	x				
2			x			
3	x		x	x		x
4	x		x		x	x

Relation R1 réduite

FIG. 3.14 – Relations flèches pour $R1$

colonne de b signale que b est inf-irréductible. La ligne 4 peut être suprimée de la relation.

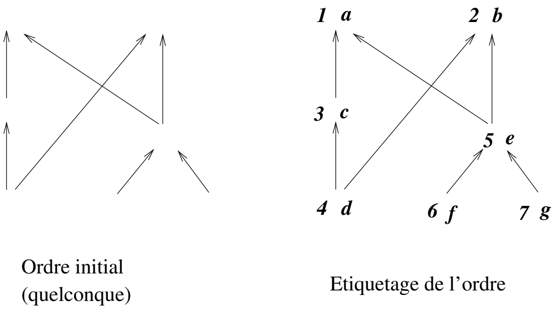
De manière symétrique, une colonne (c'est-à-dire un objet p de F) ne doit être enlevée que si le concept correspondant est un élément de $\mathcal{J} \setminus \mathcal{M}$, c'est-à-dire un sup-irréductible qui n'est pas inf-irréductible. Le concept $C_p = (g(\{p\})), f(g(\{p\}))$, engendré à partir de p, n'est pas inf-irréductible si la colonne ne contient pas de relation $\leftrightarrow$. Pour déterminer si C_p est par ailleurs un sup-irréductible, il suffit de vérifier s'il existe un objet $o \in g(\{p\})$ qui est à la fois défini par C_p $(f(\{o\}) = f(g(\{p\})))$ et muni d'une relation $\leftrightarrow$ dans sa ligne. Dans l'exemple de la relation $R3$, la colonne e peut être supprimée par ce procédé.

Dans la relation $R1$, on a vu que f serait effacé par la w-réduction. Dans le calcul proposé ci-dessus pour la shg-réduction, le concept engendré par f n'est pas inf-irréductible car il n'y a pas de relation $\leftrightarrow$ dans la colonne de f. Aucun des objets de son extension $g(\{f\})$ n'engendre le même concept que f c'est-à-dire $(\{34\}, \{acf\})$, or les sup-irréductibles sont forcément engendrés par des objets. Donc le concept engendré par f n'est pas sup-irréductible et f ne doit pas être supprimée. En ce qui concerne la ligne 5, elle ne contient pas non plus de relation $\leftrightarrow$, et le concept engendré par 5, c'est-à-dire $(\{1345\}, \{a\})$ n'est donc pas sup-irréductible. Cependant, ce même concept est engendré par a et la colonne de a contient une relation $\leftrightarrow$, donc est inf-irréductible. La ligne 5 peut être supprimée pour la shg-réduction, comme elle l'est également par la w-réduction.

Une évaluation très grossière de la complexité d'un algorithme calculant la réduction en suivant directement les définitions précédentes serait la suivante. Soit R^- la relation complémentaire de R, c'est-à-dire $E \times F \setminus R$.

- Le calcul des relations flèches serait la somme des calculs des relations $\leftarrow$, $\rightarrow$ et $\leftrightarrow$. Examinons le calcul de $\leftarrow$. Pour chaque couple (o, p) inexistante dans R (il y en a $|R^-|$), il s'agit de vérifier que si la ligne $f(o)$ est incluse dans une autre ligne $f(i)$ alors $p \in f(i)$. Le calcul des relations $\leftarrow$ s'évalue ainsi à $\mathcal{O}(|R^-| * |F|^2)$. Similairement celui des relations $\rightarrow$ s'évaluerait à $\mathcal{O}(|R^-| * |E|^2)$.
- La suppression des lignes (au plus |E|) ne contenant pas de $\leftrightarrow$ demande pour chacune de ces lignes d'examiner les propriétés p y apparaissant (au plus |F|), et de vérifier si $g(p) = g(f(o))$ (de taille $|E|$) et que la colonne p ne contient pas de relation $\rightarrow$. Ceci peut être fait en $\mathcal{O}(|E|^2 * |F|)$. Symétriquement la suppression des colonnes se ferait en $\mathcal{O}(|E| * |F|^2)$.

Ce qui précède n'a pas de prétention algorithmique et permet seulement de vérifier que l'on reste dans des complexités de l'ordre de celles de CERES ou acceptables (on a un facteur supplémentaire dans l'évaluation des relations flèches), mais la question de trouver un algorithme efficace de calcul de la shg-réduction, en tout cas suffisamment efficace en pratique pour qu'il puisse diminuer le coût total de la construction de la sous-hiérarchie de Galois reste ouverte.



	a	b	c	d	e	f	g
1	x						
2		x					
3	x		x				
4	x		x	x			
5	x	x			x		
6	x	x			x	x	
7	x	x			x		x

Relation construite

FIG. 3.15 – Construction d'une relation binaire dont la sous-hiérarchie de Galois est isomorphe à un ordre donné

3.6.4 Ordre partiel et Sous-hiérarchie de Galois

De même qu'à tout treillis on peut faire correspondre une infinité de relations binaires dont il est treillis de Galois, à tout ordre partiel P on peut associer une relation binaire dont elle est sous-hiérarchie de Galois. Sans chercher à trouver une relation de plus petite taille réalisant ce codage de l'ordre par une relation binaire, on peut se convaincre qu'il en existe une en associant bijectivement à chaque sommet s de l'ordre une étiquette dans un ensemble E (par une bijection b_E) et une étiquette dans un ensemble F (par une bijection b_F). Une relation R est ensuite établie comme une partie de $E \times F$ en admettant $(o, p) \in R$ si $b_E^{-1}(o) \leq_P b_E^{-1}(p)$ (voir l'exemple figure 3.15). Elle peut être shg-réduite au besoin.

Si l'ordre partiel est un treillis, il est facile de minimiser la taille de la relation dont le treillis est sous-hiérarchie de Galois en utilisant directement les aspects structurels montrés auparavant. En effet dans le cas d'un treillis, il suffit d'étiqueter :

- chaque inf-irréductible (et le maximum s'il existe) par une propriété (dans F),
- chaque sup-irréductible (et le minimum) par un objet (dans E),
- un sommet non irréductible au choix par un objet ou (exclusif) une propriété.

Le point à retenir ici n'est pas d'approfondir la recherche d'un codage minimal, mais de montrer que le modèle n'est pas limitatif, puisqu'il n'interdit aucune sorte d'ordre partiel, ni n'en promeut non plus aucun. C'est très important dans le contexte des hiérarchies de classes. De plus nous disposons ainsi d'un procédé de génération d'exemples très utiles pour confronter des hypothèses ou des algorithmes à des cas standards avant de passer à une preuve formelle.

3.6.5 Pour une typologie des relations

Nous avons vu quelques exemples ci-dessus où certaines particularités structurelles de la relation ont une résonance « conceptuelle » au sens où elles peuvent s'interpréter dans l'application. Si l'on s'intéresse à construire des hiérarchies de classes, le fait d'avoir deux lignes[1] égales révèle que deux classes ont exactement même définition, ce qui, en génie logiciel, est plutôt signe d'une mauvaise conception puisqu'on aurait développé deux fois le même code. Si l'on construit des hiérarchies d'interfaces, l'interprétation est toute autre, ce qui se produit est au contraire intéressant car cela montre l'intérêt de coder une interface dont on sait qu'elle sera implémentée par ces deux classes.

D'autres exemples peuvent être donnés.

Certaines approches de la construction automatique de hiérarchies de classes [LBSL90, LBSL91, Moo96] font l'hypothèse qu'une classe instanciable est toujours feuille de la hiérar-

[1] Par abus de langage nous parlons de ligne pour désigner $f(\{e\})$, où $e \in E$.

chie de classes. Cela peut être érigé en critère de qualité de la hiérarchie. Ce qui est notable, c'est que lorsqu'une hiérarchie de classes est reconstruite en partant de ses classes instanciables (qui forment alors l'ensemble E), si l'on veut préserver leur position de feuilles, alors aucune ligne ne doit être incluse dans une autre. Si cela n'est pas vérifié naturellement, on peut l'obtenir en ajoutant une propriété spécifique à chaque classe instanciable. La complexité de la construction est alors naturellement réduite car il n'est plus nécessaire que de construire et d'organiser les concepts de O_P. C'est l'élément qui explique la différence de complexité théorique entre AISGOOD [GC00] (qui fait l'hypothèse d'une propriété spécifique à chaque classe d'origine, résolvant ainsi un problème moins complexe) et ARES [DDHL94a] (qui ne fait pas cette hypothèse). L'approche duale (une colonne n'est jamais strictement incluse dans une autre colonne) est peu réaliste dans notre contexte et signifierait que les propriétés seront toutes déclarées par une classe différente, ces classes de déclaration formant les éléments maximum de la hiérarchie d'arrivée.

On a vu précédemment les propriétés de la relation qui feraient que la sous-hiérarchie de Galois est en fait le treillis de Galois, on pourrait chercher à énoncer celles qui font que la sous-hiérarchie de Galois est, dans un ordre de « complexité » croissante un arbre, un ordre de dimension 2 (intersection de deux ordres totaux sur ses éléments), et ainsi de suite jusqu'au treillis de Galois. Ces énoncés existent vraisemblablement déjà, et leur interprétation génèrerait un autre éclairage sur la structure des hiérarchies de classes.

Ces remarques mènent à penser qu'il serait vraiment pertinent d'envisager une véritable étude typologique des relations, à des fins algorithmiques tout autant que d'interprétation.

3.7 Une sous-hiérarchie de Galois généralisée

Nous nous sommes restreints, dans les deux sections qui précèdent, aux structures associées à une relation binaire. La notion de sous-hiérarchie de Galois peut sans doute se généraliser et se concevoir dans un cadre plus large. Revenons au départ sur sa définition. La sous-hiérarchie de Galois se définit constructivement comme la fermeture des atomes des deux treillis booléens 2^E et 2^F.

On peut ainsi envisager de généraliser la définition de la sous-hiérarchie de Galois de différentes manières, telles que la fermeture des atomes dans un treillis booléen, plus généralement la fermeture des atomes dans un treillis atomique, ou pourquoi pas la fermeture des éléments irréductibles dans un treillis quelconque.

Une application qui nous intéresse particulièrement serait de disposer d'une définition qui permette de simplifier les treillis de Galois tels que ceux définis dans [LS98] et dont [Mic00] montre l'intérêt dans le cadre de diagrammes statiques UML. Dans ces travaux, le problème traité consiste à généraliser des graphes étiquetés (dont les diagrammes de classes et d'instances UML sont des exemples). Deux opérations d'infimum et de supremum sur ces graphes étiquetés induisent une correspondance et par suite un treillis de Galois. Une approche, évoquée par M. Liquière et qui mériterait l'approfondissement, pourrait être de déterminer l'équivalent d'une base de graphes (donnés par les inf-irréductibles et qui seraient en quelque sorte les "attributs" des exemples décrits) et de construire la fermeture de chacun des éléments de cette base.

3.8 Conclusion

Dans ce chapitre, nous avons cherché à placer la notion de sous-hiérarchie de Galois dans son contexte théorique. L'objectif était essentiellement de poser le cadre d'une étude structurelle, car la notion de sous-hiérarchie de Galois n'a pas, à notre connaissance, été étudiée en détail, étant le plus souvent succinctement décrite en toile de fond des applications qui l'utilisent. Cependant, les treillis de Galois ont été abondamment étudiés, et on peut espérer que de mettre en lumière la sous-hiérarchie de Galois permettra d'y rattacher des résultats déjà existants ou d'en susciter de nouveaux. Déjà on trouve des propositions allant dans cette direction [BS02].

Chapitre 4

Synthèse des recherches effectuées

Préambule

Ce chapitre propose une description des recherches effectuées dans un cadre unifié. Nous ne chercherons pas à reprendre en détails tous les travaux réalisés. Plutôt, nous décrirons la problématique générale et les résultats obtenus. Nous chercherons également à rapprocher les travaux, par exemple les différents algorithmes proposés, ou nous évoquerons comment certaines propositions en généralisent d'autres.

4.1 Introduction

Dans ce chapitre, nous faisons une synthèse des recherches effectuées. Nous commençons par présenter un exemple concret de reconstruction de hiérarchie de classes (Partie 4.2). Nous illustrons ainsi le problème de la description des classes (chapitre 2) et l'application pratique des treillis de Galois (chapitre 3). Nous présentons ensuite les études algorithmiques réalisées ou en cours (partie 4.3). Nous décrivons une application pratique qui a été approfondie dans le cadre de la thèse d'Hervé Leblanc et qui consiste à produire automatiquement un ensemble structuré d'interfaces associées à une hiérarchie de classes Java (partie 4.4). Dans la partie 4.5, nous évoquons le problème lié à l'évolution d'une hiérarchie en cours d'exploitation, en particulier instanciée. Nous présentons également des travaux concernant l'évaluation de la généralisation et de la spécialisation dans une hiérarchie de classes par des métriques (partie 4.6). Nous ne sortons pas du contexte car les métriques sont basées sur une comparaison de la hiérarchie de classes et de sa reconstruction grâce à notre méthode. Enfin nous étudions actuellement la construction simultanée de hiérarchies de spécialisation sur les classes, les attributs, les méthodes et les associations (dans le cadre des diagrammes UML). Cette recherche très récente est décrite dans la partie 4.7.

4.2 Souvent Flot varie (un exemple de reconstruction)

Nous étudions quelques variations de nos propositions théoriques sur un exemple pratique. Dans cette application, E est l'ensemble des classes ($E = \mathcal{C}$), F est l'ensemble des référents ($F = \mathcal{R}_{ef}$), f et g sont déterminées par $Intension_r$ ($f = Intension_r$ et $\forall r \in \mathcal{R}_{ef}, g(r) = \{C \in \mathcal{C} / r \in Intension_r(C)\}$),

La hiérarchie choisie (déjà étudiée de manière plus schématique dans [DHL97b, HDL00, Leb00]) est une version simplifiée de la hiérarchie des flots du langage C++ telle qu'elle apparaît dans la norme [sI98]. Les paramètres[1] des classes génériques et des méthodes ont été omis et les préfixes `basic_` ont été supprimés. Nous envisageons (Figure 4.1) une implémentation imaginaire[2] de cette hiérarchie dans laquelle les méthodes *str* et *close* ont un corps identique, tandis que les méthodes *open* ont un code qui ne varie que sur le mode d'ouverture (obligatoirement in pour $ifstream$, *out* pour $ofstream$ mais indifféremment in, *out* ou in/out pour $fstream$). Ce mode est passé en paramètre dans tous les cas, car C++ n'admet pas de variation dans les listes de paramètres lors de la redéfinition des méthodes (sinon on est dans un cas de surcharge dont la résolution est statique).

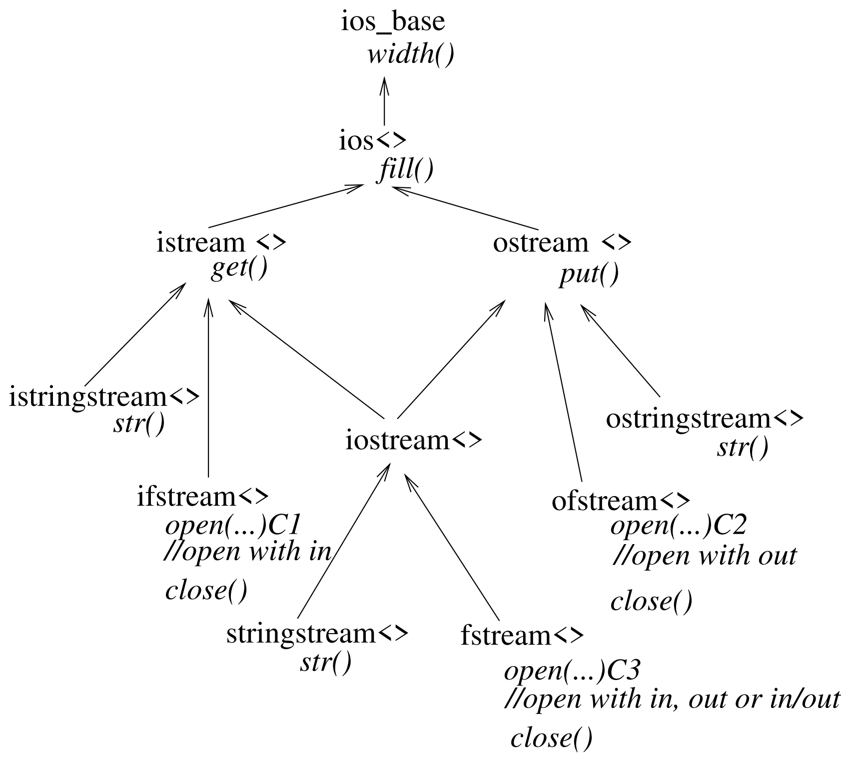


FIG. 4.1 – Extrait des classes flot de C++

Cette hiérarchie doit être exprimée dans notre langage de description en termes de référents et de propriétés génériques. Nous envisageons trois interprétations possibles, dictées par des heuristiques ou par un concepteur. Elles sont bien entendu discutables, mais l'important est de se focaliser sur le procédé employé pour reconstruire la hiérarchie à partir de ces interprétations. Nous détaillons avec plus de précisions la première, les autres suivant le même principe.

Interprétation 1 La première (Figure 4.2) consiste à considérer que la variation sur les méthodes *open* n'est pas suffisante pour considérer qu'elles sont différentes. Après tout, le programmeur pourrait n'en avoir qu'une à sa disposition et à lui de se débrouiller pour passer le paramètre correspondant au type de fichier (par ex. in pour $ifstream$). Pour gérer d'éventuelles erreurs, on peut par exemple vérifier de manière générale (indépendamment

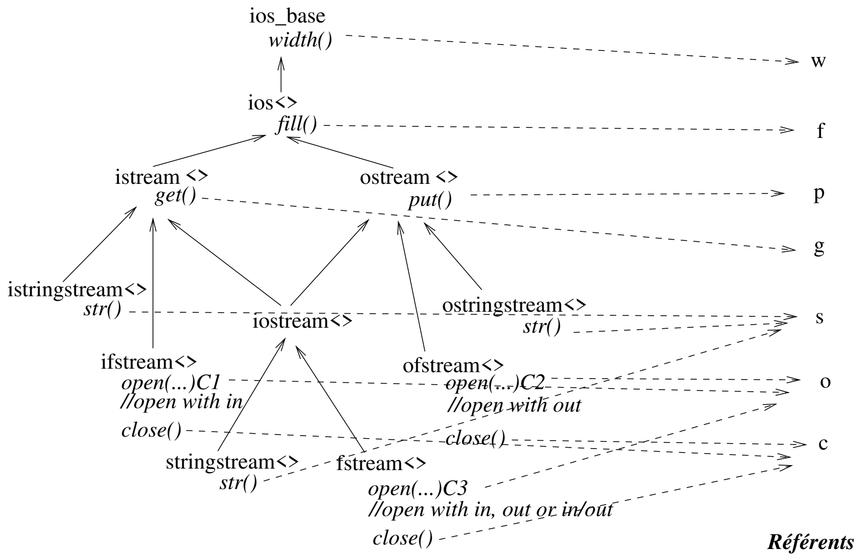
[1] Type de caractères qui peut être instancié par `char` correspondant généralement aux caractères ASCII ou `wchar_t` pour un ensemble de caractères plus large comme l'UNICODE. L'implémentation des caractères n'est pas parfaitement standardisée [Str98].

[2] Inspirée par l'implémentation actuelle de gnu 2.96 qui ne suit malheureusement pas la norme pour les flots.

du type de fichier) que le mode d'ouverture correspond au mode obligatoire qui pourrait être stocké par ailleurs. A droite de la figure, on voit les référents associés à cette première interprétation. Les trois référents s, o et c représentent respectivement trois co-référents chacun. Le concepteur peut également considérer qu'il n'y a pas lieu de regrouper plusieurs référents dans une propriété générique, ces dernières sont donc réduites à un seul élément. L'application $Intension$ est construite avec les hypothèses de la section 2.5 que nous rappelons. $r \in Intension(C)$ si l'une des conditions suivantes est vérifiée :

- C déclare un co-référent de r,
- C hérite un co-référent de r non masqué par un autre co-référent de la même propriété générique,
- r généralise un co-référent que C déclare ou hérite sans qu'il soit masqué.

Dans notre première interprétation, il n'y a pas de cas de masquage et le dernier point ne s'applique pas car les propriétés génériques sont réduites à un référent. Au bas de la figure, nous avons représenté la relation binaire que l'on peut tirer des intensions des classes : $(C, r) \in$ *a-le-référent* si $r \in Intension(C)$.



a–le–référent	w	f	g	p	s	c	o
ios_base	x						
ios<>	x	x					
istream<>	x	x	x				
istringstream<>	x	x	x		x		
ifstream<>	x	x	x			x	x
ostream<>	x	x		x			
ostringstream<>	x	x		x	x		
ofstream<>	x	x		x		x	x
iostream<>	x	x	x	x			
stringstream<>	x	x	x	x	x		
fstream<>	x	x	x	x		x	*x*

FIG. 4.2 – Interprétation 1

La figure 4.3 présente la sous-hiérarchie de Galois associée à la relation binaire de la figure 4.2. Les arcs ont tous la même signification, mais certains apparaissent en pointillé pour mieux mettre en évidence la structure obtenue qui correspond à un losange central (flots organisés selon la direction des données) dupliqué selon deux points de vue différents : les entrées-sorties en mémoire ($string$) et les entrées-sorties fichier ($file$).

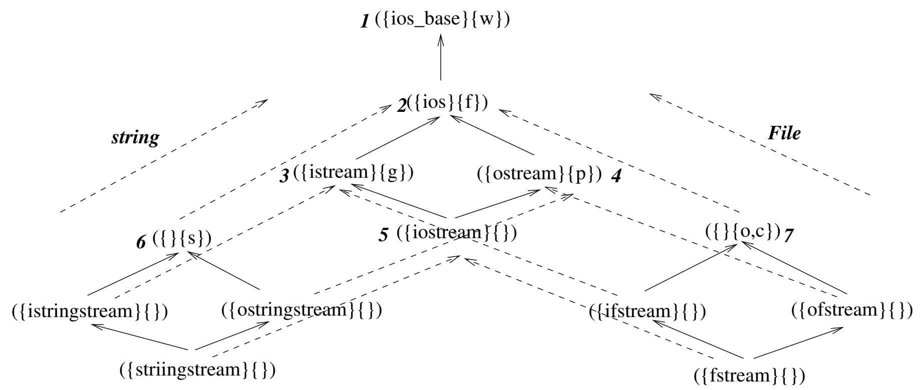


FIG. 4.3 – Sous-hiérarchie de Galois pour l'interprétation 1

La figure 4.4 montre le résultat de l'interprétation des concepts de la sous-hiérarchie de Galois de la figure 4.3. Les noms des nouvelles classes sont bien entendu inventés mais un lexique ou une construction à partir des noms des référents (comme le propose H. Leblanc pour des noms d'interfaces [Leb00]) pourrait permettre de les proposer. Les attributs et méthodes de la hiérarchie résultat sont obtenus en utilisant les référents. Dans ce résultat il n'y a par construction aucune duplication de propriété, et la factorisation a apporté deux nouvelles classes qui représentent les concepts généraux de flot-mémoire ($stringstream\_base$) ou de fichier ($fstream\_base$) dont on peut dire qu'ils faisaient défaut dans la hiérarchie initiale. Des liens de spécialisation/généralisation (par exemple entre $fstream$ et $ifstream$) sont apparus, qui peuvent être intéressants du point de vue de la sémantique du domaine représenté, si l'on considère qu'un fichier en entrée/sortie est en même temps un fichier en entrée ainsi qu'un fichier en sortie ce qui est correct du point de vue de la substituabilité. La référence [Str98] laisse planer une certaine ambiguïté sur ce point en n'indiquant que le cas par défaut : « a fstream ... by default can be both read from and written to ».

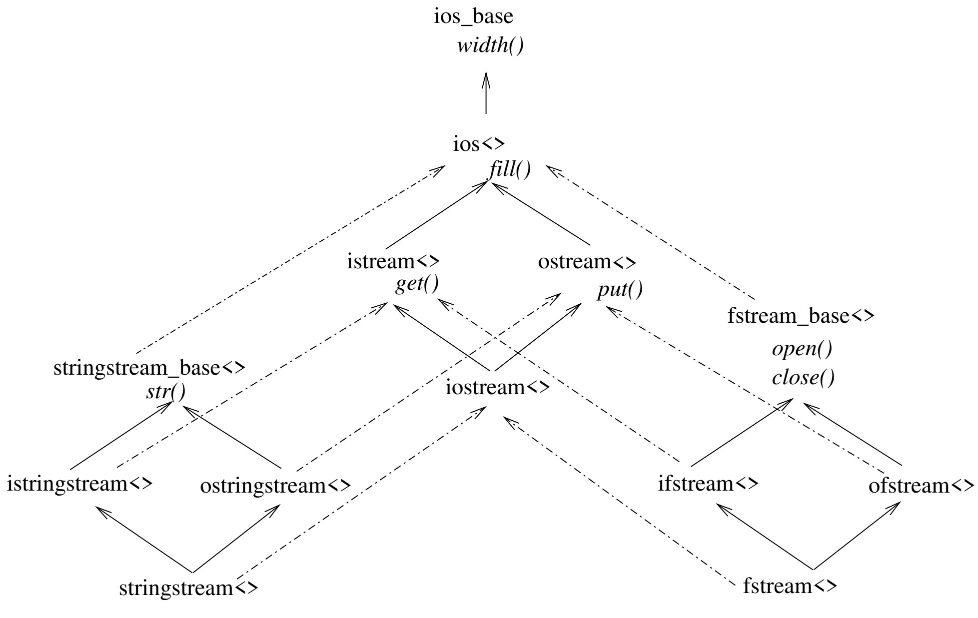
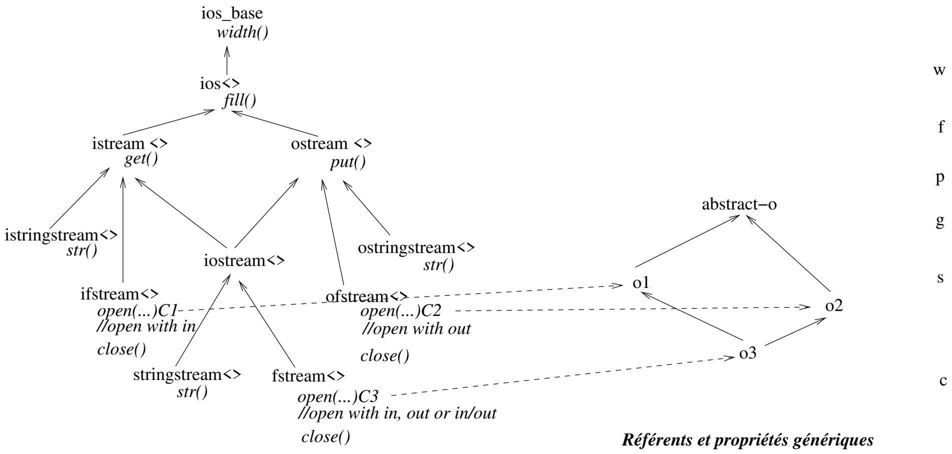


FIG. 4.4 – Interprétation 1 - résultat de la reconstruction

Interprétation 2 Dans la deuxième interprétation (Figure 4.5), on considère que les méthodes *open* doivent être distinguées, donc nous exprimons ce fait par la présence de plusieurs référents, un par co-référent de *open*. Ces référents appartiennent à une propriété générique qui n'est cette fois pas réduite à un singleton et un ordre doit être proposé pour ordonner partiellement ces éléments. Un concepteur peut considérer qu'ouvrir dans tous les modes possibles est une spécialisation de l'ouverture dans un mode particulier. Une heuristique usuelle peut également amener l'ajout d'une forme abstraite de la méthode *open*. Le code commun devrait[3] être délégué dans une méthode appelable dans les méthodes *open* et que nous ne représentons pas ici pour simplifier, car elle peut appartenir à une autre classe. Tout ceci s'exprime par la propriété générique de la droite de la figure. Les croix en caractères gras sur la figure proviennent de l'application de la règle selon laquelle un référent est dans l'intension d'une classe si l'une de ses spécialisations s'y trouve.



a−le−référent	w	f	g	p	s	c	o1	o2	o3	abstract−o
ios_base	x									
ios<>	x	x								
istream<>	x	x	x							
istringstream<>	x	x	x		x					
ifstream<>	x	x	x			x	x			**x**
ostream<>	x	x		x						
ostringstream<>	x	x		x	x					
ofstream<>	x	x		x		x		x		**x**
iostream<>	x	x	x	x						
stringstream<>	x	x	x	x	x					
fstream<>	x	x	x	x		x	**x**	**x**	x	**x**

FIG. 4.5 – Interprétation 2

Nous ne montrons pas la sous-hiérarchie de Galois mais seulement (Figure 4.6) la nouvelle hiérarchie qui est tirée de cette structure. Elle possède la même forme que la précédente, mais les méthodes *open* sont réparties en respectant la spécialisation.

Interprétation 3 Cette dernière interprétation est une variante de la précédente où le concepteur considère au contraire qu'ouvrir seulement en lecture (in) ou seulement en écriture (out) sont des spécialisations de l'ouverture dans tous les modes possibles. Les figures 4.7 et 4.8 montrent les schémas associés à cette construction. Il est intéressant de remarquer que ce résultat (pour la partie concernant les fichiers) se rapproche de la hiérarchie des flots telle qu'elle est implémentée dans la version gnu 2.96. Dans cette interprétation, $open()C3$ n'est pas appelable sur $ifstream$ car elle y est spécialisée, ce qui est cohérent

[3] Si ce n'est pas déjà le cas.

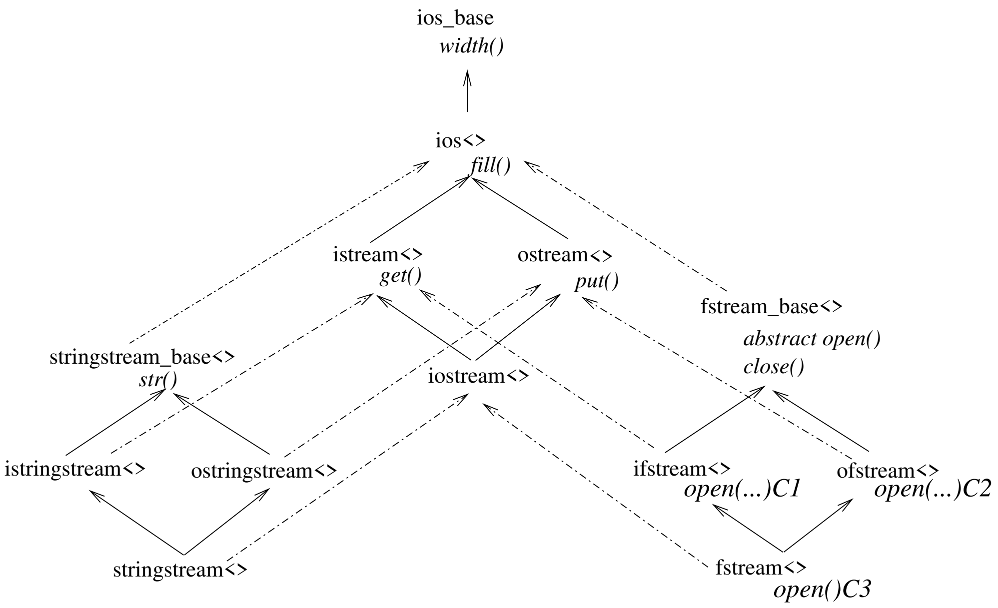


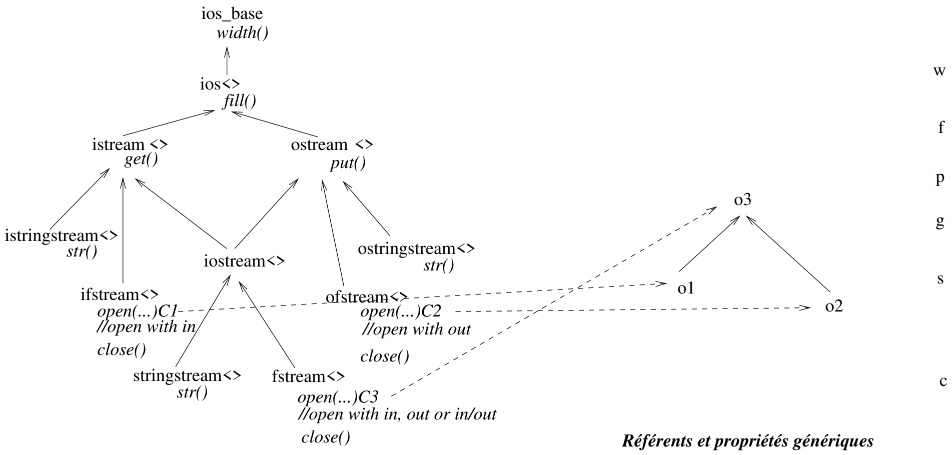
Fig. 4.6 – Interprétation 2 - résultat de la reconstruction

avec l'interprétation faite. On modélise cette fois en supposant que les trois sortes de fichiers forment une partition, ce qui est un point de vue qui diffère des interprétations 1 et 2 qui considéraient que les $fstream$ étaient l'intersection des $ifstream$ et des $ofstream$. Peut-être peut-on dire que cette troisième modélisation se base sur l'idée qu'un fichier de type $fstream$ est un fichier qui peut être exclusivement en entrée ou en sortie, ce n'est donc pas toujours simultanément un $ifstream$ et un $ofstream$. Dans la hiérarchie résultat, $fstream$ n'est plus sous-classe de $ifstream$ et $ofstream$, mais il reste le concept général de fichier avec $fstream\_base$.

Conclusion et état des travaux de validation de la reconstruction D'autres interprétations que nous ne développerons pas ici pourraient considérer d'autres relations entre les référents de *open*. Nous avons illustré le procédé technique de reconstruction et montré l'influence des choix initiaux de description de la hiérarchie sur le résultat. Mais dans tous les cas, le résultat final est cohérent avec les choix initiaux. Dans notre modélisation, la hiérarchie résultat est en effet une factorisation maximale plus spécifique (FMS) et la plus compacte possible dont nous rappelons quelques propriétés remarquables :

- un référent n'apparaît qu'une fois dans la hiérarchie (factorisation maximale des caractéristiques) ;
- les liens de spécialisation/généralisation entre les classes correspondent aux relations d'inclusion entre intensions ;
- l'ordre dans les propriétés génériques est en accord avec l'ordre de spécialisation/généralisation ;
- ceci est obtenu avec un nombre de classes minimal.

On voit aussi que les principaux points de frottement sont à l'initialisation (pour déterminer les référents et les propriétés génériques) ainsi qu'à l'issue de la reconstruction (lorsque les référents sont utilisés pour les classes du résultat). Dans le deuxième cas, en effet, si plusieurs co-référents de formes syntaxiques différentes ont été abstraits par un même référent, il faudra choisir ou composer une forme syntaxique correcte pour le langage cible. De même, dans le cas des méthodes, les règles de redéfinition du langage cible



a–le–référent	w	f	g	p	s	c	o1	o2	o3
ios_base	x								
ios<>	x	x							
istream<>	x	x	x						
istringstream<>	x	x	x		x				
ifstream<>	x	x	x			x	x		**x**
ostream<>	x	x		x					
ostringstream<>	x	x		x	x				
ofstream<>	x	x		x		x		x	**x**
iostream<>	x	x	x	x					
stringstream<>	x	x	x	x	x				
fstream<>	x	x	x	x		x			x

FIG. 4.7 – Interprétation 3

devront être respectées dans les formes syntaxiques finales. Ce peut être assez délicat à faire automatiquement. Un aspect particulièrement sensible concerne le traitement de probables conflits.

La méthode, sous la forme actuelle que nous venons d'illustrer, est en passe d'être testée sur des applications réelles (autres que des cas d'école). En réalité, des hiérarchies réelles (API Java, Pétra [dt97b, dt97a]) ont déjà été reconstruites par cette méthode, mais nous n'avons pas utilisé le résultat de la reconstruction directement (voir la partie 4.6 sur les métriques). Les résultats de la reconstruction ont été étudiés pour des hiérarchies de petite taille, telle que la hiérarchie des flots C++.

4.3 Etudes algorithmiques

La sous-hiérarchie de Galois étant beaucoup moins complexe que le treillis de Galois, il est plus intéressant de concevoir des algorithmes spécifiques que de construire le treillis puis de le réduire comme proposé dans [YLB97]. Une autre dimension que nous avons privilégiée dans la recherche d'algorithmes est la simplicité : il doit être possible de proposer un schéma général facile à comprendre, même si l'algorithme implémenté est plus complexe. En effet, sous réserve de faisabilité en temps de calcul, un algorithme destiné à aider un concepteur doit pouvoir être compris de celui-ci, au moins dans ses grandes lignes et non pas apparaître comme une boîte noire aux résultats incompréhensibles et donc soupçonnables. Les étapes intermédiaires des algorithmes pourraient être par exemple visualisées par l'outil de conception. En cela nous avons donc une position un peu décalée par rapport à une recherche en algorithmique traditionnelle. Par contre, ce qui reste fondamental dans notre approche est de spécifier clairement et de prouver la correction des algorithmes.

Ce mémoire n'est pas le lieu pour rappeler en détails toutes les études algoritmiques,

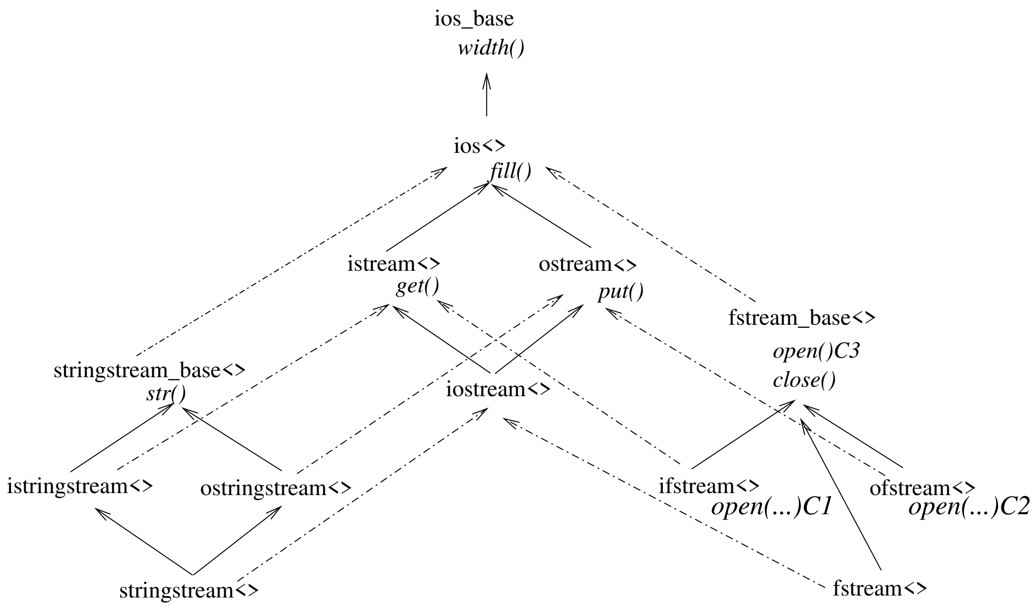


FIG. 4.8 – Interprétation 3 - résultat de la reconstruction

mais il est utile de donner schématiquement les processus de construction. Dans le cas de la construction globale, nous n'entrons pas dans trop de détails et renvoyons à la thèse d'Hervé Leblanc [Leb00] car l'algorithme proposé n'a pas subi de variations. Dans le cas de la construction incrémentale (par ajout successif de classes), par contre, plusieurs algorithmes ont été publiés dont nous avons souhaité faire ici une synthèse. Cette synthèse prend la forme d'un nouveau schéma de construction (que nous espérons plus clair) et d'une explicitation des correspondances entre ce schéma et les versions antérieures. Le nouveau schéma de construction est ensuite réécrit comme un algorithme de maintien d'une sous-hiérarchie de Galois dans le cas de l'ajout d'un objet (d'une ligne dans la matrice). Une version duale nous procurerait de ce fait un algorithme de maintien d'une sous-hiérarchie de Galois dans un cas important de l'ajout d'une propriété (ajout d'une colonne dans la matrice).

Nous utilisons les notations du chapitre 3.

4.3.1 Construction globale

Spécification

Donnée. Une relation binaire $R \subseteq E \times F$
Résultat. Le graphe de la réduction transitive de la sous-hiérarchie de Galois associée à R, chaque sommet contient un concept et sa forme simplifiée.

Un algorithme simple La construction globale se base sur la définition constructive (Def. 3.19). Un premier schéma est présenté dans la figure 4.9. Il consiste à calculer tout d'abord les concepts de la sous-hiérarchie de Galois en utilisant la définition constructive, par application successive de f (resp. g) puis de g (resp. f) à chaque élément de E (resp. F).

La figure 4.10 illustre, avec la relation de la figure 4.2, les trois opérations importantes dans une construction globale utilisant la définition constructive.

Construction globale simple
Construction des concepts
Calcul de l'ensemble $O_C \cup O_P$
Organisation des concepts
Utilisation des relations d'inclusion entre fermés
Simplification des concepts
Utilisation des liens construits pour produire la forme simplifiée des concepts

FIG. 4.9 – Schéma simple de construction

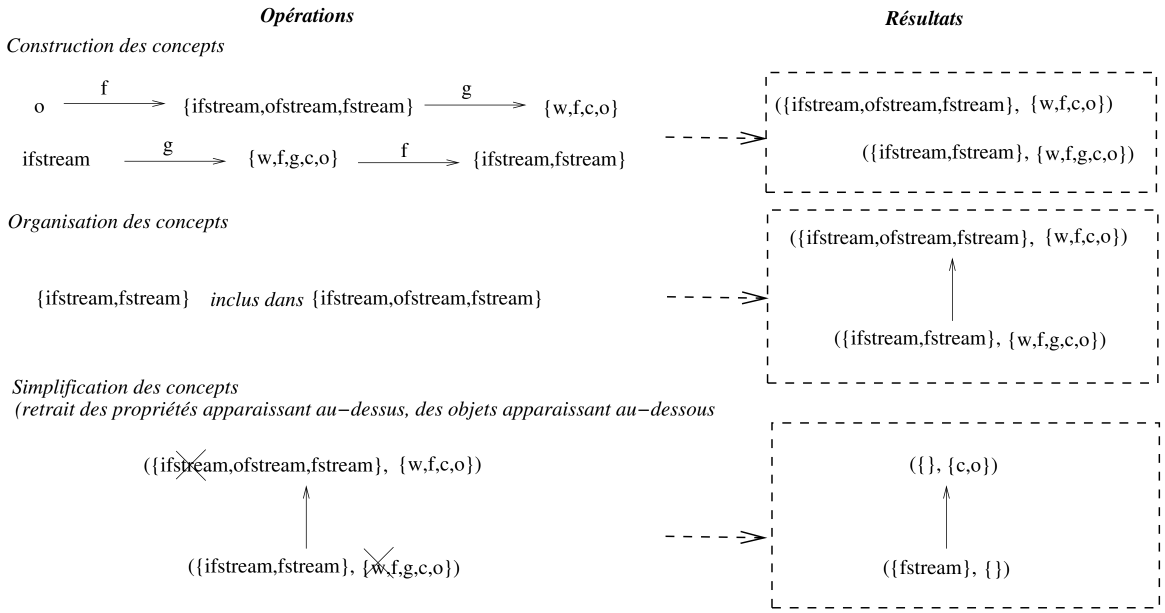


FIG. 4.10 – Opérations majeures de la construction globale

Optimisation Améliorer ce premier schéma peut se faire en prêtant attention aux deux points suivants :

- il faut éviter de générer plusieurs fois un même concept, en calculant $f(o)$ et $f(c)$ par exemple, on trouve le même concept $(\{ifstream, ofstream, fstream\}, \{w, f, o, c\})$;
- les engendrer dans un ordre respectant l'ordre d'inclusion des extensions (ou l'ordre inverse) permet de limiter le nombre de relations d'inclusion à examiner pour construire les liens.

L'algorithme CERES [HL00, Leb00] est construit selon ce principe. CERES est entièrement décrit et démontré, et sa complexité est étudiée dans [Leb00]. Il a été implémenté en Java par H. Leblanc et testé dans le cadre de la construction d'interfaces Java (section 4.4) et du calcul de métriques (section 4.6). Il a été également implémenté en langage J dans Objecteering par C. Roume en avance de phase sur le sous-projet MACAO 2.3.

Nous décrivons ci-dessous les grandes lignes de cet algorithme.

Les concepts de O_P sont calculés en suivant une extension linéaire (tri topologique) de l'ordre $<_{\mathcal{S}}$. Les concepts de $O_C \setminus O_P$ sont calculés dès que possible toujours en respectant une extension linéaire de $<_{\mathcal{S}}$. Nous rappelons tout d'abord la définition d'extension linéaire.

DÉFINITION 4.1 (EXTENSION LINÉAIRE) *Soit l'ordre partiel* $(\mathcal{H}, <_{\mathcal{H}})$, *une extension linéaire* $<_l$ *de* $<_{\mathcal{H}}$ *est un ordre total*[4] *compatible avec* $\mathcal{H}$, *i.e.* $x \leq_{\mathcal{H}} y \Rightarrow x \leq_l y$.

CERES utilise une extension linéaire de $>_{\mathcal{S}}$ prise dans l'ensemble $ExtG$ défini ci-dessous [Leb00] .

DÉFINITION 4.2 (EXTG) $ExtG$ *est l'ensemble des suites de concepts de* $2^E \times 2^F$, *notées* $extg = ((X_1, Y_1) \dots (X_i, Y_i) \dots (X_n, Y_n))$ *et telles que :*

- *tous les éléments de la sous-hiérarchie de Galois apparaissent dans la suite.* $\{(X_i, Y_i), i \in [1 \dots n]\} = \mathcal{S}$
- *les concepts sont ordonnés par taille décroissante de leur extension :* $\forall (i,j) \in [1 \dots n]^2$, $(X_i, Y_i) <_{extg} (X_j, Y_j)$ *ssi* $|X_i| \geq |X_j|$

Les éléments de O_P sont générés et classés sous la forme (X, Y_s) (intension simplifiée) et en respectant une extension de $ExtG$. Pour cela, les colonnes de la matrice sont triées par poids décroissant (cardinal décroissant des extensions). Une deuxième optimisation consiste à calculer les éléments de O_P par groupes de concepts dont l'extension est de même cardinal. Ceci permet de considérer une partie seulement de la matrice lors de la génération des éléments de O_P. Ces éléments de O_P sont classés d'après l'inclusion de leurs extensions et en évitant la génération d'arcs de transitivité. Leur extension simplifiée est également calculée, ainsi on connaît quels objets sont définis par le nœud. Il sera inutile de chercher à calculer le concept associé à ces objets puisqu'on l'a déjà obtenu par calcul à partir d'une propriété. Enfin, les éléments de $O_C \setminus O_P$ sont introduits dès que leur ensemble de propriétés est déclaré par les éléments de O_P calculés. On note, pour un concept c donné, $Ext(c)$ son extension, $Ext_s(c)$ son extension simplifiée, $Int(c)$ son extension et $Int_s(c)$ son extension simplifiée.

CERES
Trier R par poids décroissant des lignes
Pour chaque poids i possible de ligne (en décroissant) faire
    Calculer CC, ensemble des concepts $c \in O_P$ tels que $|Int| = i$
            $Ext(c)$ et $Int_s(c)$ sont obtenus lors de ce calcul
    Pour chaque élément c de CC
        Rattachement de c à ses sur-concepts directs et calcul combiné de $Int(c)$
        calcul de $Ext_s(c)$
        Pour les concepts c' de $O_C \setminus O_P$ dont l'intension est à présent couverte
            calcul de $Ext(c')$, $Int(c')$, $Ext_s(c')$ et $Int_s(c')$
            et rattachement de c' à ses sur-concepts

FIG. 4.11 – Schéma de CERES

Déroulement sur un exemple L'exemple des figures 4.2 et 4.3 va nous servir d'appui. Les poids possibles de colonnes sont $11, 10, 6, 3$.
Poids=11 Le concept 1 est généré, ainsi que $Ext(1)$ (toutes les classes de départ) et $Int_s(1) = \{w\}$.

[4] $\forall x, y \in \mathcal{H}$ avec $x \neq y$ on a $x <_l y$ ou $y <_l x$

Poids=10 Le concept 2 est généré, ainsi que $Ext(2)$ (toutes les classes de départ sauf ios_base) et $Int_s(2) = \{f\}$. le concept 2 est placé sous 1 et $Int(2)$ est complétée par w (héritée de 1).
Poids=6 Les concepts 3 et 4 sont générés. $Ext(3) = \{istream, istringstream, ifstream, iostream, stringstream, fstream\}$ et $Int_s(3) = \{g\}$. $Ext(4) = \{ostream, ostringstream, ofstream, iostream, stringstream, fstream\}$ et $Int_s(4) = \{p\}$. Les concepts 3 et 4 sont placés sous 2 et leurs intensions sont complétées par w, f (héritées de 2). A ce moment le concept de $O_C \setminus O_P$ engendré par $iostream$ voit toutes ses propriétés couvertes (déjà placées dans la hiérarchie). Il doit être créé (donnant le concept 5) et placé sous 3 et 4.
Poids=3 Les concepts 6 (engendré par s) et 7 (engendré par o ou c) sont créés. 6 est attaché à 3 et 7 attaché sous 4. A ce moment-là, tous les concepts restants (engendrés seulement par une classe) doivent être créés et proprement rattachés (pour respecter l'extension linéaire ils sont traités par intensions croissantes).

Indications sur les complexités théoriques et pratiques La complexité théorique en temps a été évaluée grossièrement comme étant dans $O(|E|(|E| + |F|)^2)$ (il faut se rappeler qu'une réduction transitive partielle est effectuée). La complexité en espace est dans $O(|E||F|)$.

L'implémentation de l'algorithme en Java (JDK 1.2) a bénéficié des structures `ArrayList` pour la représentation de E et F, ainsi que `BitSet` pour R (plus précisément f et g ont été stockées toutes les deux ce qui peut être amélioré). La faisabilité pratique a été démontrée en Java par des tests réalisés avec des données tirées de l'API Java. Pour donner un exemple de ces tests, pour 696 classes et 4697 propriétés, R contenait 14309 paires, et le temps d'exécution était inférieur à 11 sec sur un PC standard sous Linux. Pour plus de détails, le lecteur est convié à lire [Leb00].

Algorithmes connexes Dans une étude comparative [HDL00], nous avons étudié divers algorithmes globaux de construction de hiérarchies de classes. La plupart des références ne définissent pas la structure obtenue [Coo92, CL96, MC96], et ont donc nécessité un travail de caractérisation. Nous avons ainsi montré que ces approches construisent des hiérarchies qui se rapprochent d'une sous-hiérarchie de Galois, mais oublient certains liens et ajoutent quelques classes. L'algorithme proposé par [MGG90] construit une sous-hiérarchie de Galois dans le cas restreint où les classes initiales (les objets formels base de la construction) sont définies par les concepts feuilles. CERES est donc actuellement le seul algorithme connu qui construise globalement une sous-hiérarchie de Galois.

4.3.2 Ajout de classes

L'algorithme d'ajout de classe a été le premier étudié chronologiquement. Les études algorithmiques proprement dites ont été réalisées avec H. Dicky et T. Libourel [DDHL94a, DDHL94b, DDHL95, DDHL96, DHL97a] (C. Dony et N. Prade ayant collaboré dans ces publications à la définition d'une comparaison *ad hoc* entre propriétés dans le cadre de la programmation). Deux formes de l'algorithme ont été proposées dont nous essayons une généralisation dans ce mémoire. N. Prade a implémenté au cours de son stage de DEA la première version. La deuxième version est implémentée sous la direction de C. Dony, en liaison avec le premier méta-modèle développé par T. Brunel et est en cours de validation. La première version, qui correspond à notre méta-modèle actuel, doit être portée en J dans Objecteering lors du sous-projet MACAO 3.2.

Spécification Les premiers algorithmes ARES ayant été définis directement sur des hiérarchies de classes plutôt que comme des algorithmes d'ajout d'un objet formel dans une sous-hiérarchie de Galois, nous devons effectuer le lien entre ces deux présentations. De plus nous restreindrons ici notre étude au cas où la hiérarchie de départ peut elle-même être associée à une sous-hiérarchie de Galois comme nous le définissons ci-dessous.

DÉFINITION 4.3 (CONSISTANCE ENTRE HIÉRARCHIE ET SOUS-HIÉRARCHIE DE GALOIS) *Soit un ensemble de propriétés génériques $\mathcal{P}_{gen}$. Soit une hiérarchie H déterminée par $(\mathcal{C}, \leq_H)$ et deux applications $Intension_r$, $Propre_r$ définies comme dans la définition 2.4. Nous supposons que $Intension_r$ vérifie la propriété 2.1.*

Soit la relation $R \subseteq \mathcal{C} \times \mathcal{R}_{ef}$ définie par $(C, r) \in R$ si $r \in Intension_r(C)$, et soit $\mathcal{S}(R)$ la sous-hiérarchie de Galois associée à R.

H est consistante avec $\mathcal{S}(R)$ s'il existe un isomorphisme $\phi : H \longrightarrow \mathcal{S}(R)$ tel que pour tout $C \in \mathcal{C}$, $Intension_r(C) = Int(\phi(C))$ et $Propre_r(C) = Int_s(\phi(C))$.

Dans le paragraphe précédent nous avons ainsi vu une hiérarchie de classes (Fig. 4.3) et une sous-hiérarchie (Fig. 4.4) de Galois consistantes.

La consistance s'étend naturellement aux extensions. Soit $C \in \mathcal{C}$, $\{C'/C' \leq_H C\} = Ext(\phi(C))$.

> **Donnée.** Une hiérarchie de classes H donnée avec les ensembles $Intension_r$, $Propre_r$, une classe A donnée avec $Intension_r(A)$. Les intensions de A des classes de H respectent la contrainte de bonne formation (Prop. 2.1) et définissent une relation R telle que H est consistante avec la sous-hiérarchie de Galois $\mathcal{S}(R)$.
> Le diagramme de $<_H$ est maintenu grâce à des ensembles $superClasseIm(C)$ associant à chaque classe C ses super-classes immédiates.
>
> **Résultat.** Une hiérarchie de classes H_f consistante avec la sous-hiérarchie de Galois $\mathcal{S}(R')$ avec $R' = R \cup \{(A, r)/r \in Intension_r(A)\}$.

Schéma d'algorithme Comme nous l'avons dit ci-dessus, nous proposons un schéma de l'algorithme qui généralise les deux versions précédentes (l'une correspondant au modèle actuel où les généralisations de deux propriétés sont connues à l'avance et $Intension_r$ est donnée pour toute classe ; l'autre destinée à un calcul à la volée des généralisations les plus spécifiques de deux propriétés, il n'y a stockage, pour une classe donnée que de $Propre_r$). Nous spécifions tout d'abord abstraitement quelques ensembles et opérateurs utiles pour comparer une classe C de H et la classe ajoutée A. Ces ensembles correspondent à la généralisation des deux classes ($\wedge_a$) et au calcul d'une différence ($\setminus_a$).

- *les propriétés communes à C et A* $CP(C, A) = C \wedge_a A$
- *les propriétés communes non héritées par C* $EP(C, A) = C_s \wedge_a A$
- *les propriétés de C que A n'a pas* $RC(C, A) = C \setminus_a A$
- *les propriétés de A que C n'a pas* $RA(C, A) = A \setminus_a C$
- *l'opération de retrait (ou de factorisation) de E_1 des propriétés de E_2* $E_1 \setminus =_a E_2$ pour résumer la notation $E_1 = E_1 \setminus_a E_2$

Pour notre modèle actuel, ces ensembles sont ainsi définis.

- *les propriétés communes à C et A* $CP(C, A) = Intension_r(C) \cap Intension_r(A)$
- *les propriétés communes non héritées par C* $EP(C, A) = Propre_r(C) \cap Intension_r(A)$
- *les propriétés de C que A n'a pas* $RC(C, A) = Intension_r(C) \setminus CP$
- *les propriétés de A que C n'a pas* $RA(C, A) = Intension_r(A) \setminus CP$

- *l'opération de retrait de E_1 des propriétés de E_2* $E_1 \backslash = E_2$ résume la notation $E_1 = E_1 \setminus E_2$

L'algorithme (voir Figure 4.13) parcourt la hiérarchie du haut vers le bas dans une extension linéaire de l'ordre de spécialisation. L'exploration se limite aux classes C qui ont des référents communs avec A, c'est-à-dire telles que $CP \neq \emptyset$. Chaque fois qu'une classe est visitée (après toutes ses super-classes), elle est comparée avec la nouvelle classe. Cette comparaison s'articule autour des quatre situations suivantes (voir Figure 4.12). Dans l'analyse des cas, on suppose qu'aucune classe n'est vide.

- *Cas 1* C et A ont même intension, ou encore $Intension_r(A) = Intension_r(C)$, ou encore $RC = RA = \emptyset$.
- *Cas 2* C est super-classe de A, ou encore $Intension_r(A) \supset Intension_r(C)$, ou encore $RC = \emptyset$ et $RA \neq \emptyset$.
- *Cas 3* A est super-classe de C, ou encore $Intension_r(A) \subset Intension_r(C)$, ou encore $RC \neq \emptyset$ et $RA = \emptyset$.
- *Cas 4* C et A sont incomparables mais partagent des propriétés qu'il est nécessaire de factoriser, ou encore $RC \neq \emptyset$ et $RA \neq \emptyset$ et $EP \neq \emptyset$.

Dans le *Cas 1*, l'algorithme peut terminer puisqu'on a trouvé une classe qui représente A (ligne 6).

Dans le *Cas 2*, C est mémorisée comme super-classe de A à la ligne 9 (pas forcément directe). Les référents de A que C en propre seront hérités par A et ne lui seront plus propres (ligne 10).

Dans le *Cas 3*, si A n'a pas encore été matérialisée, elle l'est (ligne 14). On vient de rencontrer une première sous-classe de A. EP peut être vide (mais $CP(C, A) \neq \emptyset$) ou non. Les éventuels référents de A que possèderait aussi en propre C ne sont désormais plus propres à C (ligne 15). Toutes les super-classes de A sont aussi super-classes de C. De plus elles ont forcément été rencontrées par examen des super-classes de C dans la hiérarchie initiale. En effet une super-classe de A qui n'est pas super-classe de C dans la hiérarchie initiale aurait nécessairement été créée pour factoriser des référents communs à A et une certaine classe B incomparable avec C. Or ces référents sont aussi communs à C et B, contradiction avec la propriété de sous-hiérarchie de Galois. On connaît donc les super-classes immédiates de A dès ce point de l'algorithme (ligne 16). Si A est déjà matérialisée, c'est qu'une de ses sous-classes D (incomparable avec C) a déjà été rencontrée. Dans ce cas, comme on l'a déjà vu, D et C ne déclarent rien en propre qui soit également dans A et on sait aussi que D et C ont les mêmes super-classes. Il est donc inutile d'effectuer les lignes 15 et 16. Dans tous les cas, que A soit ou non matérialisée, C a comme super-classe directe A (ligne 18) et n'est plus directement reliée aux super-classes directes de A (ligne 19).

Dans le *Cas 4*, A et C sont incomparables, mais C déclare des référents que A possède. Il faut alors créer une nouvelle classe de factorisation C' (ligne 23), remettre à jour les super-classes directes de C (lignes 25 et 28), les super-classes de A (ligne 26), attacher C' à ses super-classes directes (les plus petites super-classes communes à C et A, ligne 27). Les intensions propres de A et C sont privées des référents factorisés ou généralisés (lignes 29 et 30).

Lorsqu'aucune classe ne sous-classe A, cette dernière doit être créée à la fin de l'exploration (ligne 34) et rattachée (ligne 35).

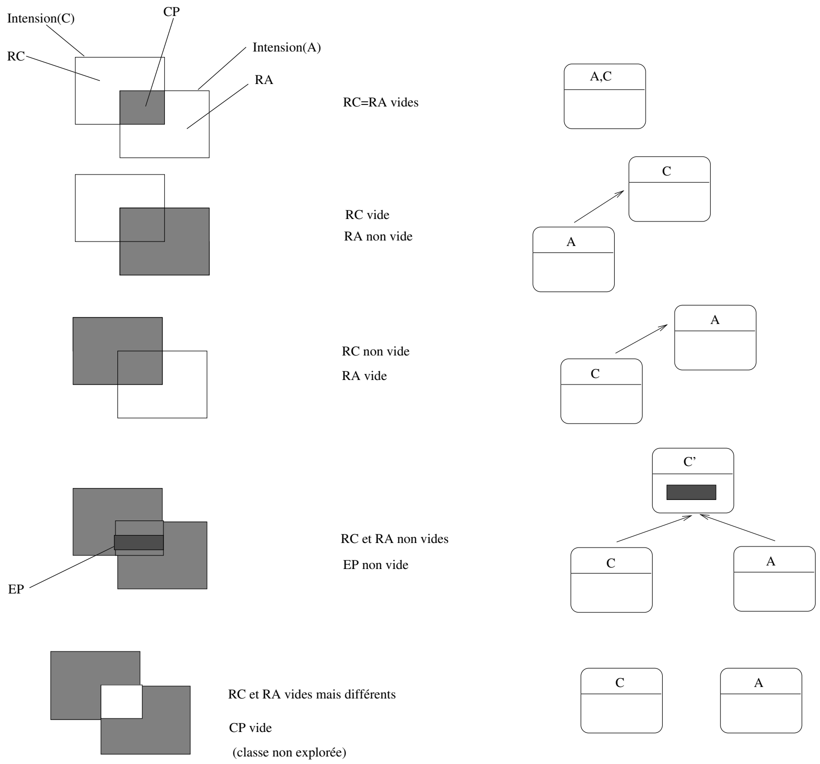


FIG. 4.12 – Cas fondamentaux de l'algorithme ARES-SCHÉMA (les parties blanches sont vides)

Correspondance avec Ares1 Pour rapprocher ARES-SCHÉMA de ARES1 et ARES2, il faut dans un premier temps supposer que l'ensemble E des classes ayant servi à la construction initiale de la hiérarchie (ou plus généralement un ensemble de classes « pertinentes ») est strictement inclus dans l'ensemble des classes de H. Il est facile d'étendre ARES-SCHÉMA en mémorisant les classes C telles que $Propre_r(C)$ se vide (cas 3 et 4) puis en les faisant disparaître. Cela correspondrait aux opérations effectuées dans ARES1 sur les classes non pertinentes et dans ARES2 sur l'ensemble $\mathcal{C}mean$. Ces instructions ne seront pas commentées dans notre mise en parallèle où nous supposerons que les classes de H et la nouvelle classe A sont toutes pertinentes.

ARES1 est divisé en plusieurs fonctions que nous rappelons dans le formalisme d'origine. Ce dernier est mis en parallèle avec celui du présent document comme suit :

- $Attributs(C)$ correspond à $Intension_r(C)$,
- $Declares(C)$ correspond à $Propre_r(C)$,
- $Herites(C)$ n'est pas mis à jour dans ARES-SCHÉMA (nous ignorerons les instructions correspondantes),
- $Superclassesdirectes$ correspond à $superClasseIm$,
- SH correspond à $superClasseA$,
- $Sups(C, SH)$ est défini comme $Min(superClasses(C), superClasseA)$.

L'algorithme principal fonctionne selon deux grandes phases, la première recherchant et/ou créant les super-classes de A et la deuxième rattachant à A ses sous-classes.

La première phase (CHERCHEETRATTACHESUPERCLASSES) visite H selon une extension linéaire descendante qui correspond au cœur de l'algorithme ARES-SCHÉMA (lignes 3 à 32). La deuxième phase (RATTACHESOUSCLASSES) est traitée par ARES-SCHÉMA dans le cas 3 (A superclasse de C) par rattachement de C comme sous-classe de A (ligne 18),

Ares(A)-schéma

Soit A la classe à ajouter, donnée avec $Intension_r(A)$

$superClassesA = \emptyset$
$Propre_r(A) = Intension_r(A)$
pour chaque classe C (t.q. $CP(C, A) \neq emptyset$) dans une extension linéaire de $>_H$ **faire**
  Calcul de $EP = EP(C, A)$, $RC = RC(C, A)$ et $RA = RA(C, A)$
  si $RC = RA = \emptyset$ **alors** *(Cas 1 - C et A sont égales)*
    fin de l'algorithme
  finSi
  si $RC = \emptyset$ et $RA \neq \emptyset$ **alors** *(Cas 2 - C super-classe de A)*
    $superClasseA \cup = \{C\}$
    $Propre_r(A) \setminus =_a Propre_r(C)$
  finSi
  si $RC \neq \emptyset$ et $RA = \emptyset$ **alors** *(Cas 3 - A super-classe de C)*
    si A n'est pas matérialisée **alors**
      créer une nouvelle classe qui représente A
      si $EP \neq \emptyset$ **alors** $Propre_r(C) \setminus =_a Intension_r(A)$ **finSi**
      $superClasseIm(A) = Min(superClassesA)$
    finSi
    $superClasseIm(C) \cup = \{A\}$
    $superClasseIm(C) \setminus = superClasseIm(A)$
    Eliminer la descendance de C de l'extension linéaire
  finSi
  si $RC \neq \emptyset$ et $RA \neq \emptyset$ et $EP \neq \emptyset$ **alors** *(Cas 4 - factoriser C et A)*
    créer une nouvelle classe C'
    $Propre_r(C') = EP$
    $superClasseIm(C) \cup = \{C'\}$
    $superClasseA \cup = \{C'\}$
    $superClasseIm(C') = Min(superClasse(C) \cap superClasseA)$
    $superClasseIm(C) \setminus = superClasseIm(C')$
    $Propre_r(C) \setminus =_a Propre_r(C')$
    $Propre_r(A) \setminus =_a Propre_r(C')$
  finSi
finPour
si aucun sommet représentant A n'existe **alors**
  créer une nouvelle classe A
  $superClasseIm(A) = Min(superClasseA)$
finSi

Fig. 4.13 – Schéma de Ares

élimination des arcs de transitivité ainsi créés (ligne 19) et élimination de la descendance de C dans les futures explorations (ligne 20).

L'algorithme VISITE de ARES1 est organisé autour de trois cas principaux au lieu des quatre de ARES-SCHÉMA.

Le premier cas de ARES1 (lignes v1-v3) correspond au cas 1 de ARES-SCHÉMA. Dans ARES-SCHÉMA, nous n'avons pas formalisé la création sous forme d'un booléen, ni traité la différence entre des classes pertinentes et des classes non pertinentes, aussi l'algorithme peut s'arrêter. H étant une sous-hiérarchie de Galois, il est en effet inutile de parcourir d'autres classes.

Le deuxième cas de ARES1 (lignes v4-v5) correspond au cas 2 de ARES-SCHÉMA. Les lignes (v5) et (9) se correspondent, tandis que la ligne (1O) est une version incrémentale de la mise à jour de l'ensemble des référents propres à A effectuée globalement dans ARES1 à la ligne (ca9).

Le troisième cas de ARES1 (lignes v6-v7) couvre le cas 4 et une partie du cas 3 (lorsque $EP \neq \emptyset$)) de ARES-SCHÉMA. Il est traité par appel de l'algorithme EXTRACTION que nous examinons à présent.

Nous rappelons que EXTRACTION est appelé lorsque C ne représente pas et n'est pas super-classe de A, et que $EP \neq \emptyset$. Il faut alors créer une classe de factorisation qui peut ou non représenter A (ligne 1 dans EXTRACTION / lignes 13 et 14 dans ARES-SCHÉMA). La création ligne 13 est conditionnelle, mais nous avons vu que si A est superclasse de plusieurs classes, on a forcément $EP = \emptyset$ donc nous sommes hors du cas étudié. La définition des référents propres à C' (ligne e2) est réalisée dans ARES-SCHÉMA-cas4 à la ligne 24. Dans le cas 3 elle est inutile car ces référents sont déjà dans $Propre_r(A)$ depuis l'initialisation (ligne 2). Le rattachement de C' à ses super-classes directes (ligne e4) est effectué ligne 16 pour le cas 3 (toutes les superclasses de A sont superclasses de C ce qui simplifie l'écriture) et ligne 27 pour le cas 4. La boucle de suppression des arcs de transitivité (lignes e5-e7) est retranscrite dans les instructions ligne 19 (cas 3) et ligne 28 (cas 4). La ligne e9 (ajout de A ou C' aux superclasses immédiates de C) est réécrite respectivement par les lignes 18 et 25. La ligne e10 (mise à jour des propriétés propres à C) correspond aux lignes 15 et 29 du cas 4. La ligne 15 est conditionnelle car elle est inutile dans le cas où A aurait plusieurs sousclasses. La ligne e15 (effectuée seulement lorsque A n'est pas superclasse de C, et qui mémorise C' comme superclasse de A) correspond à la ligne 26.

Après l'exploration la classe A doit être créée si elle ne l'a pas déjà été. Dans ARES1 cela se fait lors de appel de CREEETATTACHESH qui correspond aux instructions 33-36 dans ARES-SCHÉMA. La différence entre les deux algorithmes réside dans le fait que l'action de la ligne (ca9), qui consiste à mettre à jour les référents propres de A a été réalisée incrémentalement dans les lignes 10 et 30.

Enfin l'algorithme RATTACHESOUSCLASSES est réalisé incrémentalement par le cas 3.

Cette liaison entre les deux algorithmes nous assure la correction de l'algorithme ARES-SCHÉMA, celle de ARES1 ayant été prouvée dans [DDHL94a].

ARES1(A)
Soit A la classe à ajouter, donnée avec Attributs(A)
(a1) ChercheEtRattacheSuperClasses
(a2) RattacheSousClasses

FIG. 4.14 – Schéma de ARES1

CHERCHEETRATTACHESUPERCLASSES
(cr1)**pour** chaque classe C dans une extension linéaire de $>_H$
(cr2) Visiter(C,A)
finPour
(cr3)CreeEtAttacheSH
(cr4)SupprimeClassesVides

FIG. 4.15 – Schéma de ARES1 - ChercheEtRattacheSuperClasses

Correspondance avec Ares2 L'algorithme ARES2 est rappelé dans la figure 4.21. Dans cet algorithme, on suppose que l'on dispose d'un opérateur dont le rôle est semblable aux heuristiques de calcul des référents et des propriétés génériques. Cet opérateur, noté LCG, associe à deux référents un référent qui représente leur(s) plus petite(s) généralisation(s) commune(s). Si on reprend l'exemple de [DDHL96],soient a=≤`(Date) [Code1]` et b=≤`(Time)[Code1]`, une heuristique, telles que celles décrites dans [DDHL96] peut donner comme résultat $LCG(a,b)$=≤`(selfType)[Code1]`. Lorsque les référents sont comparables, cette généralisation peut être l'un des deux.

Les opérations utilisées dans ARES-SCHÉMA se définissent comme suit.
- *les propriétés communes à C et A* $CP(C,A) = C \wedge_a A$ est l'ensemble des $LCG(c,a)$ avec les référents c de C et a de A.
- *l'opération de retrait (ou de factorisation) de E_1 des propriétés de E_2* $E_1 \setminus =_a E_2$, noté dans l'algorithme $E_1 = Difference(E_1, E_2)$ effectue une différence ensembliste et produit en même temps une forme syntaxique plus intéressante pour représenter certains référents (par exemple introduit un appel à une partie de méthode factorisée).

VISITER(C,A)
(v1) **si** $Attributs(A) = Attributs(C)$ **alors** *(C = A)*
(v2) RendrePertinente(C)
(v3) ADéjàCréée ⟵ vrai
(v4) **sinon si** $Attributs(A) \supset Attributs(C)$ **alors** *(C superclasse de A)*
(v5) $SH \longleftarrow SH \cup \{C\}$
(v6) **sinon si** $Attributs(A) \cap Declares(C) \neq \emptyset$ **alors** *(factoriser C et A)*
(v7) Extraction(C) **finSi finSi finSi**

FIG. 4.16 – Schéma de ARES1 - Visiter

Extraction(C)

(e1) Créer C'

(e2) $Declares(C') \longleftarrow Attributs(A) \cap Declares(C)$

(e3) $Herites(C') \longleftarrow \emptyset$

(e4) $Superclassesdirectes(C') \longleftarrow Sups(C, SH)$

(e5) **pour** tout $C'' \in Sups(C, SH)$ **faire**

(e6)   **si** $C'' \in Superclassesdirectes(C)$ **alors**

(e7) $Superclassesdirectes(C) \longleftarrow Superclassesdirectes(C) \setminus \{C''\}$ **finSi**

(e8) $Herites(C') \longleftarrow Herites(C') \cup Attributs(C'')$

finPour

(e9) $Superclassesdirectes(C) \longleftarrow Superclassesdirectes(C) \cup \{C'\}$

(e10) $Declares(C) \longleftarrow Declares(C) \setminus Declares(C')$

(e11) $Herites(C) \longleftarrow Herites(C) \cup Declares(C')$

(e12) **si** $Attributs(C') = Attributs(A)$ **alors**

(e13)   RendrePertinente(C')

(e14)   ADéjàCréée $\longleftarrow$ vrai

(e15) **sinon** $SH \longleftarrow SH \cup \{C'\}$ **finSi**

(e16) **si** $Declares(C) = \emptyset$ et *non* $EstPertinente(C)$ **alors**

(e17) $ClassesVides \longleftarrow ClassesVides \cup \{C\}$ **finSi**

Fig. 4.17 – Schéma de Ares1 - Extraction

CreeEtAttacheSH

(ca1) **si** *non* AdéjàCréée **alors**

(ca2) Créer A

(ca3) RendrePertinente(A)

(ca4) $Superclassesdirectes(A) \longleftarrow \emptyset$

(ca5) $Herites(A) \longleftarrow \emptyset$

(ca6) **pour** tout $C'' \in Min(SH)$ **faire**

(ca7) $Superclassesdirectes(A) \longleftarrow Superclassesdirectes(A) \cup \{C''\}$

(ca8) $Herites(A) \longleftarrow Herites(A) \cup Attributs(C'')$

finPour

(ca9) $Declares(A) \longleftarrow Attributs(A) \setminus Herites(A)$

Fig. 4.18 – Schéma de Ares1 - CreeEtAttacheSH

SUPPRIMECLASSESVIDES

(s1) **pour** tout $T \in ClassesVides$ **faire**
(s2) **pour** tout $Y \in Superclassesdirectes(T)$ **faire**
(s3) **pour** tout $U \in Sousclassesdirectes(T)$ **faire**
(s4) $Superclassesdirectes(U) \longleftarrow Superclassesdirectes(U) \cup \{Y\}$
(s5) $\mathcal{E} \longleftarrow Superclassesdirectes(U)$
(s6) Detruit $\longleftarrow$ faux
(s7) **répéter**
(s8) $Z \longleftarrow Extraire(\mathcal{E})$
(s9) **si** $Z \in SousClasses(Y)$ **alors**
(s10) $Superclassesdirectes(U) \longleftarrow Superclassesdirectes(U) \setminus \{Y\}$
(s11) Detruit $\longleftarrow$ vrai
finSi
(s12) **jusqu'à** $Vide(\mathcal{E})$ ou Detruit
finPour **finPour**
(s13) Detruire(T)
finPour

FIG. 4.19 – Schéma de ARES1 - SUPPRIMECLASSESVIDES

RATTACHESOUSCLASSES

(r1) **pour** tout T avec $Attributs(A) \subset Attributs(T)$
(r2) et tel que $\forall Y \in Superclassesdirectes(T), Attributs(A) \not\subset Attributs(Y)$
faire
(r3) $Superclassesdirectes(T) \longleftarrow Superclassesdirectes(T) \cup \{A\}$
(r4) **pour** tout $Z \in Superclassesdirectes(T)$ **faire**
(r5) **si** $Z \in Superclasses(A)$ **alors**
(r6) $Superclassesdirectes(T) \longleftarrow Superclassesdirectes(T) \setminus \{Z\}$ **finSi**
finPour
finPour

FIG. 4.20 – Schéma de ARES1 - RATTACHESOUSCLASSES

– *les propriétés communes non héritées par C* $EP(C, A) = C_s \wedge_a A$
– *les propriétés de C que A n'a pas* $RC(C, A) = C \setminus_a A$
– *les propriétés de A que C n'a pas* $RA(C, A) = A \setminus_a C$

L'opération $UpdateSignature$ est de nature esthétique et permet de remplacer le mot-clef SELFTYPE par le nom de la classe qui déclare cette méthode à la fin de l'algorithme.

Nous pouvons à présent établir les correspondances entre les deux algorithmes (en dehors du traitement des classes vides et de l'établissement des liens $SubClasses$).

Le cas 1 de ARES-SCHÉMA correspond au cas (a2-26).

Le cas 2 est traité par les instructions (a2-22) à (a2-25).

Le cas 3 est traité des lignes (a2-8) à (a2-16) (sauf (a2-9)) avec les correspondances suivantes : (a2-8) $\longrightarrow$ 14 ; (a2-9) est inutile pour le cas 3 car $Propre_r(A)$ contient déjà les référents nécessaires ; (a2-10) (sous une forme simplifiée) $\longrightarrow$ 16 ; (a2-11) $\longrightarrow$ 18 ; (a2-12) $\longrightarrow$ 19 ; (a2-13) $\longrightarrow$ 15 ; (a2-14) est la condition de l'instruction 12 ; (a2-15) et (a2-16) correspondent aux parties ignorées.

Le cas 4 est traité des lignes (a2-8) à (a2-13) puis (a2-17) à (a2-19) avec les correspondances suivantes : (a2-8) $\longrightarrow$ 23 ; (a2-9) $\longrightarrow$ 24 ; (a2-10) $\longrightarrow$ 27 ; (a2-11) $\longrightarrow$ 25 ; (a2-12) $\longrightarrow$ 28 ; (a2-13) $\longrightarrow$ 29 ; (a2-17) $\longrightarrow$ 22 ; (a2-18) $\longrightarrow$ 26 ; (a2-19) $\longrightarrow$ 30.

Enfin si A doit être rattachée, ceci est réalisé dans les instructions (e2-27) et suivantes, ce qui correspond à la partie lignes 33 à 36.

Evolution d'une shg Une transformation simple de l'algorithme réécrit ARES-SCHÉMA dans le cadre de l'évolution d'une sous-hiérarchie de Galois par ajout de classe. Elle donne un premier cadre qui peut être optimisé en appliquant les résultats de [DDHL94a].

Nous y utilisons les notations présentées dans CERES. Pour un concept C donné, nous notons $Ext(C)$ son extension, $Ext_s(C)$ son extension simplifiée, $Int(C)$ son intension et $Int_s(C)$ son intension simplifiée. Le diagramme de Hasse de la sous-hiérarchie de Galois est matérialisé en stockant pour chaque concept C les concepts successeurs (« super-concepts immédiats » ou $supIm(C)$) et les concepts prédécesseurs (« sous-concepts immédiats » ou $subIm(C)$).

$superConcepts_A$ remplace $superClasseA$ et représente les concepts qui ont A dans leur extension. $Propre_A$ représente l'ensemble des référents de l'intension simplifiée du concept qui définira A (aura A dans son intension simplifiée).

- *les propriétés communes à C et A* $CP(C, A) = Int(C) \cap f(\{A\})$
- *les propriétés communes non héritées par C* $EP(C, A) = Int_s(C) \cap f(\{A\})$
- *les propriétés de C que A n'a pas* $RC(C, A) = Int(C) \setminus CP$
- *les propriétés de A que C n'a pas* $RA(C, A) = f(\{A\}) \setminus CP$

Nous reprenons les cas fondamentaux en les détaillant schématiquement avant de proposer l'algorithme complet. Nous rappelons que nous étudions seulement les situations où C et $f(\{A\}$ partagent au moins un référent.

Cas 1 - C a pour intension $f(\{A\})$ A doit être ajoutée dans l'extension simplifiée de C (et donc dans son extension). Il est inutile de continuer l'exploration de la sous-hiérarchie de Galois : l'ajout de l'objet formel A a simplement étendu l'extension de certains concepts (C et les éléments plus grands que C). Cette situation est décrite par la figure 4.22. Les éléments des extensions/intensions simplifiées sont en caractères gras. Lorsqu'ils apparaissent en caractères droits on ne fait pas d'hypothèse sur leur appartenance à la partie simplifiée. Lorsqu'ils seront en caractères italiques, dans les figures suivantes, cela signifiera qu'ils appartiennent à la partie héritée de l'intension (ou en sens inverse de l'extension).

Cas 2 - C plus grand que le concept d'intension $f(\{A\})$ A doit être ajoutée dans l'extension de C (mais pas dans son extension simplifiée). Le concept C doit être mémorisé comme étant un super-concept du concept qui définira A. Ce dernier héritera des référents de l'intension simplifiée de C. Comme tous les super-concepts de C seront dans le même cas, le concept qui définira A héritera par construction de tous les référents de C. Cette situation est décrite par la figure 4.23. A apparaît dans l'extension héritée de C (en italiques). a, b, et c seront hérités donc disparaissent de $Propre_A$.

Cas 3 - C plus petit que le concept d'intension $f(\{A\})$ Ce cas couvre trois possibilités :

- *cas3-a* C possède (mais hérite) tous les référents de A ; plusieurs concepts peuvent vérifier cette propriété ;
- *cas3-b* C possède tous les référents de A, et certains sont dans son intension simplifiée ; dans cette situation, le concept défnissant A a pour seul sous-concept direct C.

Dans le *cas3-a*, C hérite de tous les référents de A et $EP = \emptyset$. Plusieurs concepts peuvent avoir cette propriété (voir Figure 4.24). Lorsque le premier d'entre eux est rencontré, il est nécessaire de créer un concept CA qui définit A. Ce concept a A dans son extension simplifiée (et donc dans son extension), tandis que son intension est $f(\{A\})$, et son intension simplifiée est forcément vide ($propre_A$ a été vidé en ce point lors de l'examen des super-classes). Extensions et intensions de C ne sont en rien modifiées. Par contre C doit être

Ares2(A)

Soit A la classe à ajouter, donnée avec $Intension_r(A) = InitialProperties(A)$

(a2-1) $AalreadyCreated \longleftarrow false$; $Superclasses(A) \longleftarrow \emptyset$; $EmptyClasses \longleftarrow \emptyset$
(a2-2) $NewDeclared(A) \longleftarrow InitialProperties(A)$; $DirectSubclasses(A) \longleftarrow \emptyset$
(a2-3) $LEH_i \longleftarrow ComputeLinearExtension(\geq_{H_i})$
Comparison of A and classes C of H
(a2-4) **pour** (chaque classe C in LEH_i) **faire**
(a2-5) Calculer $EP = EP(C, A)$, $RC = RC(C, A)$ et $RA = RA(C, A)$
(a2-6) **si** $RC \neq \emptyset$ **alors**
(a2-7) **si** $RA = \emptyset$ **ou** $EP \neq \emptyset$ **alors**
creation of a factorization class C'
(a2-8) Créer C'
(a2-9) $Declared(C') \longleftarrow UpdateSignatures(EP))$
(a2-10) $ImmediateSuperClasses(C') \longleftarrow MinCommonSpus(C, SuperClasses(A))$
(a2-11) $ImmediateSuperClasses(C) \longleftarrow ImmediateSuperClasses(C) \cup \{C'\}$
(a2-12) $ImmediateSuperClasses(C) \longleftarrow ImmediateSuperClasses(C)$
$\backslash ImmediateSuperClasses(C')$
(a2-13) $Declared(C) \longleftarrow Difference(Declared(C), Declared(C'))$
(a2-14) **si** $RA = \emptyset$ **alors**
C'=A et C sous-classe de A
(a2-15) $AalreadyCreated \longleftarrow true$; $\mathcal{C}_{Mean} \longleftarrow \mathcal{C}_{Mean} \cup \{C'\}$
(a2-16) $DirectSubclasses(A) \longleftarrow DirectSubclasses(A) \cup \{C'\}$
(a2-17) **sinon** *C' est superclasse de A*
(a2-18) $SuperClasses(A) \longleftarrow SuperClasses(A) \cup \{C'\}$
(a2-19) $NewDeclared(A) \longleftarrow Difference(NewDeclared(A), Declared(C'))$
finSi
(a2-20) **si** $Declared(C) = \emptyset$ **et** $C \notin \mathcal{C}_{Mean}$ **alors**
(a2-21) $EmptyClasses \longleftarrow EmptyClasses \cup \{C\}$
finSi
finSi
(a2-22) **sinon** $RC = \emptyset$
(a2-23) **si** $RA \neq \emptyset$ **alors** *C est superclasse de A*
(a2-24) $SuperClasses(A) \longleftarrow SuperClasses(A) \cup \{C\}$
(a2-25) $NewDeclared(A) \longleftarrow Difference(NewDeclared(A), Declared(C))$
(a2-26) **sinon** $AalreadyCreated \longleftarrow true$ *RA=$\emptyset$, C=A*
finSi
finSi
finPour
Créer la classe A et la relier à ses superclasses
(a2-27) **si not** $AalreadyCreated$ **alors**
(a2-28) Créer A ; $\mathcal{C}_{Mean} \longleftarrow \mathcal{C}_{Mean} \cup \{A\}$
(a2-29) $ImmediateSuperClasses(A) \longleftarrow Min(SuperClasses(A))$
(a2-30) $ImmediateSubClasses(A) \longleftarrow DirectSubClasses(A)$
(a2-31) $Declared(A) \longleftarrow NewDeclared(A)$
finSi
(a2-32) DeleteEmptyClasses

Fig. 4.21 – Schéma de Ares2

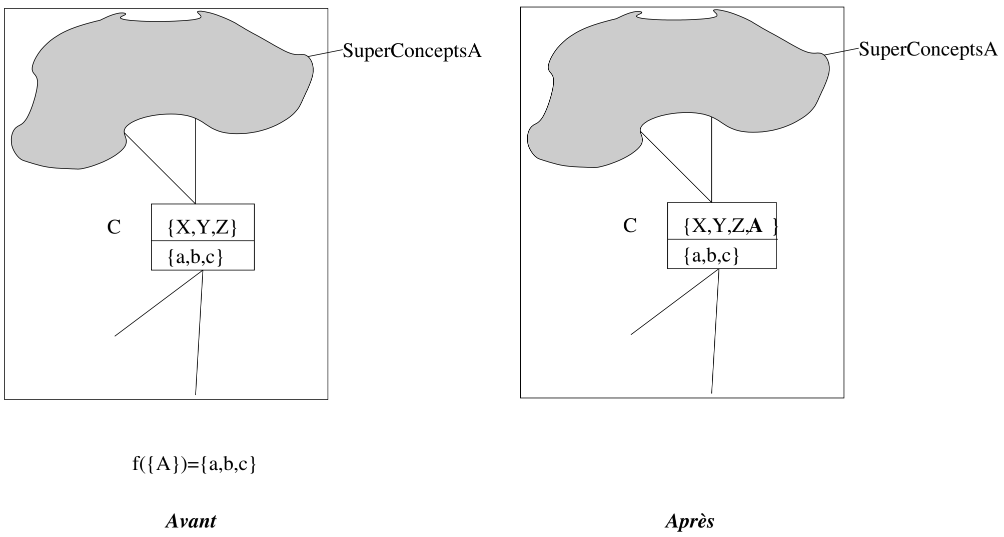


FIG. 4.22 – Cas 1 de l'algorithme ARES-SHG

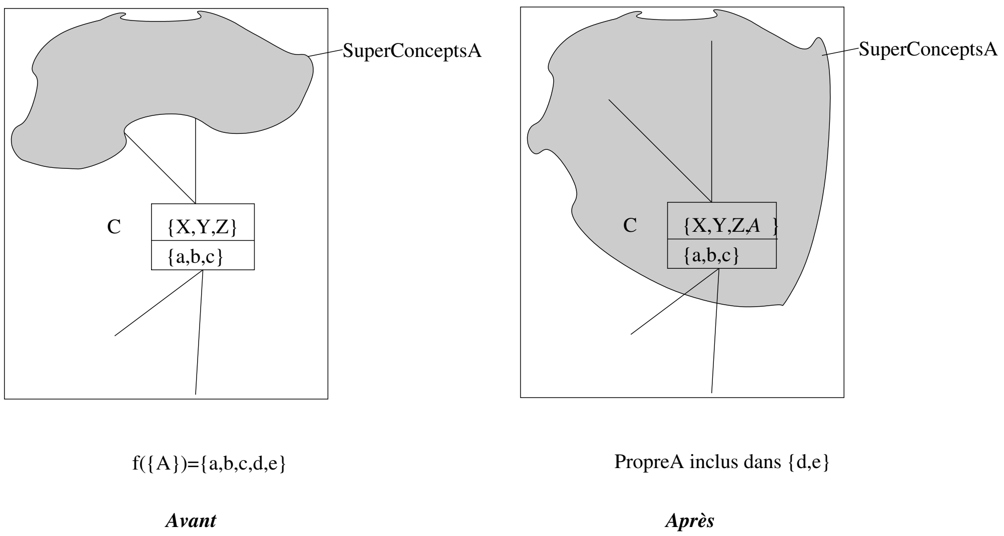


FIG. 4.23 – Cas 2 de l'algorithme ARES-SHG

détaché de ses super-concepts directs et être raccordé à CA tandis que CA est attaché aux super-concepts directs de C (qui sont les minimaux de $superConcepts_A$). Les sous-concepts de C sont ignorés car ils ne peuvent être que des sous-concepts indirects de CA. Lorsqu'un concept héritant de tous les référents de A a déjà été rencontré avant C, alors il ne faut pas créer de nouveau concept pour définir A, ce rôle étant rempli par CA. Par contre l'extension de CA doit recevoir l'extension de C, C doit être détaché de ses super-concepts directs, puis rattaché à CA.

Dans le *cas3-b*, décrit par le croquis de la figure 4.25, C possède tous les référents de A et en a dans son intension simplifiée (sur l'exemple, c) donc $EP \neq \emptyset$. Un nouveau concept CA doit être créé pour définir A. Son extension simplifiée contient A et son extension contient également l'extension de C (par héritage).Son intension est constituée de $f(\{A\})$ tandis que son intension simplifiée est $Propre_A$ (dont les référents de $f(\{A\})$ déjà rencontrés ont été enlevés). L'intension simplifiée de C doit être réduite des référents de l'intension de CA. Si son intension simplifiée devient vide et que son extension est vide, ce concept doit

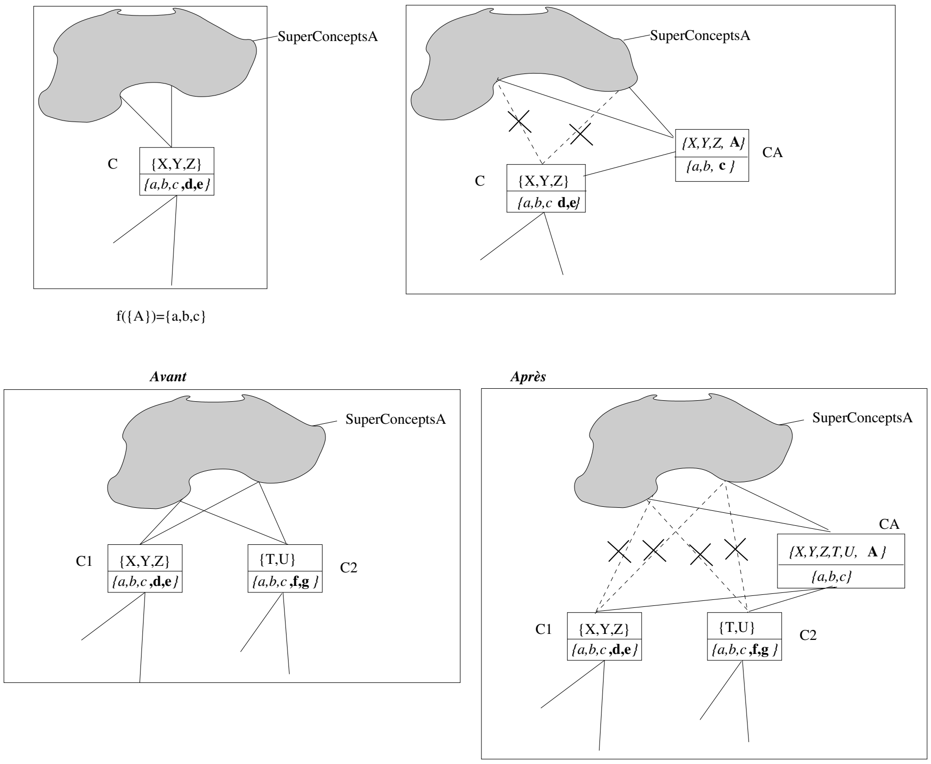


FIG. 4.24 – Cas 3-a de l'algorithme ARES-SHG

disparaître (il est mémorisé dans une liste de sommets à détruire). CA doit être rattachée aux super-concepts qui lui ont été trouvés auparavant (ils sont forcément super-concepts de C). C doit être rattaché à CA et les arcs de transitivité ainsi créés doivent être supprimés. Il est inutile de continuer l'exploration dans les sous-concepts de C qui sont des sous-concepts indirects de CA.

Cas 4 - $EP \neq \emptyset$ et C incomparable avec le concept d'intension $f(\{A\})$ Dans ce dernier cas (figure 4.26), un nouveau concept C' doit être créé pour généraliser les concepts C et CA. Son intension est CP (référents communs aux deux concepts), son intension simplifiée est EP (référents de l'intension simplifiée de C). Son extension est formée de l'extension de C à laquelle on rajoute A. Son extension simplifiée est vide puisqu'aucun objet formel initial, ni A, n'ont d'intension égale à celle de C'. C' est super-concept direct de C, et super-concept de A (pas nécessairement direct). C' a pour super-concepts directs les super-concepts de C qui sont aussi super-concepts de A. Ces super-concepts directs de C' doivent être retirés à C pour ne pas former d'arcs de transitivité. L'intension simplifiée de C et celle du concept qui définira A sont privées des éléments de EP. Si C devient vide et si l'extension de C l'est aussi, C doit disparaître.

Ajout final d'un concept - On n'a pas trouvé ni créé de concept d'intension $f(\{A\})$ A la fin de l'algorithme, si aucun concept rencontré ou créé ne définit A, il faut en créer un nouveau (CA). Ce concept a pour extension A, pour intension simplifiée $Propre_A$ qui correspond à $f(\{A\})$ diminué de tous les référents rencontrés dans H, et naturellement pour intension $f(\{A\})$. CA doit être rattaché à tous les concepts rencontrés ou créés et dont l'intension est incluse dans $f(\{A\})$.

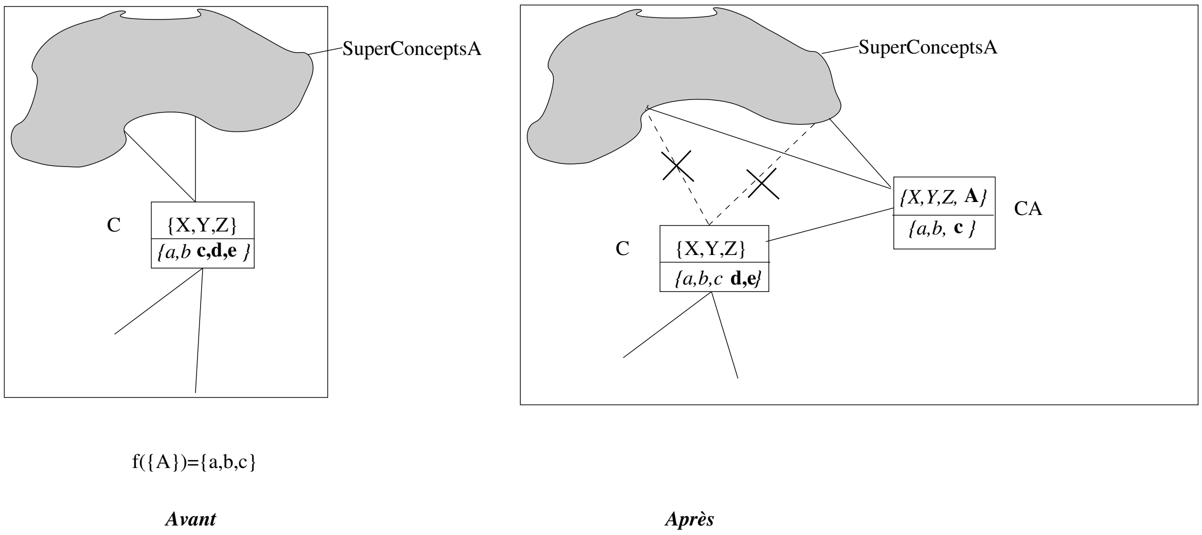


FIG. 4.25 – Cas 3-b de l'algorithme ARES-SHG

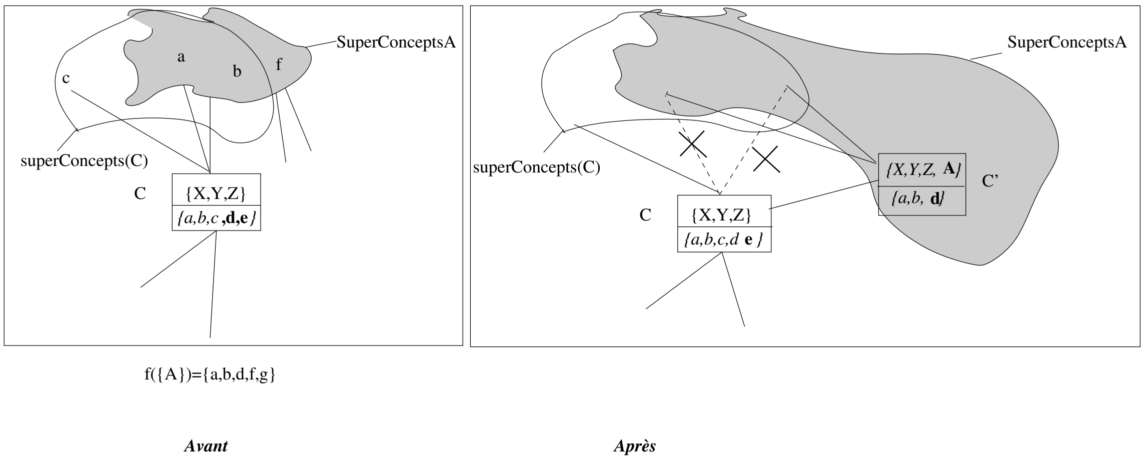


FIG. 4.26 – Cas 4 de l'algorithme ARES-SHG

Algorithmes connexes L'algorithme AISGOOD proposé dans [GC00] et dont la complexité est comparée avec celle d'ARES construit une sous-hiérarchie de Galois dans le cadre restreint où les classes initiales sont définies par les concepts feuilles. Le gain en complexité théorique d'AISGOOD est malheureusement dû à cette restriction, et nous ne pouvons ainsi pas réellement en déclarer un meilleur que l'autre. Par contre ils s'adressent à des cas pratiques distincts ce qui peut être intéressant.

Indications sur les complexités théorique et pratique Dans [DDHL94a], la complexité théorique a été évaluée dans $\mathcal{O}(|E|(\omega p+m))$, où ω est le nombre maximum de classes incomparables (largeur de la hiérarchie résultat), p le nombre maximum de propriétés pour une classe et m le nombre d'arcs.

Dans le schéma que nous avons proposé, la complexité pratique est améliorée par diverses propositions dont certaines sont inspirées par l'algorithme AISGOOD :

– seules les classes qui ont des référents communs avec la classe ajoutée sont explorées ;
– les sous-classes de A ne déclarant rien de commun avec A sont traitées sans refaire un parcours supplémentaire ;

ARES(A)-SHG *Soit A l'objet formel à ajouter, donné avec $f(\{A\})$*

(1) $superConcepts_A = \emptyset$
(2) $Propre_A = f(\{A\})$
(a) $ADefini = false$; $subIm_A = \emptyset$
(3) **pour** chaque concept C (t.q. $CP(C, A) \neq emptyset$) dans une extension linéaire de $>_H$ **faire**
(4) Calcul de $EP = EP(C, A)$, $RC = RC(C, A)$ et $RA = RA(C, A)$
(5) **si** $RC = RA = \emptyset$ **alors** *(Cas 1 - C définit A)*
ARES(A)-SHG-CAS 1
(7) **finSi**
(8) **si** $RC = \emptyset$ et $RA \neq \emptyset$ **alors** *(Cas 2 - A est dans l'extension de C)*
ARES(A)-SHG-CAS 2
(11) **finSi**
(12) **si** $RC \neq \emptyset$ et $RA = \emptyset$ **alors** *(Cas 3 - le concept qui définit A est plus grand que C)*
ARES(A)-SHG-CAS 3
(21) **finSi**
(22) **si** $RC \neq \emptyset$ et $RA \neq \emptyset$ et $EP \neq \emptyset$ **alors** *(Cas 4 - factoriser C et A)*
ARES(A)-SHG-CAS 4
(31) **finSi**
(32) **finPour**
(33) **si not** $ADefini$ **alors**
(34) créer un nouveau concept CA
(h) $Ext_s(CA) = \{A\}$; $Ext(CA) = \{A\}$; $Int(CA) = f(\{A\})$; $Int_s(CA) = Propre_A$
(35) $supIm(CA) = Min(superConcept_A)$
(i) **pour** chaque $C \in Min(superConcept_A)$ **faire** $subIm(C) \cup = \{CA\}$
(36) **finSi**
(l) Détruire les concepts vides

FIG. 4.27 – Schéma de ARES-SHG

- on sort de l'algorithme dès que l'on rencontre A ;
- on n'explore pas la descendance des sous-classes de A les plus hautes.

4.3.3 Vers d'autres opérations d'évolution

Dans le cadre du projet MACAO, nous étudions de manière générale toutes les opérations de maintien d'une sous-hiérarchie de Galois : ajout/suppression de classe, de référent, de lien d'héritage, fusion de hiérarchies [DVH+02]. Ces opérations présentent une certaine dualité. Par exemple, d'un point de vue théorique, l'ajout d'un nouveau référent à une ou plusieurs classes (ajout d'une colonne dans la matrice) est l'opération duale de l'ajout d'une classe. Cette étude a été commencée par H. Leblanc qui a spécifié certaines opérations lors de sa thèse et se poursuit actuellement par les travaux de Jean Villerd [Vil02] sous la direction de M. Dao. Un algorithme d'ajout de référent est en cours de développement dans Objecteering. Comme le montre le travail de Jean Villerd, il est aussi important d'étudier les aspects algorithmiques que de proposer une stratégie d'interaction avec l'utilisateur d'un environnement tel qu'Objecteering.

ARES(A)-SHG-CAS 1
C a pour intension $f(\{A\})$

(b) $Ext(C) \cup = \{A\}$; $Ext_s(C) \cup = \{A\}$
(6) fin de l'algorithme

FIG. 4.28 – Schéma de ARES-SHG-CAS1

ARES(A)-SHG-CAS 2
C plus grand que le concept d'intension $f(\{A\})$

(9) $superConcepts_A \cup = \{C\}$
(c) $Ext(C) \cup = \{A\}$
(10) $Propre_A \backslash = Int_s(C)$

FIG. 4.29 – Schéma de ARES-SHG-CAS2

4.3.4 Conclusion sur les aspects algorithmiques

On peut toujours chercher de nouveaux algorithmes quoique ceux dont nous disposons sont en pratique suffisamment rapides pour les applications actuelles. Une étude intéressante à mener consisterait à étudier l'apport en complexité pratique d'une réduction de la relation avant construction. La complexité théorique mériterait d'être réétudiée en utilisant d'autres paramètres tels que la taille de la relation (nombre de couples).

Une autre étude certainement instructive considèrerait les algorithmes de construction du treillis de Galois [Bor86, Gué90, Bor92, KO01] pour (1) voir si construire la sous-hiérarchie de Galois puis calculer les sommets du treillis à partir des fermés de la sous-hiérarchie serait profitable au moins en pratique car certains sommets absents de la sous-hiérarchie de Galois peuvent avoir un grand intérêt sémantique (2) faire un rapprochement entre les principes généraux des algorithmes.

4.4 Application au calcul d'interfaces Java

Les travaux sur le calcul automatique d'interfaces Java ont été réalisés, à la charnière entre les deux projets (CNET et MACAO), en collaboration avec H. Leblanc (cette étude est une partie de sa thèse) et M. Dao.

4.4.1 La notion d'interface en Java

Comme l'illustre la figure 4.32, les interfaces Java permettent de définir des types sans les implémenter. Elles se composent de méthodes publiques abstraites (décrites uniquement par leur signature) et de variables de classes constantes (introduites par le mot-clef `static`). Elles s'organisent au sein de hiérarchies d'héritage multiple (lien `extends`) tandis que les hiérarchies de classes Java (également lien `extends`) sont réduites à de l'héritage simple. Une classe peut implémenter une ou plusieurs interfaces (lien `implements`). Un grand

ARES(A)-SHG-CAS 3

C *plus petit que le concept d'intension* $f(\{A\})$
(13) **si not** $ADefini$ **alors**
(14) créer un nouveau concept CA
(d) $ADefini = true$
(e) $Ext_s(CA) = \{A\}$; $Ext(CA) = \{A\}$; $Int(CA) = f(\{A\})$; $Int_s(CA) = Propre_A$
(15) **si** $EP \neq \emptyset$ **alors**
(15) $Int_s(C) \backslash =_a f(\{A\})$
(f) **si** $Int_s(C) = Ext_s(C) = \emptyset$ **alors** $Vides \cup = \{C\}$
(g) **finSi**
(16) $supIm(CA) = Min(superConcepts_A)$
(17) **pour** chaque $C' \in Min(superConcept_A)$ **faire**
(h) $subIm(C') \cup = \{CA\}$
(18) $subIm(C') \backslash = \{C\}$
(19) **finPour**
(20) **finSi**
(i) $subIm(CA) \cup = \{C\}$; $Ext(CA) \cup = Ext(C)$
(21) $supIm(C) \cup = \{CA\}$
(22) $supIm(C) \backslash = supIm(CA)$
(23) Eliminer la descendance de C de l'extension linéaire

FIG. 4.30 – Schéma de ARES-SHG-CAS3

intérêt des interfaces Java est que ce sont réellement des types du langage Java, donc utilisables pour typer les variables et les paramètres, ce qui permet d'écrire du code plus générique. Ainsi la définition d'une interface décrivant les objets comparables, c'est-à-dire munis de méthodes de comparaison, autorise l'écriture d'une méthode générique de tri qui s'applique à tout tableau d'objets comparables.

4.4.2 Intérêt de la construction automatique des interfaces

Dans un processus standard idéal de développement, qui les considère comme des types, les interfaces devraient être conçues et développées avant les classes. Mais les pratiques actuelles de développement sont loin de cet idéal. Souvent peu d'interfaces sont développées, elles le sont en même temps que les classes et les types sont finalement dispersés au sein des classes et des interfaces. La contrainte d'héritage simple entre classes imposée par Java amène à développer des interfaces « utilitaires », qui ne sont pas forcément dictées par une modélisation du domaine d'application mais par des besoins de codage. Il nous semble vraiment important dans la modélisation objet de séparer clairement les interfaces, qui devraient correspondre aux spécifications abstraites du problème, des classes dont le rôle est l'implémentation. Si le code est déjà écrit sans prendre en compte ce point de vue, il peut être utile de procéder à une rétro-conception qui extraie les interfaces du code. Ce type de rétro-conception peut être fort utile pour valider un développement, par comparaison entre la spécification faite lors de la conception et la spécification telle qu'elle apparaît dans le code (récupérée par extraction).

Pour résumer voici les points que l'on souhaite améliorer par cette construction auto-

Ares(A)-SHG-CAS 4

C incomparable avec le concept d'intension $f(\{A\})$

(23) créer un nouveau concept C'
(24) $Int_s(C') = EP$
(j) $Int(C') = CP$; $Ext_s(C') = \emptyset$; $Ext(C') = Ext(C) \cup \{A\}$; $subIm(C') = \{C\}$
(25) $supIm(C') = Min(superConcepts(C) \cap superConcept_A)$
(26) **pour** chaque $C'' \in Min(superConcepts(C) \cap superConcept_A)$
(27) $\quad subIm(C'') \cup = \{C'\}$
(28) $\quad subIm(C'') \setminus = \{C'\}$
(29) **finPour**
(30) $supIm(C) \cup = \{C'\}$
(31) $superConcept_A \cup = \{C'\}$
(32) $supIm(C) \setminus = supIm(C')$
(33) $Int_s(C) \setminus = Int_s(C')$
(k) **si** $Int_s(C) = Ext_s(C) = \emptyset$ **alors** $Vides \cup = \{C\}$
(34) $Propre_A \setminus = Int_S(C')$

Fig. 4.31 – Schéma de Ares-SHG-cas4

matique.
- *Manque d'exhaustivité des types*
- *Séparation peu claire entre types et implémentation*
- *Modèle peu satisfaisant de l'application*

4.4.3 L'exemple des polygones

Nous illustrons le procédé sur un exemple présenté dans [HL00, Leb00] et que j'avais à l'origine imaginé à partir de la hiérarchie des polygones de Yafool [Duc88]. Nous partons d'une hiérarchie de classes Java dont nous conservons les signatures de méthodes publiques et les variables de classes constantes. Cette hiérarchie est présentée Figure 4.33. Nous pouvons y relever quelques défauts tels que la répétition de certaines signatures dans des classes incomparables (comme `side() :float`) ou l'absence de spécialisation entre `Square` et `Rhombus`, due à la contrainte d'héritage simple.

4.4.4 Mise en œuvre

H. Leblanc a implémenté en Java [LDH00, LDH01, Leb00] les différentes étapes de la construction des interfaces, principalement :
- utilisation de l'introspection Java pour produire une relation binaire liant les classes (objets formels) aux référents (signatures des méthodes publiques et variables de classe constantes) ; elle est présentée pour l'exemple dans la table 4.34 ;
- application de l'algorithme Ceres ;
- parcours de la sous-hiérarchie produite pour interpréter les concepts comme des interfaces Java, et génération du code Java correspondant.

Dans le cas des polygones, nous obtenons la hiérarchie d'interfaces de la figure 4.35, qui est liée à la hiérarchie de classes comme le montre la figure 4.36.

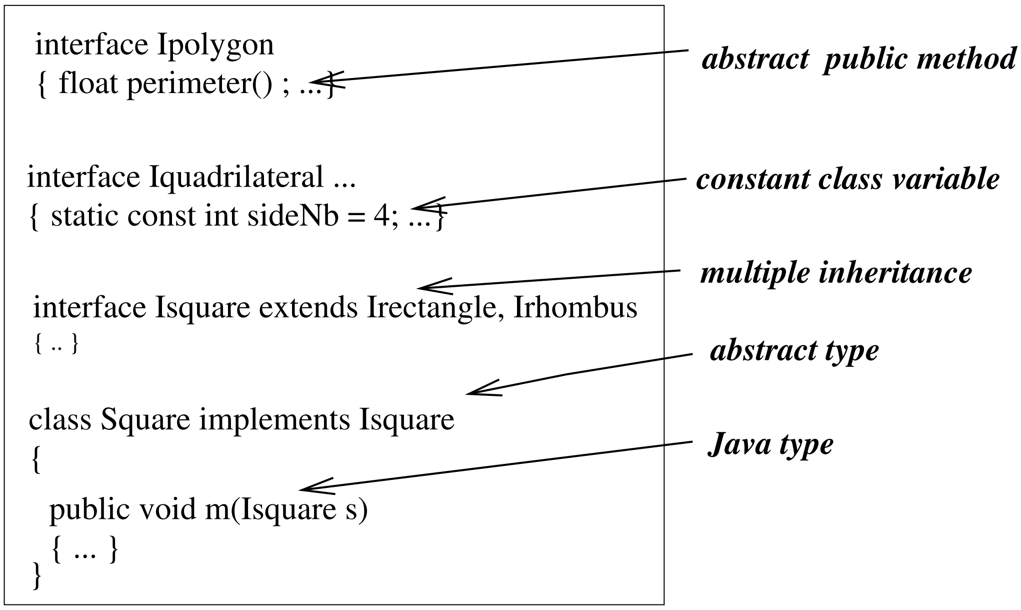


FIG. 4.32 – Interfaces en Java

4.4.5 Résultats des expériences effectuées

Polygones Les résultats obtenus sur l'exemple des polygones révèlent les principaux apports de la technique :

- les caractéristiques répétées sont factorisées (par exemple, `side` est factorisée dans l'interface `IRegularSidePolygon`) ;
- des liens de spécialisation manquant (comme entre `Square` et `Rhombus`) sont ajoutés ;
- des généralisations intéressantes sont mises à jour, telles que `IRegularPolygon` qui représente les polygones dont les angles (resp. les côtés) sont égaux.

Le procédé crée également des abstractions moins importantes, telles que `ITwoFloat-DeterminedPolygon`. Dans cet exemple d'école, cependant, on ne découvre pas de véritable anomalie dans les abstractions découvertes. La construction de l'exemple en est la cause, car il provient de la mise à plat d'une hiérarchie d'héritage multiple (celle de Yafool). En particulier le vocabulaire est uniforme et permet de repérer facilement les redondances, mais surtout les concepts sont décrits de manière cohérente. Par exemple, la méthode `side` existe dans toutes les classes où elle présente un intérêt. C'est évidemment une situation idéale. Sans détailler les autres expériences réalisées cette fois sur des applications en vraie grandeur, nous en donnons quelques traits marquants.

Pétra Pétra [dt97b, dt97a] est une plate-forme dédiée à la description et à la simulation de réseaux de télécommunication. Pétra était à l'origine conçue en OMT avec de l'héritage multiple et a une structuration en couches rassemblant des classes de différents niveaux d'abstraction tout à fait particulière [Dao99], allant de la couche des classes décrivant la notion de graphe, passant par la couche de description d'un réseau général, spécialisée par la couche des classes décrivant la simulation d'un réseau télécom pour n'en citer qu'une partie. La construction des interfaces s'est faite en analysant le code Java de Pétra. Dans le résultat, il a été intéressant de retrouver la structure d'héritage multiple du niveau analyse. De manière générale, des interfaces correspondant à toutes les classes du modèle d'analyse de Pétra apparaissent dans le résultat, ce qui permet de penser que le code reflète correctement le modèle d'analyse (sous la contrainte d'héritage simple). Quelques liens de

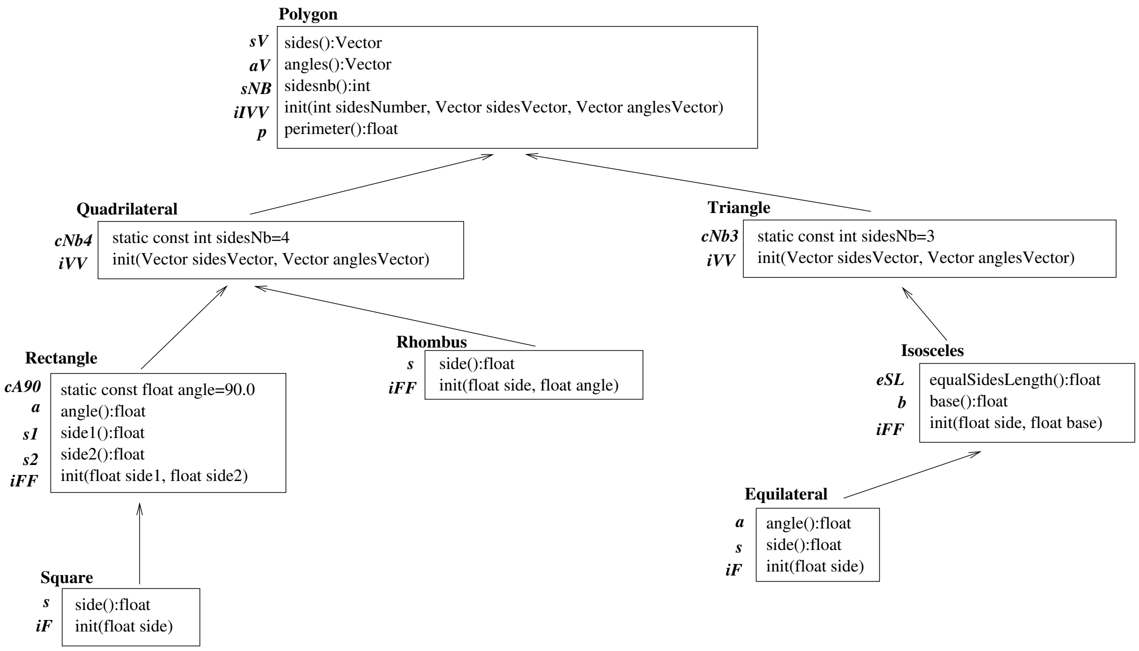


Fig. 4.33 – Extrait des classes décrivant les polygones

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	iIVV	SNB	p	sV	aV	iVV	iFF	C4	s	C3	a	S2	S1	eSL	c90	b	iF
0 -P	X	X	X	X	X												
1 -T	X	X	X	X	X	X				X							
2- Q	X	X	X	X	X	X		X									
3-Rh	X	X	X	X	X	X	X	X	X								
4-I	X	X	X	X	X	X	X			X				X		X	
5-Re	X	X	X	X	X	X	X	X			X	X	X		X		
6-E	X	X	X	X	X	X	X		X	X	X			X		X	X
7-S	X	X	X	X	X	X	X	X	X		X	X	X		X		X

Fig. 4.34 – *Binary relation "owns-property"*

spécialisation dans le résultat sont incorrects car les classes ne contiennent pas certaines informations qui auraient permis d'éviter de les construire. Par exemple, les signatures peuvent ne pas être assez précises soit parce que le modèle de redéfinition des méthodes est invariant en Java, soit parce que l'absence de types paramétrés ne permet pas de différencier entre une énumération d'objets d'un type X et une énumération d'objets d'un type Y.

API Java Nous avons étudié également la construction des interfaces de l'API Java. Cette expérience avait pour but de vérifier la faisabilité en temps et mémoire de l'outil (mentionné dans le chapitre précédent), mais également d'évaluer la quantité et la qualité des interfaces créées.

D'un point de vue quantitatif, les paquetages principaux ont été analysés (`lang`, `io`, `util`, `awt`, `swing`). Dans le meilleur des cas, le nombre d'interfaces est inférieur au nombre de classes (paquetage `swing`), dans le pire des cas, le volume d'interfaces créé est de 40% plus élevé que le nombre de classes (paquetage `awt`). Sur ces exemples, on ne note donc pas d'explosion dramatique du nombre de concepts créés.

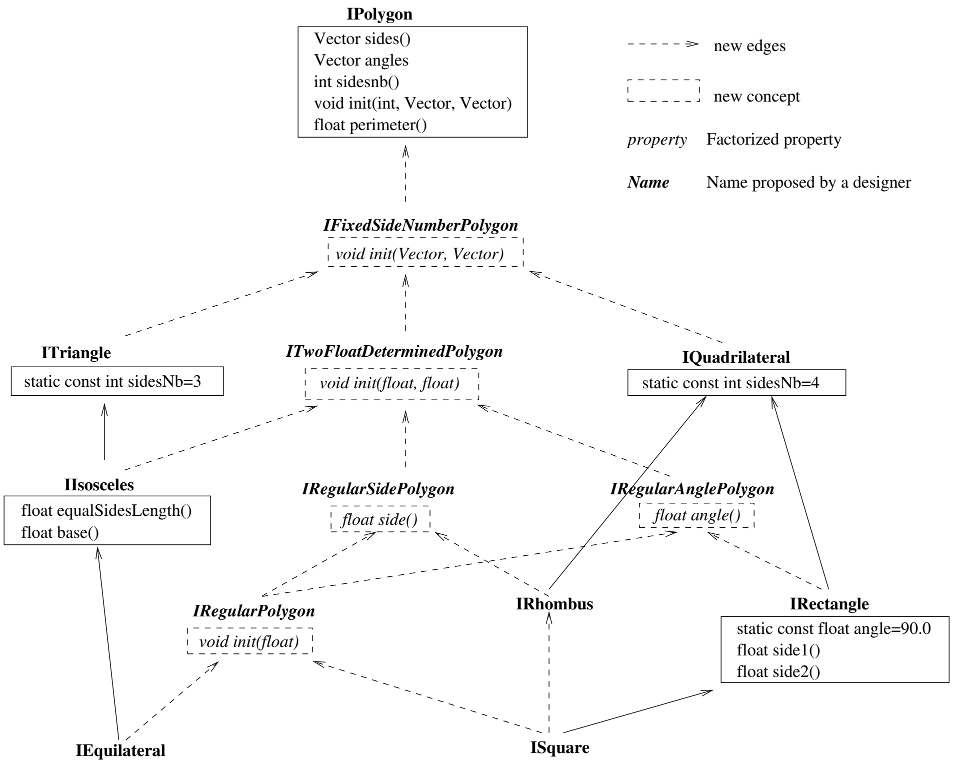


FIG. 4.35 – Interfaces construites pour les polygones

L'analyse de la qualité n'a porté que sur le paquetage décrivant les entrées/sorties (`io`). Cette analyse avait pour ceci de différent de la précédente (API de Pétra) que cette fois nous n'avions pas d'interlocuteur pour discuter de nos conclusions.

Elle nous a permis de mettre en évidence les défauts suivants. Dans le cas des flots d'entrée/sortie, une symétrie peut être attendue entre les concepts décrivant les flots en entrée et les concepts représentant les flots en sortie, comme nous la constatons dans l'implémentation des flots en C++. La hiérarchie des interfaces générée ne montre pas cette symétrie, et une analyse plus fine montre que cela vient en particulier d'une description incohérente (non systématique) des concepts. Les interfaces de Java sont rarement retrouvées. L'ensemble de propriétés d'une interface I de Java est en effet généralement complété par l'opération de fermeture, et l'interface I' ainsi construite peut être spécialisée par plus de classes que I. Cela peut indiquer que ces classes auraient dû implémenter (par le lien `implements`) I dans la hiérarchie d'origine et on ne peut guère ici incriminer la contrainte de l'héritage simple pour expliquer ce défaut d'implémentation. Plus vraisemblablement on peut suspecter un défaut de conception.

4.4.6 Conclusion et perspectives

Pour résumer les apports concrets de la construction automatique des hiérarchies d'interfaces, en plus d'offrir un modèle plus clair de l'application et une bonne séparation des types et des classes, celle-ci permet :

- une évaluation de l'écart entre l'implémentation et la spécification (qui montre par exemple dans le cas de Pétra que la spécification est respectée) ;
- dans le cas de l'API Java, la mise en évidence de descriptions non systématiques ou incomplètes des concepts (méthodes portant des noms/signatures différents ; opé-

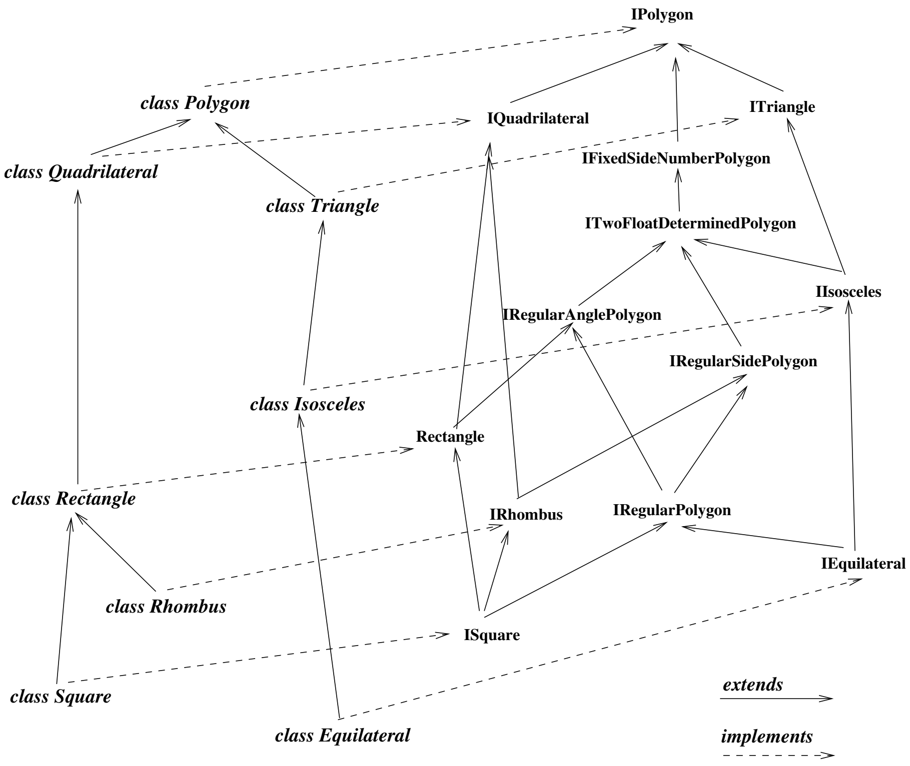


FIG. 4.36 – Classes et interfaces

rations structurées différemment amenant un découpage en méthodes différent ; méthodes manquantes) ; en particulier dans le cas de Java cela se traduit par un manque de symétrie dans les concepts construits (mais ceci est assez spécifique aux entrées/sorties) ;
- des propositions pour compléter les interfaces existantes par l'opération de fermeture ;
- des suggestions pour compléter les liens d'implémentation classe-interface.

Les analyses ont montré un certain nombre de points à améliorer dans notre approche et qui ouvrent ainsi la voie à de futurs travaux.

- La factorisation effectuée par la sous-hiérarchie de Galois impose deux types de contraintes qui peuvent être trop fortes dans certains cas concrets.
  - la notion de fermeture impose que les propriétés appartenant à un fermé *simplifié* soient factorisées ensemble (théoriquement ces propriétés sont « équivalentes » car l'une d'elles suffit à créer le concept). Pourtant ces propriétés peuvent ne pas avoir de cohésion sémantique. Par exemple, un fermé simplifié pourrait être composé de propriétés (méthodes ou variables de classes) ayant trait à l'insertion dans un flot (groupe M_1) et de propriétés correspondant à l'extraction (groupe M_2) depuis un flot. Il peut être intéressant de séparer ces propriétés qui participent de différents « aspects » (sauvegarde vs. saisie).
  - de manière plus générale, les propriétés appartenant à un fermé apparaissent ensemble. Or le fait que la présence d'un groupe de méthodes M_2 implique la présence d'un autre groupe de méthodes M_1 n'assure pas non plus que ces deux groupes de méthodes forment ensemble un concept cohérent.

  Dans les deux cas, les concepts pourraient être éclatés, par exemple sur la base de mesures de cohésion (voir Figure 4.37). On peut penser que le résultat serait plus stable car il y a toutes les chances que l'un des groupes de méthodes apparaisse plus

tard (lors d'ajout de nouveaux concepts) indépendamment de l'autre. En somme, on factorise pour préparer l'avenir.
– Une limite forte de la méthode est due à l'absence de finesse des mécanismes de contrôle d'accès statique en Java. Dans de nombreux cas, le développeur n'a pas d'autre choix que de déclarer publique une méthode alors que celle-ci ne devrait être accessible que dans un contexte limité (seulement quelques classes de deux paquetages par exemple). Une documentation du code analysable automatiquement, par exemple à l'aide de graphes d'accès [AH02], pourrait nous permettre de résoudre ce problème.
– L'invariance dans la redéfinition des méthodes et l'absence de types paramétrés en Java limite la précision de la description. Une analyse du code permettrait vraisemblablement de découvrir les types réellement passés en paramètres aux méthodes ou dans les structures de données et augmenterait la qualité des factorisations effectuées.
– Une amélioration facile à mettre en place consiste à prendre en compte les exceptions dans les signatures pour en déduire des propriétés génériques et affiner ainsi les interfaces construites.
– Ceci nous amène à notre dernier point. La notion de coréférent n'était pas prise en compte à l'époque de ce travail, il est donc clair qu'il serait nécessaire de revoir une partie de l'outil en amont de l'algorithme CERES, de manière à intégrer quelques heuristiques plus fines que la simple reconnaissance de l'égalité de signatures.

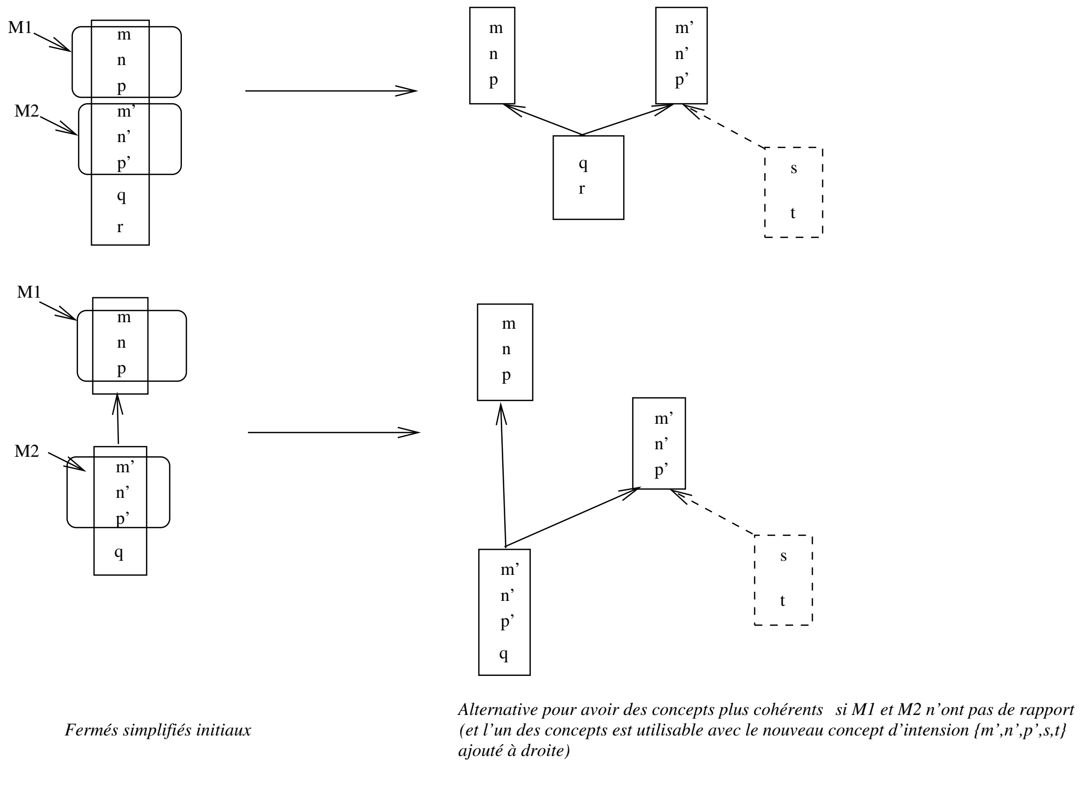


FIG. 4.37 – Alternatives à la factorisation trop compacte de la sous-hiérarchie de Galois

Cette proposition de construction automatique d'interfaces pourrait avoir une application dans un autre domaine qui est celui du « middleware », une forme de logiciel qui sert à connecter différentes applications, en distribuant des services selon la métaphore « client-serveur ». Certaines des technologies suivent une approche orientée objet et proposent l'utilisation d'interfaces très proches de celles de Java : COM/DCOM [Micb, Mica], Java RMI (*Remote call Invocation*) [Mic99] et CORBA (*Common Object Request Broker Archi-*

tecture) [Obj01]. Les interfaces IDL de CORBA par exemple, contiennent des attributs, des méthodes, des exceptions et admettent une organisation avec spécialisation multiple. L'utilisation de CERES dans ce contexte serait intéressante soit pour extraire et publier les services d'applications Java existantes, tout en donnant une meilleure vue conceptuelle de ces mêmes services, soit pour améliorer une hiérarchie d'interfaces déjà construite.

4.5 Classification de classes vs. classification d'instances

Nous présentons ici une étude de la sémantique de l'ajout de classe, détaillée dans [Huc99], et qui est une réponse à une question posée par R. Ducournau dans le cadre de ses réflexions sur la classification [Duc96]. D. Bardou, T. Libourel et surtout A. Napoli en ont également lu et discuté des versions et ont aidé à clarifier la formulation du problème.

Elle déplace le cadre de travail vers la phase d'exploitation d'une hiérarchie et se concentre notamment sur les conséquences de la présence d'instances.

Informellement il s'agit de s'intéresser aux retentissements que peut avoir sur d'éventuelles instances l'ajout d'une classe par un algorithme tel qu'ARES. Répondre à ce problème implique de définir ce qu'est un système bien formé composé de classes et d'instances. Nous proposerons une définition dont nous pensons qu'elle respecte les hypothèses et la pratique de la programmation par objets et que la composante classe est une sous-hiérarchie de Galois (section 4.5.1). Nous étudions ensuite l'opération d'évolution par ajout de classe et concluons curieusement par un résultat négatif sur sa correction (section 4.5.2) : alors que cette opération d'ajout est censée préserver les services rendus par les classes, elle ne préserve pas une classification correcte des instances dans le cas général. Nous élargissons le problème en montrant comment pourraient être contournés les problèmes du modèle sur lesquels nous butons et surtout nous avançons l'idée que cette étude devrait être étendue et généralisée (section 4.5.5).

4.5.1 Système classificatoire de Galois pour les langages à objets

Nous utilisons une version simplifiée et adaptée de la notion de système classificatoire proposée dans [Duc96]. Dans un tel système, les principaux constituants sont une hiérarchie de classes ; une description en extension des classes qui leur associe un ensemble d'instances ; une description en intension des classes par un ensemble de propriétés (ou référents) ; une description des instances en conformité avec les classes.

Les langages à objets les plus courants (C++, Eiffel, Java), imposent quelques contraintes sur cette définition générale.

- *classification « à la main »* Une instance n'est pas classée automatiquement, elle est créée explicitement dans le programme par « instanciation » de sa classe.
- *monoinstanciation* Une instance n'est rattachée directement qu'à une classe, autrement dit, pour toute instance i, l'ensemble des classes C qui possèdent i dans leur extension a un plus petit élément que nous appellerons la classe de i.
- *pérennité de l'attachement* Une instance ne peut être changée de classe une fois créée (sa classe fait partie de son identité comme le montre parfaitement bien le système de nommage des instances en Java).
- *description* Une instance est seulement décrite par un vecteur de valeurs qui s'apparie précisément avec la suite des déclarations d'attributs de sa classe.
- *classes instanciables* Toute classe qui ne possède pas de méthode abstraite peut être instanciée.

– *Spécialisation des attributs* En Eiffel, le domaine d'un attribut est spécialisable explicitement (peut être restreint dans une sous-classe). En Java et C++, cette spécialisation est exprimée par des moyens détournés (commentaires, méthodes d'accès spécialisées en particulier pour vérifier une nouvelle contrainte sur le type).

Nous pouvons à présent avancer une définition de la notion de « système classificatoire pour les langages à objets ».

Définition 4.4 (Système classificatoire pour les langages à objets (OOL-sc))
Un système classificatoire pour les langages à objets se compose :

- *d'une hiérarchie de classes $\mathcal{H} = (\mathcal{C}_\mathcal{H}, \leq_\mathcal{H})$, $\mathcal{C}_\mathcal{H}$ est l'ensemble des classes.*
- *d'une composante intensionnelle basée sur un ensemble de propriétés ou référents ($\mathcal{R}_{ef}$) qui se divise en $\mathcal{R}_{ef_M}$ (référents de méthodes) et $\mathcal{R}_{ef_A}$ (référents d'attributs). Les référents d'attributs se présentent sous la forme $p : D$ où p est le nom, et D est une description. Nous considérons ici des types (au sens de la programmation), c'est-à-dire des domaines de valeurs munis d'opérations. Les descriptions seront manipulées dans la suite comme des ensembles. On note $\mathcal{D}$ l'ensemble des descriptions. L'ordre $\leq_{\mathcal{P}_{gen}}$ correspond ainsi, à nom égal, à l'ordre d'inclusion des domaines : $p : D_1 \leq_{\mathcal{P}_{gen}} p : D_2$ équivaut à $D_1 \subseteq D_2$.*
  Chaque classe est décrite par une intension (l'ensemble de ses propriétés), qui lui est associée par l'application décroissante
  $Intension \ : \ (\mathcal{C}_\mathcal{H}, \leq_\mathcal{H}) \longrightarrow (2^{\mathcal{R}_{ef}}, \subseteq)$
  Cette intension vérifie, pour tout référent $r \in \mathcal{R}_{ef}$, que si $r \in Intension(C)$ alors pour tout $r' \geq_{\mathcal{R}_{ef}} r$, $r' \in Intension(C)$. Dans le cas des référents d'attributs, cela revient à dire que si l'intension d'une classe contient $p : D$, elle contient aussi $p : D'$ avec $D \subseteq D'$.
- *d'une partie extensionnelle, constituée d'un ensemble d'instances $\mathcal{I}$ et d'une application croissante $Extension \ : \ (\mathcal{C}_\mathcal{H}, \leq_\mathcal{H}) \longrightarrow (2^{\mathcal{I}}, \subseteq)$. Cette application respecte la monoinstanciation et le fait que toute classe ne possédant pas de méthode abstraite est potentiellement instanciable.*
  Les instances sont décrites par un ensemble d'expressions. Les expressions de la forme $(i\ p\ v)$ associent, à une instance $i \in \mathcal{I}$ et un référent d'attribut $p \in \mathcal{R}_{ef_A}$, une valeur v qui appartient à une description $D \in \mathcal{D}$. Une instance ne peut avoir qu'une valeur pour une propriété donnée (mais bien sûr D peut être un produit cartésien).
  On définit la classe de rattachement d'une instance comme étant la plus petite classe C qui vérifie :
  [c1] pour tout référent d'attribut $p : D$ minimal de $Intension(C)$, il existe une expression $(i\ p\ v)$ avec $v \in D$
  [c2] réciproquement, pour toute expression $(i\ p\ v)$ avec $v \in D$, il existe $p : D'$ référent d'attribut minimal dans $Intension(C)$ avec $D \subseteq D'$
  Dans un système bien formé la plus petite classe C (pour $\leq \mathcal{H}$) qui possède une instance i dans son extension est sa classe de rattachement (elle doit exister)

Nous nous essayons à présent à définir un système classificatoire de Galois. Pour respecter la pratique du développement en génie logiciel, nous imaginons qu'un tel système est obtenu à partir d'un ensemble de classes $\mathcal{C}$, proposées par un concepteur ou un programmeur, ensemble qui est ensuite organisé/réorganisé de manière à être une sous-hiérarchie de Galois.

Définition 4.5 (Système classificatoire de Galois (Galois-SC)) *$SCG(\mathcal{C})$ est un*

Galois-SC *construit sur un ensemble de classes* $\mathcal{C}$ *si* $SCG(\mathcal{C})$ *est un* OOL-SC *tel que la composante hiérarchie* $(\mathcal{C}_{\mathcal{H}}, \leq_{\mathcal{H}})$ *soit consistante avec la sous-hiérarchie de Galois construite sur la relation binaire* $\mathcal{R}$ *incluse dans* $\mathcal{C} \times \mathcal{R}_{ef}$ *et définie par* $(C, r) \in \mathcal{R}$ *si* $r \in Intension(C)$.

4.5.2 Evolution par ajout de classe

L'opération d'ajout de classes dans un tel système (où nous rappelons que les extensions ne peuvent être modifiées du fait de l'hypothèse de pérennité de l'attachement) ne modifie que la composante hiérarchie et la composante intensionnelle. Les extensions restent inchangées. En pratique, il faut imaginer que l'on dispose d'une hiérarchie de classes instanciées, et que ces instances sont enregistrables, en mémorisant leur classe, dans un fichier ou une base pour sauvegarde et recréation. La hiérarchie de classes évolue pour intégrer une nouvelle classe tout en préservant la propriété de sous-hiérarchie de Galois. Les classes de $\mathcal{C}$ sont toutes définies par un concept de la sous-hiérarchie de Galois, que ce soit avant ou après l'ajout. Pour ces classes, l'extension peut être recréée à partir de la base d'instances pour préserver la pérennité de l'attachement. Il est peu envisageable dans le modèle des langages à objets de pratiquer une vraie classification des instances, car les valeurs qu'elles contiennent ne sont pas très informatives : il faut classer avec le dictionnaire des variables d'instances pour obtenir une classification porteuse de sens.

DÉFINITION 4.6 (AJOUT D'UNE CLASSE A DANS UN *Galois-SC*)

ÉVOLUTION DE LA HIÉRARCHIE *Etant donné un* Galois-SC $SC_1(\mathcal{C}_1)$*, l'opération d'ajout d'une classe A (pour laquelle on ne donne que son intension, laquelle vérifie la spécialisation des référents) consiste à produire le* Galois-SC $SC_2(\mathcal{C}_2)$*, avec* $\mathcal{C}_2 = \mathcal{C}_1 \bigcup \{A\}$.
TRANSMISSION DES EXTENSIONS *Soient* $SC_1(\mathcal{C}_1) = (\mathcal{C}_{\mathcal{H}_1}, \leq_{\mathcal{H}_1})$ *et* $SC_2(\mathcal{C}_2) = (\mathcal{C}_{\mathcal{H}_2}, \leq_{\mathcal{H}_2})$*. Soit* $f : \mathcal{C}_{\mathcal{H}1} \longrightarrow \mathcal{C}_{\mathcal{H}2}$*, qui associe à une classe* $C_1 \in \mathcal{C}_{\mathcal{H}1}$ *la classe* $C_2 \in \mathcal{C}_{\mathcal{H}2}$ *telle que* $Int(C_1) = Int(C_2)$*, si elle existe. Pour chaque classe* $C_1 \in \mathcal{C}_{\mathcal{H}1}$ *qui a une image par* f*,* $Extension(f(C_1)) = Extension(C1)$.

Nous allons, dans les deux sections qui suivent préciser l'effet de cette opération sur les classes, puis montrer que le système obtenu n'est pas correct du point de vue des instances.

Evolution des classes

Nous identifions trois cas essentiels :
- *Disparition de classe*
- *Création de classe*
- *Modification des intensions propres, héritées et déclarées*

Nous rappelons les applications suivantes qui concernent l'intension d'une classe C :
- $propre_r$ ou ici $propre$ car il n'y a pas d'ambiguïté, associe à C les référents qui n'apparaissent pas dans les super-classes ;
- $declar_r$ ou ici $declar$ associe à C les plus petits référents de l'intension propre ;
- $herit_r$ ou ici $herit$ associe à C les référents qui apparaissent dans les super-classes de C.

Disparition de classe Les classes de $\mathcal{C}_{\mathcal{H}_1}$ qui disparaissent sont les classes sur lesquelles f n'est pas définie, ce qui se produit lorsque les conditions suivantes sont réunies :
- $C \notin \mathcal{C}_1$ (c'est une classe de factorisation),
- $Intension(C)$ et $Intension(A)$ sont incomparables pour l'inclusion,
- $declar(C) \subset Intension(A)$.

Création de classe Une nouvelle classe est ajoutée pour toute classe C telle que $propre(C) \cap Intension(A) \neq \emptyset$, mais $Intension(C) \not\subset Intension(A)$ (ni l'inverse). Cette nouvelle classe a pour intension $Intension(C) \cap Intension(A)$.

Modification des intensions propres, héritées et déclarées Les classes pour lesquelles f est définie ne changent pas d'intension, mais les parties propres, déclarées et héritées peuvent être modifiées. Soit une classe $C \in \mathcal{C}_1$ qui a une image par f :

- si $Intension(C) \subset Intension(A)$ alors $declar(C) = declar(f(C))$ et $propre(C) = propre(f(C))$;
- sinon $declar(f(C)) = declar(C) \setminus Intension(A)$ et $propre(f(C)) = propre(C) \setminus Intension(A)$.

4.5.3 Effet sur les instances

L'opération d'ajout de classes ainsi définie serait également correcte du point de vue des instances si toute instance de I était toujours, après l'ajout, dans l'extension d'une classe qui vérifie la condition de rattachement.

Or deux problèmes essentiels peuvent se produire dans le système résultat :

- la classe de i n'a pas d'image par f,
- la plus petite classe C qui possède i dans son extension n'est pas la classe de rattachement (elle n'est plus la plus petite classe vérifiant [c1] et [c2]).

Nous illustrons l'un de ces problèmes sur un exemple inspiré de la botanique [IS88], les autres situations typiques sont présentées dans [Huc99]. La figure 4.38 montre deux attributs génériques d et f décrivant respectivement quelques formes de tiges et de fleurs.

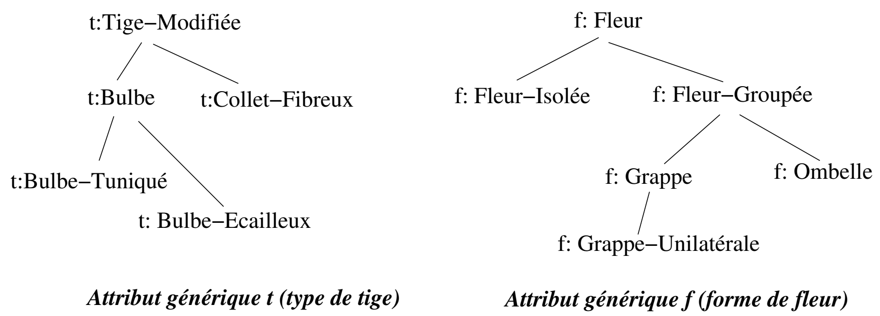


FIG. 4.38 – Deux attributs génériques décrivant les plantes

On découvre ensuite (partie gauche de la Figure 4.39) le schéma d'un premier système classificatoire. La hiérarchie de classes a pour racine une classe C_1 décrivant les plantes à tige modifiée et dont les fleurs sont en grappes. Cette classe est spécialisée d'un côté par les plantes à bulbes écailleux (classe C_2) et de l'autre par les plantes dont la grappe de fleurs est unilatérale au sens où toutes les fleurs apparaissent du même côté de la tige (classe C_3). Les référents qui apparaissent en gras sont déclarés par les classes, tandis que ceux qui sont en italiques correspondent à la partie propre de l'intension mais ne sont pas minimales dans l'ordre de la propriéte générique.

Trois instances ont été créées à partir de ces classes :

- un *muscari* dont les fleurs sont en grappe (non unilatérale) et la tige est un bulbe tuniqué (donc une tige modifiée) ;
- un *lis de mer* dont les fleurs sont groupées et qui possède un bulbe écailleux ;
- un *dicentra*, plante à collet fibreux et dont les fleurs sont en grappe unilatérale.

Ne nous y trompons pas : un concepteur pourrait avoir créé des classes pour chacune de ces plantes (par exemple *la* classe des muscaris) mais nous supposons que ce n'est pas le cas ici (nous avons bien *un* muscari). On ne peut avoir dans un système une classe pour chaque instance, c'est pour cela que l'on regroupe les instances en classes, ce qui est fait nous l'avons vu *a priori* dans le développement par objets standard. Le choix dans cette première version du système classificatoire est donc de n'avoir que ces trois classes.

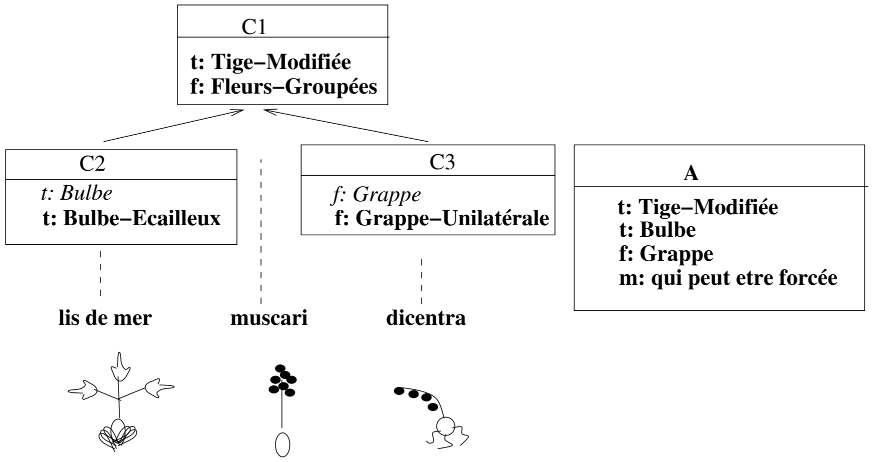


FIG. 4.39 – Système classificatoire 1 (à gauche) et classe A (à droite)

Dans cette même figure 4.39 apparaît, dans la partie droite, une classe A que l'on va ajouter au système et qui représente les plantes de la famille des jacinthes. Cette nouvelle classe a dans son intension les référents t : $Bulbe$, donc aussi t : $Tige - Modifiee$, f : $Grappe$ et m : $qui - peut - etre - forcee$. Le dernier référent appartenant à une nouvelle propriété générique m qui décrit les modes de culture et pour laquelle nous n'étudierons pas d'autres référents ici. La figure 4.40 présente le système classificatoire résultat. Après cette opération, l'instance *muscari* se trouve dans l'extension de la classe C_1, et C_1 vérifie toujours les conditions [c1] et [c2], mais elle n'est pas la plus petite classe qui les vérifie, car c'est aussi le cas des classes C_4 et C_5 qui ont été ajoutées pour respecter la structure de sous-hiérarchie de Galois.

4.5.4 Analyse et éléments de solution

Une interprétation est que la classification des classes ne peut pas être seule à évoluer. L'opération d'ajout de classe doit s'accompagner d'une évolution conjointe des instances.

Un premier élément important de la solution est évidemment de conserver les classes instanciables, cela évite les problèmes liés à la disparition de classes créées pour des besoins de factorisation, instanciées, puis détruites par un ajout de classe ultérieur car elles ne déclarent plus de propriété (et ne représentent aucune des classes de $\mathcal{C}_1$).

Une deuxième approche consiste à doter le système d'une possibilité de migration des instances. Cette migration doit s'accompagner d'une recherche de la classe de rattachement appropriée pour chaque instance.

Mais la migration n'est pas toujours une solution car parfois il n'existe plus de classe de rattachement pour i :

- soit aucune classe ne possède toutes les propriétés de i (aucune classe ne vérifie la condition [c2]),
- soit plusieurs classes vérifient les conditions [c1] et [c2] (c'est le cas dans l'exemple ci-dessus pour C_4 et C_5).

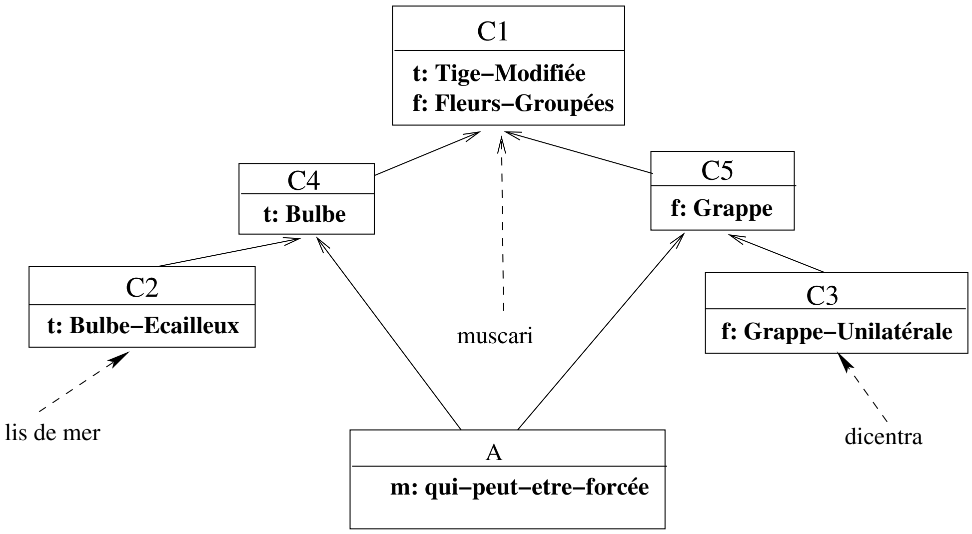


Fig. 4.40 – Système classificatoire 2 (après ajout de A)

Un certain nombre de classes supplémentaires peuvent donc devenir nécessaires, dans l'exemple des plantes une sous-classe de C_4 et C_5 serait nécessaire pour une bonne classification de *muscari*. Dans [Huc99] une notion de « signature » d'instance permet de caractériser la portion de hiérarchie dans laquelle une instance peut être migrée et détermine en cas de problème la classe qu'il est nécessaire d'ajouter. L'inconvénient d'une telle approche est le risque d'explosion du nombre de classes.

4.5.5 Conclusion

Pour une étude générale de l'évolution dans un système instancié La question que nous avons traitée ici peut se généraliser à toute opération d'évolution.

Problème 1 (Problème de l'évolution d'un système instancié) *Etant donné un système composé de classes et d'instances et une opération d'évolution T effectuée sur les classes, comment assurer que le système restera correctement formé ?*

La référence [Cas95] fait une synthèse d'un certain nombre d'opérations d'évolution dans les systèmes à objets (et particulièrement les bases de données) et de la manière dont elles ont été abordées dans la littérature. Mais ces évolutions ont un caractère spécifique : ce sont souvent des changements « ponctuels » de la hiérarchie, comme l'ajout, la suppression, le renommage de propriétés dans une classe, l'ajout et la suppression de classes, l'ajout de liens d'héritage, et surtout elles ne prétendent pas préserver dans leur intégralité les "services" que rendait la hiérarchie avant la transformation, il s'agit au contraire de modifier (au sens le plus large possible) ce que peut faire le système. Par ailleurs aucune de ces opérations n'est supposée préserver une structure de sous-hiérarchie de Galois. Il y a donc fort à faire dans ce cadre précis.

Une remise en cause du modèle de la programmation par objets Mais cette réflexion remet fondamentalement en cause le problème de la programmation par objets car il met en évidence ses faiblesses en matière de classification.

- *absence de classification des instances* On a fait l'hypothèse que les instances étaient bien placées au départ ce qui est déjà de la responsabilité du progammeur et

peut donc être erroné en pratique. S'il y a nécessité de migration, c'est entièrement à la charge du développeur.
- *monoinstanciation* Cette hypothèse est extrêmement forte, et implique l'explicitation de nombreuses classes. Deux types de solution permettraient de s'en dégager, la multi-instanciation (spécialement telle qu'elle est décrite dans [BG95] où une instance a plusieurs classes qui couvrent son ensemble de propriétés) ou la notion d'« instances sans classe » créées et décrites sans recourir à des classes, assorties d'un mécanisme de rattachement aux classes, telle que l'on peut la trouver dans les logiques de description [DLNS96].
- *appariement exact* Une instance doit posséder une valeur pour chaque attribut de sa classe et inversement chaque champ qui la constitue doit être associé à un attribut de sa classe.

Nous nous risquerons ici à nous interroger sur ces hypothèses : sont-elles réellement nécessaire dans le cadre du développement logiciel ou signalent-elles des points à améliorer ? Quoi qu'il en soit, elles mettent clairement en conflit la classification des classes et celle des instances dans des situations révélées mais non pas créées seulement par le respect de la structure de sous-hiérarchie de Galois (les modifications de classe pourraient se produire avec n'importe quel autre principe d'évolution des classes). Cette même structure aurait plutôt tendance à limiter les possibilités de transformation dans un cadre restreint et connu et à permettre de formuler des hypothèses sur les lieux possibles de migration des instances.

4.6 Métriques

Le projet MACAO nous a procuré une opportunité de pénétrer un domaine voisin de celui de la construction, qui est celui de l'évaluation. Notre rôle dans le projet consistait à réaliser une étude bibliographique sur le domaine des métriques dans le cadre du développement par objets (conception et programmation) et à identifier les métriques les plus pertinentes pour l'évaluation de la qualité des hiérarchies de classes.

4.6.1 Aspects des métriques dans le domaine des objets

Une analyse d'un grand nombre d'articles de la littérature (voir [DHLR01a] et le site du projet MACAO[5]) a été réalisée en collaboration avec M. Dao, T. Libourel et C. Roume. L'étude de métriques spécifiques au développement par objets a été amorcée assez tardivement par rapport aux avancées sur les langages. La première référence marquante du domaine est [CK91, CK94]. Elle propose essentiellement un ensemble de propriétés permettant de juger de la qualité des métriques adapté de [Wey80] ainsi que des métriques qui ont été par la suite très étudiées et dont de nombreuses variantes ont été proposées (voir [LK94] ou encore [BWDP98] qui rapporte ainsi une cinquantaine de métriques définies uniquement sur les classes). Les concepts fondamentaux introduits par [CK94] sont les notions de :
- *complexité*, mesurée par exemple par le nombre de méthodes par classe,
- *profondeur du graphe* d'héritage et *degré des sommets* qui permettent de vérifier que les propriétés ont des chances d'être réutilisées,
- *couplage* entre classes : de manière simplifiée une classe est couplée avec une autre si l'une des deux utilise un attribut ou appelle une méthode de l'autre,
- degré de *cohésion* : dans cette première approche, c'est essentiellement la cohésion d'un ensemble de méthodes qui est mesurée. La cohésion est mesurée en faisant

[5]http ://www.lirmm.fr/~macao

l'intersection des ensembles d'attributs utilisés respectivement par les méthodes.

En ce qui concerne la grande quantité de métriques proposées, une étude telle que [BWDP98] montre qu'elle est vraisemblablement excessive car les métriques sont souvent très corrélées et un petit sous-ensemble devrait être suffisant pour mesurer les défauts essentiels d'un logiciel.

Par ailleurs l'interprétation des résultats des métriques est un sujet très délicat. Les métriques indiquent des valeurs sur des aspects techniques du développement tels que le nombre de classes, le nombre d'appels de méthodes ou le nombre de liens d'héritage. Les différents acteurs (concepteur, développeur, chef de projet) sont pour leur part plutôt intéressés par des aspects de plus hauts niveau tels que le potentiel de réutilisabilité et d'extensibilité, la maintenabilité ou la portabilité. Le passage des métriques très techniques à l'interprétation de haut niveau est loin d'être résolu. Les travaux de notre partenaire dans le projet vont dans cette direction et l'on trouve également des études qui visent à établir l'influence de telle ou telle métrique sur un critère de haut niveau.

Notre étude révèle également qu'à l'heure actuelle, encore peu de métriques ont été conçues spécifiquement pour les phases d'analyse et de conception mais ce thème est en voie de développement, notamment grâce aux travaux présentés dans [Mar98, HSZKP02].

Pour conclure sur le point plus technique qui nous intéressait, c'est-à-dire les métriques liées à l'héritage, ce que nous avons trouvé ne forme pas un ensemble satisfaisant [DHLR01a]. La référence [LK94] propose ainsi des métriques telles que :

- HNL (Hierarchy Nesting Level) qui mesure à quelle distance une classe se trouve de la racine de la hiérarchie ; plus une classe est enfouie, plus il y a de chances qu'elle spécialise des méthodes ; selon les auteurs, en-dessous de 6 niveaux dans une hiérarchie, on n'a pas assez utilisé l'abstraction et la spécialisation, au-delà de 6 on risque d'avoir abusé de l'héritage pour représenter des relations entre classes qui ne sont pas de la spécialisation. *Outre que le seuil donné ne peut être fondé qu'expérimentalement, même s'il y a 6 niveaux, rien ne prouve qu'ils soient justifiés ?* ;
- Le pourcentage de classes abstraites (qui, toujours selon les auteurs, doit être proche de 10/15%) est un indice du fait que les concepteurs ont tenté de généraliser. *Mais il n'est pas certain que ces abstractions soient heureuses, ni exhaustives, ni bien placées* ;
- MUI (MUltiple Inheritance) qui compte le nombre de classes qui ont plusieurs super-classes directes (les auteurs conseillent de ne pas utiliser l'héritage multiple). *Là encore on ne regarde pas dans les faits si les relations entre classes sont correctes, on se contente d'argumenter que l'héritage multiple pose des problèmes techniques et serait plus difficile à comprendre* ;
- NMA (Number of Added Methods), qui devrait décroître en descendant dans les niveaux, peut avoir diverses interprétations : *on devrait ajouter des méthodes quand on spécialise, mais quelquefois une classe est ajoutée à des fins d'organisation, et par ailleurs une classe qui concrétise une classe abstraite peut, sans que cela soit une erreur de conception, n'avoir que des spécialisations des méthodes abstraites de sa super-classe.*

D'autres métriques nous ont paru plus intéressantes :

- NMO (Number of Overriden Methods) [LK94] compte le nombre de redéfinitions de méthodes dans une classe, sans comptabiliser les redéfinitions d'une méthode abstraite ou qui invoquent la méthode de la super-classe ; *une méthode redéfinie (dans ce sens précis) peut en effet indiquer que la classe n'est peut-être pas une spécialisation de sa super-classe* ;
- NMI (Number of Inherited Methods) [LK94] permet de mesurer dans quelle proportion l'héritage est utilisé ; *en effet, toute méthode héritée peut être considérée comme*

une méthode factorisée ;
- SIX (Specialisation IndeX) [LK94] a pour rôle de vérifier si une classe est bien placée (sous sa super-classe) ; pour cela, elle ne doit pas masquer ni refuser trop de propriétés héritées et plus elle est basse dans la hiérarchie moins elle doit ajouter de nouvelles méthodes ; une méthode de calcul proposée est $\frac{NMO*HNL}{total\ number\ of\ methods}$;
- PII (Pure Inheritance Index) [MHK99], qui serait un peu plus longue à définir, montre la capacité d'une classe à utiliser l'héritage suivant ce qu'elle préserve, ajoute ou redéfinit ;
- MIF/AIF (Method/Attribute Inheritance Factor) [eAM96] mesurent le taux de méthodes (resp. d'attributs) héritées, globalement sur une application ; *ces métriques globales permettent de mesurer la quantité et non la qualité de l'héritage [HCN98] et donc de comparer le degré d'utilisation de l'héritage de différentes applications.*

Notre étude nous a ainsi convaincu d'élaborer de nouvelles métriques répondant à plusieurs critères qui ne nous semblaient pas couverts par l'existant :
- étude systématique de la qualité de la factorisation et de la généralisation/spécialisation dans les hiérarchies d'héritage ;
- utilisation d'un « référentiel » permettant d'utiliser effectivement les résultats de ces métriques et d'éviter de tomber dans le travers de nombreuses études qui se cantonnent à donner des bornes fixes (voir ci-dessus la critique portée sur la limite à six niveaux d'une hiérarchie d'héritage) dont l'interprétation n'est pas claire.

4.6.2 Métriques pour évaluer la qualité de la factorisation et de la généralisation

Nous avons proposé (Hervé Leblanc ayant rejoint notre groupe à ce moment-là) une suite composée d'une trentaine de métriques spécifiquement dédiées à la mesure de la factorisation et de la spécialisation/généralisation qui est décrite dans [DHL$^+$01b, DHL$^+$01a, DHL$^+$02, Rou02a]. Luc Bernard-Gazagne a discuté certaines de ces métriques lors de son stage [BG02].

Dans ce travail, la sous-hiérarchie de Galois est considérée comme l'« idéal » en matière de factorisation et de généralisation/spécialisation. Les métriques sont conçues en référence à ce modèle et cherchent à mesurer une sorte d'écart entre la hiérarchie évaluée et la sous-hiérarchie de Galois.

Aperçu des métriques proposées

Les métriques portent sur des éléments de différents niveaux, les référents (propriétés), les propriétés génériques, les classes et la hiérarchie toute entière. Pour chacun de ces niveaux, nous donnerons ici un exemple de métrique pour illustrer la démarche employée. Ces métriques sont pour l'essentiel prévues pour avoir des valeurs idéales sur une sous-hiérarchie de Galois.

Au premier niveau, on effectue des mesures de redondances, de degré de factorisation et d'importance d'un co-référent dans la factorisation. L'exemple choisi est une métrique toute simple qui mesure le taux de redondance d'une propriété.

Metrique 4.1 (Taux de redondances d'un référent f (RR))

$$SoitRN(f) = |\{C \in H/f \in declar(C)\}| - 1 \tag{4.1}$$

$$RR(f) = \frac{RN(f)}{|\{C \in H/f \in Intension(C)\}|} \tag{4.2}$$

Pour mesurer la cohérence de la hiérarchie avec les propriétés génériques (et donc le fait que la généralisation/spécialisation effectuée dans la hiérarchie est convenable), on étudie le nombre de généralisation manquantes ainsi que l'accord entre les ordres de spécialisation dans les propriétés génériques $\leq_{\mathcal{P}_{gen}}$ et l'ordre de spécialisation entre classes $\leq_H$. Par exemple, une métrique est associée, pour une propriété générique $\mathcal{F}$ munie de l'ordre de spécialisation $\leq_{\mathcal{F}}$, au nombre de relations de spécialisation entre classes qui contredisent $\leq_{\mathcal{F}}$.

Metrique 4.2 (Nombre de relations d'héritage incorrectes (NIHR))

$$NIHR(\mathcal{F}) = |\{(Ci, Cj)/Ci <_H Cj, fi \in Int(Ci), fj \in Int(Cj),\ et\ fi \not\leq_{\mathcal{F}} fj)\}| \tag{4.3}$$

Avec Int diminutif pour $Intension$.

On peut s'intéresser ensuite aux métriques qui mesurent le rôle d'une classe dans la factorisation d'une propriété, d'une propriété générique ou plus généralement dans la factorisation. Nous proposons par exemple une mesure du rôle d'une classe dans la factorisation de toutes les propriétés. Cette mesure se base sur l'idée qu'une classe est responsable, vis-à-vis de ses sous-classes de toutes les propriétés de son intension. On cumule alors le nombre de fois où une sous-classe de C détient une propriété du fait de C (car C la lui transmet) sur le nombre total de relations d'appartenance d'une propriété à une classe.

Metrique 4.3 (Rôle d'une classe dans la factorisation (RCF))

$$RCF(C) = \frac{|Intension(C)| \times |\{C'/C' \leq_H C\}|}{|\mathcal{R}_{HF}|} \tag{4.4}$$

Au niveau de la hiérarchie, on peut utiliser diverses moyennes pondérées ou non des métriques des niveaux précédents. On peut également chercher à évaluer le degré de factorisation et de généralisation par rapport à une hiérarchie à plat ou étudier des propriétés de la sous-hiérarchie de Galois associée. Le degré de factorisation est ainsi mesurable en partant de la constatation qu'à chaque fois qu'une propriété est héritée, elle est factorisée.

Metrique 4.4 (Degré de factorisation d'une hiérarchie (LFH))

$$LFH(H) = \frac{|\{(C, f)/\ f \in herit(C)\ et\ f \notin declar(C)\}|}{|\mathcal{R}_{HF}| - |Feat(H)|} \tag{4.5}$$

$(C, f) \in \mathcal{R}_{HF}$ si $f \in Intension(C)$
$Feat(H)$ est l'ensemble des propriétés qui apparaissent dans la hiérarchie

Contextes de mesure

Une grande difficulté pour notre évaluation est que nous ne disposons généralement pas des référents ni *a fortiori* des propriétés génériques. Il faut là encore faire des hypothèses sur ce que seraient des propriétés génériques « modèles », respectant de bonnes règles de conception. Dans le cas de l'évaluation de hiérarchies codées dans un langage de programmation, en particulier, ces propriétés génériques doivent être construites en extrayant les co-référents du code puis en les interprétant pour produire et organiser des référents.

Une heuristique simple pour détecter des co-référents consiste à considérer que tous les attributs de même nom constituent une propriété générique, le type de l'attribut déterminant un référent. Les méthodes de même nom peuvent également constituer une propriété générique, d'autres informations telles que signature, liste d'exceptions, caractère abstrait/concret et code le cas échéant servant à ordonner les référents selon différentes heuristiques. Nous présentons ci-dessous un extrait des heuristiques utilisées :
- les référents d'attributs sont ordonnées d'après le type,
- une méthode abstraite ne spécialise jamais une méthode concrète,
- si m_1 et m_2 sont dans une même méthode générique, m_1 spécialise m_2 si tout co-référent associé à m_1 appelle l'un des co-référents associés à m_2 *via* le mot-clef `super` ou une construction assimilée,
- si une méthode générique ne comporte pas de référent correspondant à une forme abstraite, celle-ci est ajoutée.

Un *contexte de mesure* est un ensemble de telles heuristiques : il est important de souligner que les résultats des métriques ne mesurent rien d'autre que la cohérence de la construction de la hiérarchie vis-à-vis de ces hypothèses, donc vis-à-vis d'un ensemble choisi de règles de bonne conception.

4.6.3 Expérimentations

Les métriques ont été implémentées tout d'abord en Java par C. Roume qui a effectué des expériences sur les paquetages de Java JDK 1.3 [Rou01]. Puis L. Bernard-Gazagne [BG02] a transféré les plus importantes dans Objecteering et testé son développement sur Pétra et Résyn (logiciel développé au LIRMM et dédié à la manipulation de molécules chimiques).

Les expérimentations ont permis de donner des indications générales sur la qualité de la factorisation et de la généralisation (à interpréter dans le contexte de mesure choisi) mais également d'identifier les propriétés à mieux factoriser (recopie conforme du code déclarant un attribut jusqu'à six fois dans *java.io*), des méthodes héritées mais pourtant redéclarées dans Résyn (même signature/même corps), les classes fondamentales d'une hiérarchie (qui aident à en saisir rapidement l'architecture), des masquages systématiques de propriétés (dans Java), des méthodes concrètes spécialisées par des méthodes abstraites (ce qui peut être discutable) ou encore un certain manque de méthodes abstraites.

4.6.4 Conclusion

Cette question devrait être abordée très en détail dans la thèse de Cyril Roume. Les métriques peuvent être également associées à des opérations de réorganisation [SGM00] ou de transformation des hiérarchies [Rou02b]. Les concepts obtenus dans la sous-hiérarchie de Galois pourraient être mesurés en particulier pour en vérifier la cohésion. Enfin, l'utilisation pratique de ces métriques serait améliorée par la conception d'un outil capable de produire une synthèse des résultats assortie d'un ensemble de conseils de haut niveau pour améliorer la hiérarchie, pour suivre la démarche proposée dans [DMT01].

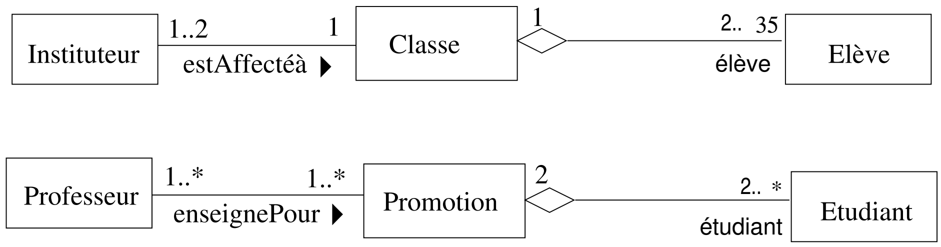
4.7 Généralisation de descriptions circulaires

4.7.1 Origine du problème

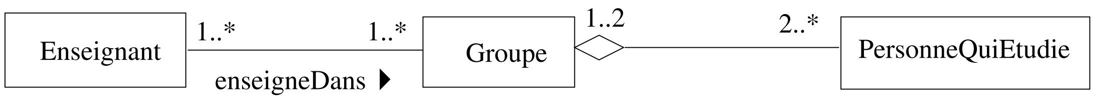
Le projet MACAO, centré sur l'évaluation mais aussi l'amélioration de diagrammes UML, nous a apporté une nouvelle problématique de généralisation. Le méta-modèle UML, par rapport à un méta-modèle objet « pur », ajoute la notion d'*association*. En UML, une association est l'abstraction d'un groupe de relations (ou *liens*) entre instances, groupe de liens qui ont une structure et une sémantique commune. Ces associations sont, tout autant que les classes, généralisables et il apparaît évident que de la généralisation des classes dépend celle des associations car une association est décrite pour partie par ses deux classes extrémités ; inversement de la généralisation des associations dépend celle des classes car les classes sont à présent décrites non plus seulement par des référents d'attributs et de méthodes mais également par leur implication dans des associations.

Nous illustrons à présent ce problème. Si l'on examine la figure 4.41, on trouve dans sa partie supérieure deux situations assez semblables de notre point de vue. Sans entrer dans tous les détails, la première décrit qu'un instituteur est affecté à une classe laquelle contient entre 2 et 35 élèves. La deuxième explicite le fait qu'un professeur enseigne dans plusieurs promotions, chacune constituée d'au moins deux étudiants. Un diagramme UML complet contiendrait plus d'informations sur certains des concepts (classe ou association). Imaginons que des attributs communs (nom, prénom, niveau d'étude, etc.) permettent de repérer que les classes `Elève` et `Etudiant` soient généralisables en `PersonneQuiEtudie`. Dans ce cas, on peut inférer que les deux agrégations qui y aboutissent présentent un caractère commun, qui est d'avoir comme extrémité des personnes qui étudient. On peut les généraliser en une nouvelle agrégation sans origine précise mais qui aboutit sur `PersonneQuiEtudie`. Les liens de cette nouvelle agrégation ont comme origine des instances de `Classe` ou de `Promotion`, ce qui est de fait un caractère commun de ces deux dernières classes, d'où une nouvelle généralisation, généralisation de classe cette fois, que nous pouvons nommer `Groupe` (de personnes qui étudient). Par étapes successives où l'on répercute une généralisation sur les concepts concernés, on peut imaginer aboutir au schéma du bas de la figure. D'autres scénarii sont possibles : par exemple ce pourrait être des caractères communs aux associations `estAffectéà` et `enseignePour` qui auraient pu déclencher ces vagues de généralisation. Ainsi ces deux associations pourraient être précisées respectivement par deux classes d'association possédant des caractères communs tels qu'une liste de matières enseignées, une période d'affectation, etc. Cet exemple nous montre comment les relations qui existent entre les différents concepts (classes et associations), relations qui établissent des circularités dans la description car `Elève` est décrit par l'agrégation tout autant que l'agrégation est décrite par `Elève`, permettent d'enrichir les possibilités de généralisation.

Mais à y regarder de plus près, ce problème touche en vérité les attributs et les méthodes, car en UML comme dans les langages de programmation ils peuvent remplacer ou accompagner une association, et de manière générale référencer des classes, au minimum par le biais du type (pour les attributs), du type de retour ou de la signature (pour les méthodes). Nous illustrons ceci dans le cas des attributs (Figure 4.42), en reprenant l'exemple ci-dessus dans un méta-modèle objet sans associations. Dans cet exemple, les attributs `eleves` et `etudiants` des classes `Classe` et `Promotion` peuvent dans un premier temps être considérés comme sans lien apparent, soit dans deux composantes connexes différentes d'une même propriété générique, soit dans deux propriétés génériques différentes.



– Deux situations généralisables –



– une généralisation possible –

FIG. 4.41 – Interdépendance des généralisations de classes et d'associations

Une généralisation `PersonneQuiEtudie` des classes `Elève` et `Etudiant` amenée comme ci-dessus par des caractères communs peut faire reconsidérer ce premier point de vue et nous inciter à ajouter aux attributs `eleves` et `etudiants` une généralisation commune, un attribut `disciples` de type `PersonneQuiEtudie`. De fil en aiguille, on pourra apporter de nombreuses possibilités de généralisation dans ce premier schéma et aboutir au diagramme présenté dans la figure 4.43.

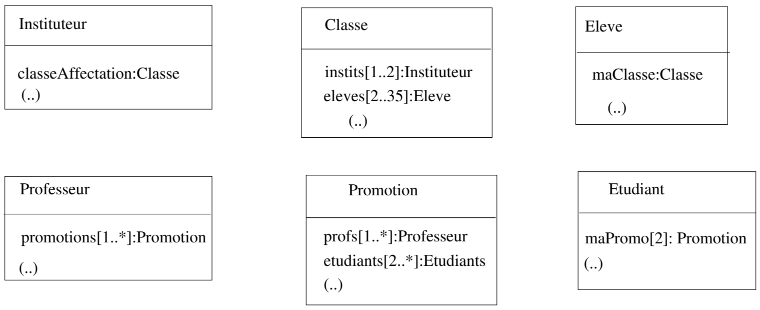


FIG. 4.42 – Interdépendance des généralisations de classes et d'attributs

Dans ce travail sur les attributs, nous ne les avons pas traités comme des entités préexistantes et dotées d'une organisation préalable, mais au contraire comme si elles étaient sujettes à de nouvelles organisations et généralisations, en bref, comme des concepts qu'il est utile aussi de généraliser. A supposer qu'avant généralisation toutes les propriétés génériques étaient réduites à un référent, la figure 4.44 montre l'effet que la généralisation a eu sur elles.

4.7.2 Illustration du procédé pour UML

La mise en place d'un mécanisme de généralisation adapté à ces descriptions circulaires nous a conduit à apporter une contribution originale au domaine de l'analyse formelle de concepts. Nous instancions ici ce mécanisme sous une forme appliquée à UML, et le lecteur

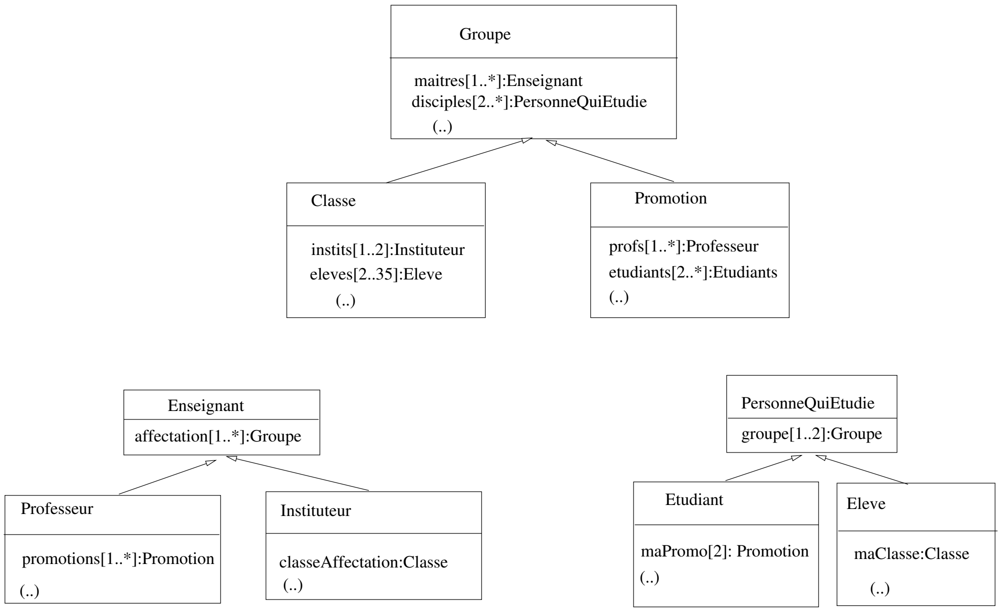


FIG. 4.43 – Résultat possible de la généralisation

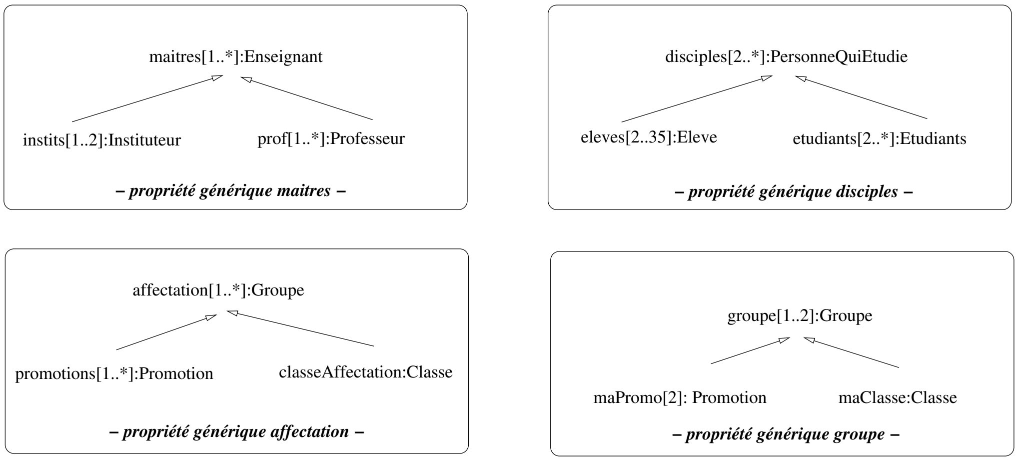


FIG. 4.44 – Propriétés génériques après généralisation

en trouvera une généralisation à toute description circulaire dans [HRV02]. Nous essayons de donner le sens de ce que nous faisons, sens qui peut paraître délicat à percevoir dans la formalisation de la référence [HRV02], même si elle présente également un exemple en UML réduit aux associations. Nous en profitons pour approfondir quelques aspects de la mise en pratique dans le cas d'UML.

Principe général des quatre contextes

Le principe général consiste, dans le cas d'UML, à considérer quatre types de « contextes formels » (pour reprendre la terminologie de [GW99]) : ces quatre contextes décrivent respectivement les classes, les attributs, les méthodes et les associations.

Dans chacun de ces contextes, les objets formels (resp. les classes, attributs, méthodes et associations) sont décrits par des attributs formels (ou référents pour éviter toute ambiguïté ici) qui seront de différentes natures.

– Les réferents sont partitionnés selon qu'ils mettent ou non en relation deux contextes en :
  – référents non relationnels (par exemple la multiplicité ou le nom d'un référent d'attribut),
  – référents relationnels (par exemple le type d'un référent d'attribut car il fait référence aux objets formels du contexte des classes).
– Une autre partition (indépendante de la précédente) distingue :
  – les référents moteurs de la généralisation, c'est-à-dire qui sont une forte indication pour généraliser (par exemple le type ou le nom des référents d'attributs),
  – les référents secondaires qui ne servent pas à créer de nouvelles généralisations, mais sont utiles *a posteriori* pour raffiner la description de généralisations existantes (toujours dans le cas des référents d'attributs, la multiplicité est rarement une indication pour généraliser, mais elle est intéressante à généraliser si d'autres éléments ont indiqué une généralisation utile).

L'exemple de la figure 4.45 va nous servir de fil conducteur. Dans cet exemple, nous retrouvons les instituteurs et leurs classes, associés par `estAffectéA`. Cette association est décrite par la classe d'association `Affectation`. Les professeurs sont par ailleurs associés par `enseignePour`, précisée par la classe d'association `Intervention`. Pour simplifier l'exemple, nous oublierons ici les élèves et les étudiants. Dans un premier temps nous explicitons les différents contextes, en nous appuyant sur les propositions de la notation UML et nous étudierons seulement ensuite comment ils peuvent s'enrichir mutuellement.

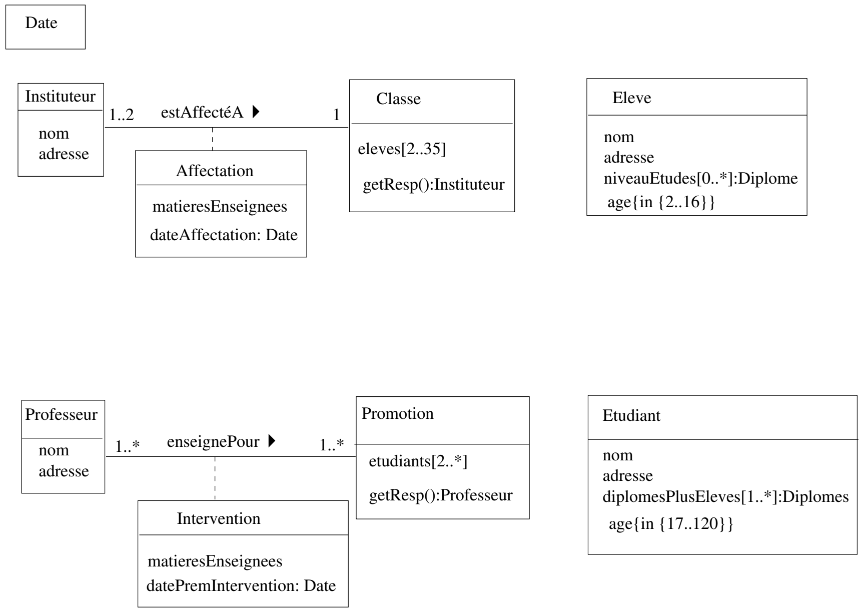


FIG. 4.45 – Enseignement : un diagramme de classes à généraliser

Contexte des attributs Selon UML (voir Def. 4.7), un attribut est décrit par différents éléments syntaxiques qui n'ont malheureusement pas tous une sémantique exploitable. Par exemple, la visibilité est supposée préciser le niveau de contrôle d'accès appliqué à l'attribut. Mais les trois niveaux (`public`, `private`, `protected`), suivant les versions d'UML, soit sont supposés avoir la sémantique du langage cible de la conception, soit ne sont pas documentés. C'est donc une information difficilement exploitable dans ces conditions [Rat99]. La multiplicité est un ensemble d'entiers qui expriment le nombre de valeurs possibles de l'attribut. Nous tiendrons pour correct qu'une multiplicité en spécialise une autre si elle est incluse et nous généraliserons naturellement deux multiplicités par union. La liste de propriétés permet par exemple de décrire si l'attribut est constant. Seul le nom est obligatoire. Il peut y avoir spécialisation entre propriétés, par exemple *addOnly* spécialise *Changeable*.

Définition 4.7 (Syntaxe d'un attribut en UML [BRJ99])
$[visibility]name[multiplicity][: type][= initial - value][\{property - string\}]$

Nous montrons un contexte formel pour les référents d'attributs dans le haut de la figure 4.46. Les référents (colonnes) utilisés sont la multiplicité, le type et le nom. La sous-hiérarchie de Galois associée à ce contexte apparaît sous une forme simplifiée au bas de la figure. Un seul concept de factorisation apparaît, montrant l'intérêt de généraliser les référents représentant des dates. Les autres concepts définissent les objets du contexte, c'est-à-dire les attributs UML.

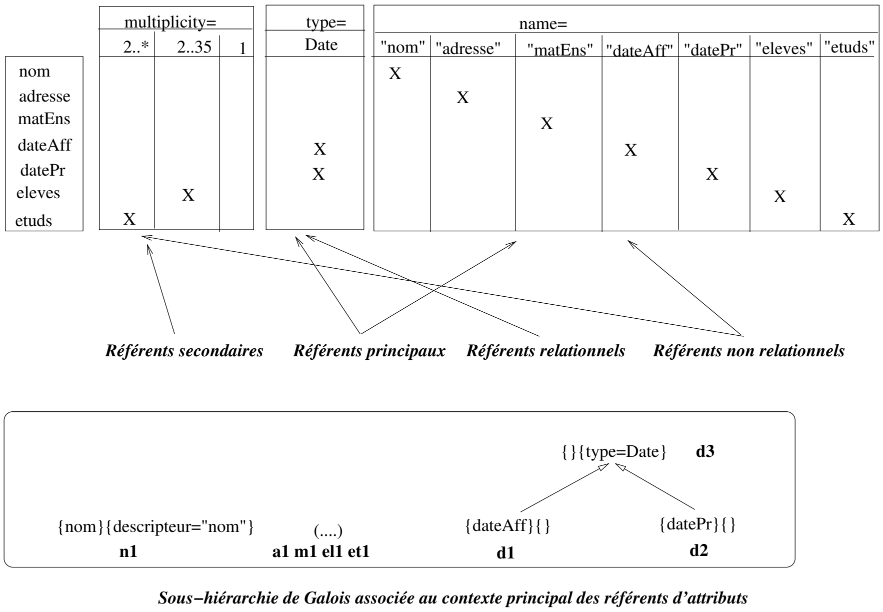


Fig. 4.46 – Un contexte décrivant les attributs

Contexte des classes Le contexte formel des classes et la sous-hiérarchie de Galois associée apparaissent dans la Figure 4.47. Une classe est décrite par des référents d'attributs et de méthodes, auxquels s'ajoutent des référents situant la classe par rapport aux associations. La relation (`Instituteur`,`origOf=eA`) (resp. (`Classe`,`destOf=eA`)) décrit que les objets de la classe Instituteur (resp. `Classe`) peuvent être origine (resp. extrémité) de `eA`. Remarquons que nous utilisons implicitement une direction dans la lecture de l'association. La relation (`Affectation`,`clOf=eA`) décrit le fait que la classe `Affectation` est classe d'association de `eA`. Si l'on interprète la sous-hiérarchie de Galois comme une hiérarchie de classes, deux nouvelles classes sont apparues, correspondant respectivement aux concepts `pers` et `ai1`. La première généralise les concepts qui définissent respectivement `Instituteur` et `Professeur`. Par abus de langage disons que la deuxième généralise les deux classes d'association `Affectation` et `Intervention`.

Contexte des Méthodes Nous présentons tout d'abord une interprétation de ce qu'UML entend par méthode (spécification d'une opération accompagnée d'un corps).

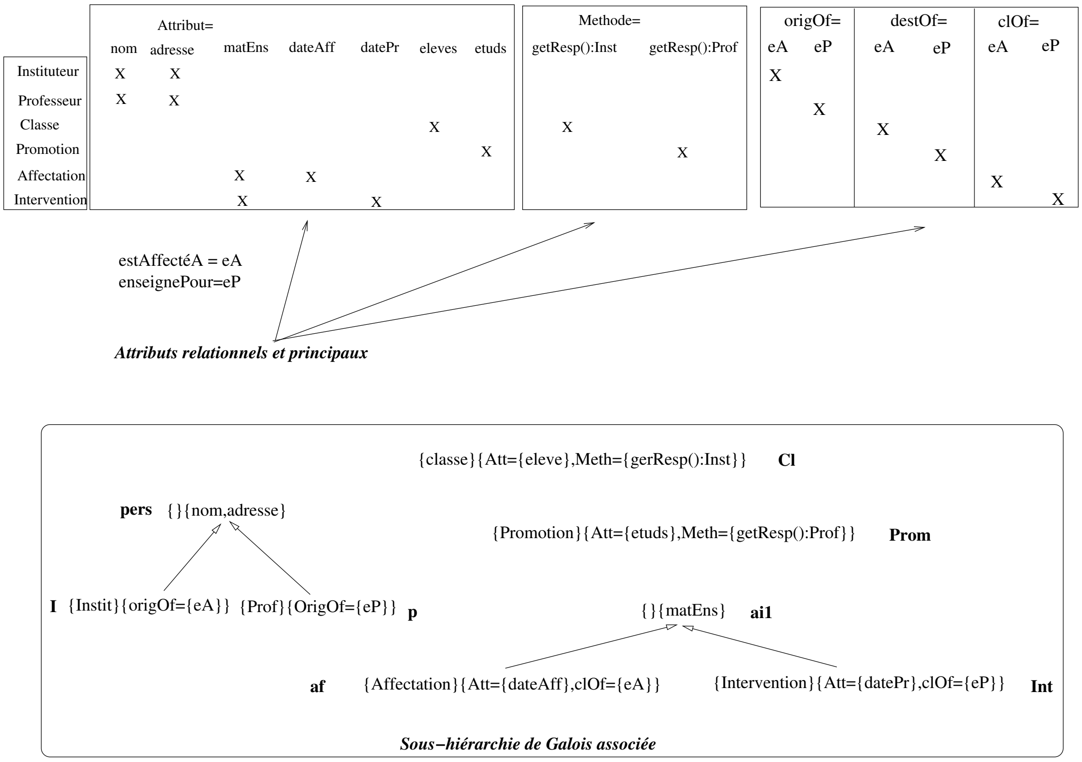


FIG. 4.47 – Un contexte décrivant les classes

DÉFINITION 4.8 (SYNTAXE D'UNE MÉTHODE D'APRÈS UML [BRJ99])
$[visibility]name[(param{-}list)][: return{-}type][= initial{-}value][\{property{-}string\}][body]$

La figure 4.48 présente le contexte décrivant les méthodes. La sous-hiérarchie de Galois associée fait apparaître une généralisation possible des deux méthodes existantes car elles portent le même nom.

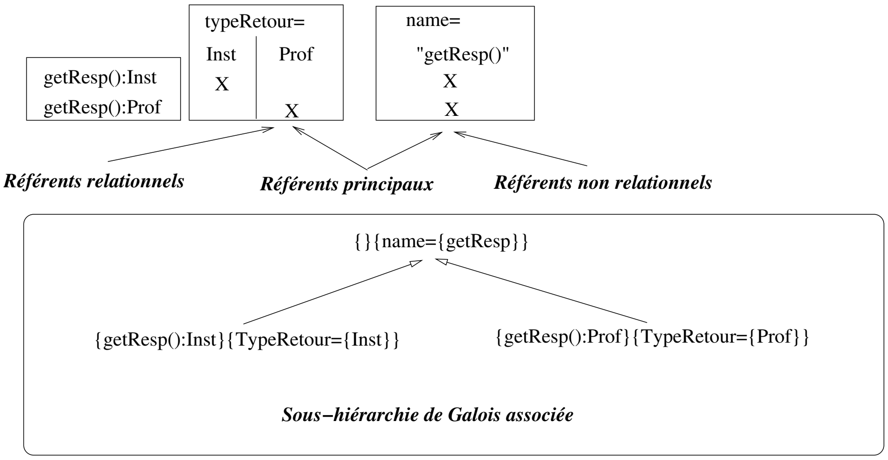


FIG. 4.48 – Un contexte décrivant les méthodes

Contexte des Associations Enfin la figure 4.49 propose un contexte décrivant les associations présentes. Nous y faisons figurer quelques-unes des nombreuses informations qu'UML propose d'utiliser pour la description des associations et que nous avons évoquées plus en détails dans [DVH+02] : les multiplicités à l'origine (`mo`) et à la destination (`md`), les types dont les objets peuvent être origine (`to`) et destination (`td`), enfin la classe d'association (`ca`). L'inclusion des multiplicités et la spécialisation des classes dictent la spécialisation des référents.

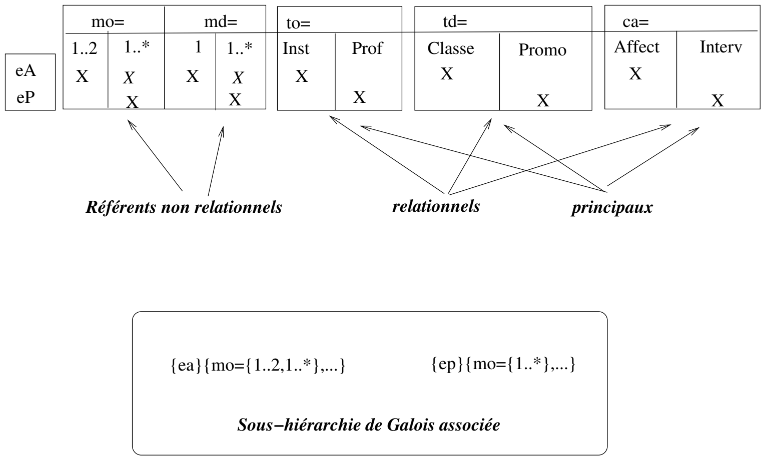


FIG. 4.49 – Un contexte décrivant les associations

Un schéma général (Figure 4.50) résume les relations entre les contextes et les référents qui établissent ces relations.

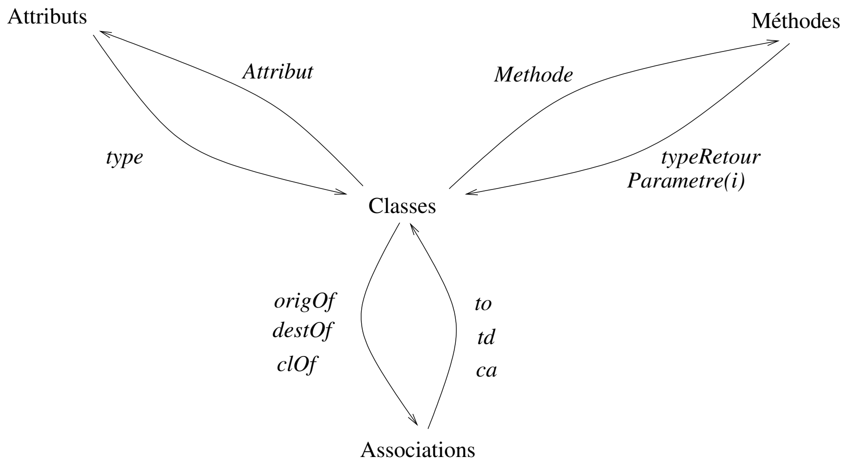


FIG. 4.50 – Relations entre les contextes

Enrichissement mutuel des contextes

Nous exposons à présent la manière dont un contexte peut en enrichir un autre. Le premier point à rappeler est que la sous-hiérarchie de Galois associée à chacun des contextes est réinterprétable respectivement comme une hiérarchie de classes, d'associations, ou des propriétés génériques regroupant des attributs ou des méthodes. Par exemple, la figure 4.51 présente une hiérarchie de classes correspondant à l'interprétation de la sous-hiérarchie de Galois de la figure 4.47. Le concept `pers` est devenu la classe `Personne`, le concept `ai1` a donné naissance à `EnseignementCourant`, les associations ont été replacées (avec une présentation simplifiée) pour mémoire.

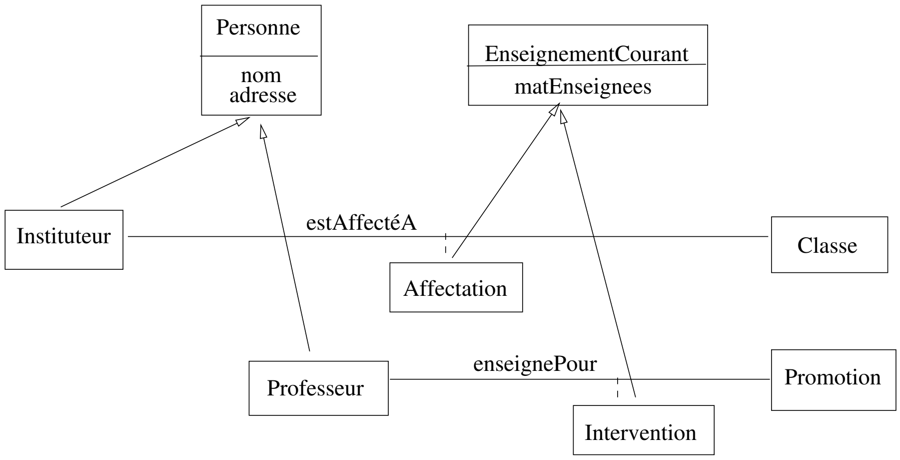


FIG. 4.51 – Une interprétation de la sous-hiérarchie de Galois de la figure 4.47

Cette interprétation peut être utilisée pour enrichir les connaissances que nous avons sur les associations de la manière suivante.

- Puisque (Figure 4.49) les objets origine de `eA` sont des `Instituteur` (classe définie par le concept `I`), ce sont aussi des `Personne` (classe représentant le concept `pers`). Le même raisonnement peut être appliqué à `eP`. D'où une colonne supplémentaire `to=pers` dans la relation de la figure 4.52. Elle ne doit surtout pas être interprétée comme le fait que `Personne` est devenue origine de `eA` ou `eP`, mais plutôt que `Personne` est potentiellement origine d'une généralisation de `eA` et `eP`.
- Les classes `Affectation` et `Intervention`, décrivant respectivement les associations `estAffectéA` et `enseignePour`, sont généralisées par `EnseignementCourant` (correspondant au concept `ai1`). On attribue pour cela une nouvelle caractéristique commune à `eA` et `eP`, en l'occurrence `ca=ai1` (toujours figure 4.52).

	mo=		md=		to=			td=		ca=		
	1..2	1..*	1	1..*	I	p	pers	Cl	Prom	af	Int	ai1
eA	X	X	X	X	X		X	X		X		X
eP		X		X		X	X		X		X	X

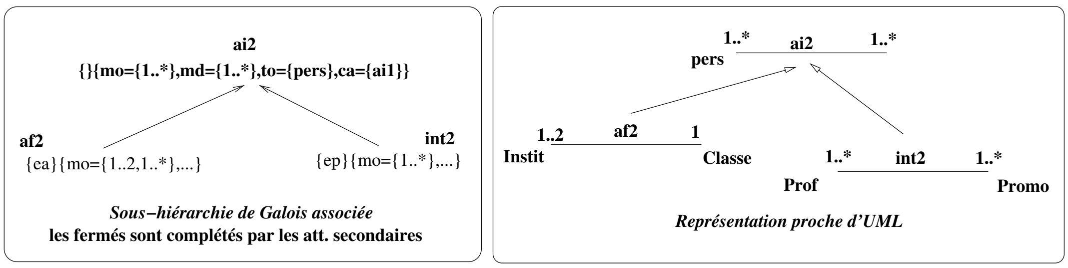


FIG. 4.52 – Nouvelle généralisation des associations

Ces nouvelles caractéristiques font apparaître (Figure 4.52) des aspects communs aux deux associations, et permettre ainsi leur généralisation. Au bas de la figure, on peut constater qu'un nouveau concept `ai2` est apparu dans la sous-hiérarchie de Galois. Cette association n'est pas tout à fait précisée. Ce qui est déterminé est qu'elle a pour origine des personnes, et ses multiplicités sont calculées par union des multiplicités de `af2` et `int2`.

Parallèlement, les connaissances sur les méthodes sont également augmentées. Les deux méthodes `getResp()` retournent respectivement des instituteurs ou des professeurs, ce que nous pouvons à présent généraliser en disant qu'elles retournent donc des personnes (une nouvelle colonne `typeRetour=pers` est ajoutée). Le concept le plus haut de la sous-hiérarchie de Galois peut être complété comme cela est illustré dans la figure 4.53.

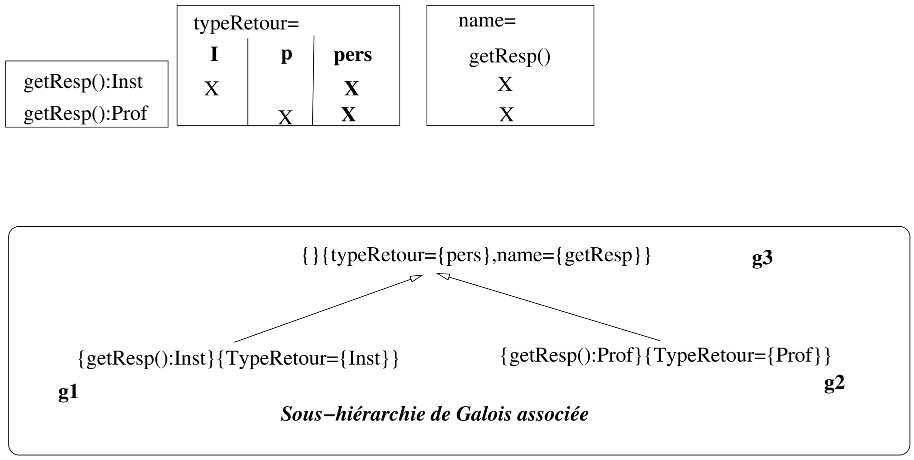


FIG. 4.53 – Nouvelle généralisation des méthodes

Le report de toutes ces nouvelles connaissances sur les classes est développé figure 4.54. Le point principal à remarquer est la découverte de `Groupe`, nouvelle généralisation des concepts de classe et de promotion, que nous n'aurions pas obtenue sans les étapes précédentes.

Sans détailler toutes les étapes, un tel procédé nous permet d'obtenir le diagramme de classes de la figure 4.55. Un concepteur peut alors décider quelles sont les généralisations intéressantes, factoriser ou non l'attribut `d:Date`, garder une signature covariante pour la méthode `getResp` ou n'en garder qu'un exemplaire, etc. Ce qui est important, c'est qu'il dispose de toutes les généralisations possibles pour effectuer ses choix et améliorer son diagramme de classes.

4.7.3 Conclusion et prospectives

Ce travail présente de nombreuses possibilités de développement autant d'un point de vue pratique que d'un point de vue théorique.

Scenarii d'utilisation Pour rendre une telle méthode utilisable dans un atelier d'analyse et de conception, il est important d'étudier de bons scenarii d'utilisation. La méthode que nous présentons, qui a la particularité d'être incrémentale, avec une émergence pas à pas des généralisations permet au concepteur de comprendre le processus et d'interagir avec lui. Divers types d'interaction sont envisageables pour valider ou infirmer les généralisations effectuées, stopper le processus avant sa convergence, simplifier les diagrammes, etc. La procédure de généralisation peut employer n'importe quel circuit du schéma de la figure 4.50 mais certains circuits sont peut-être plus significatifs pour le concepteur, la spécialisation des classes étant plus au cœur de son travail.

Syntaxe vs. sémantique Pénétrer la sémantique de la modélisation au travers de l'aspect plutôt syntaxique des diagrammes UML est un véritable challenge comme nous

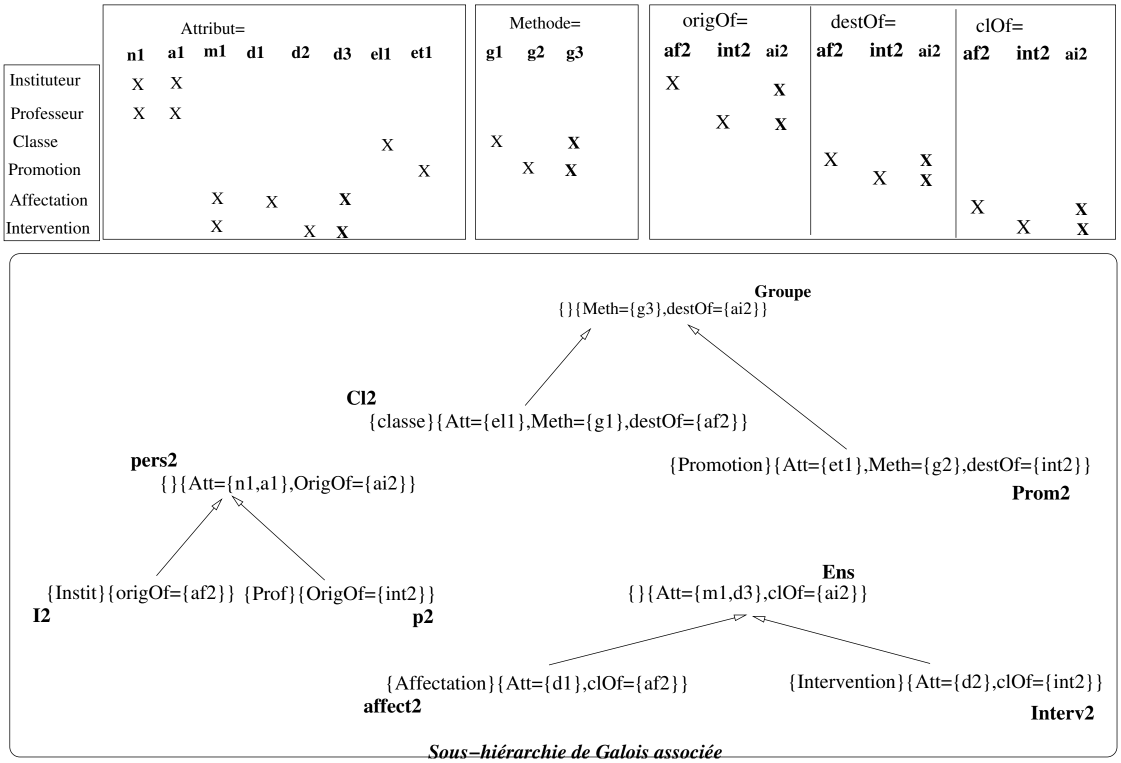


FIG. 4.54 – Nouvelle généralisation des classes

l'avons mentionné tout au long de ce document. Parmi les problèmes les plus notables qui concernent plus spécifiquement les associations, notons :

- *noms et rôles* : les noms des associations sont tout autant que les autres une source de problèmes de conflits ou de synonymie, qui ne peuvent être traités que par un module spécifique de linguistique ou par le concepteur lui-même ; de plus, le nom de l'association n'est pas toujours mentionné sur un diagramme et l'on doit se contenter de noms de rôles ou parfois même du seul symbole d'agrégation ou de composition ;
- *ordre sur les extrémités* : Le nom de l'association et le triangle noir qui y est attaché ordonnent explicitement les classes extrémités [Rat99]. Nous avons préservé cette dissymétrie en identifiant une origine et une destination dans notre manière de décrire les associations. Il n'est pas certain que cet ordre soit toujours approprié, lorsqu'on ne dispose que des noms de rôles ou lorsque l'association est très symétrique ;
- *statut des classes d'association* : Comme cela est dit dans [Rat99], « la classe d'association et l'association constituent la même entité sémantique ». Ceci suggèrerait que les variables et les méthodes de l'association pourrait être utilisées comme attributs formels de l'association et que la classe d'association pourrait ne pas apparaître dans le contexte des classes. Je ne suis personnellement pas favorable à cette interprétation, mais le débat est toujours ouvert dans notre groupe de travail ;
- *Niveau d'importance des attributs* : Comme le mentionne [RC00] en se référant à la tradition aristotélicienne, certaines descriptions sont « essentielles » (disent ce qu'est le sujet) tandis que d'autres sont plus « contingentes » (énoncent des faits annexes pour le sujet). Nous avons essayé de commencer à mettre en place cette distinction par le biais des référents secondaires *versus* principaux, et ce travail pourrait être approfondi.

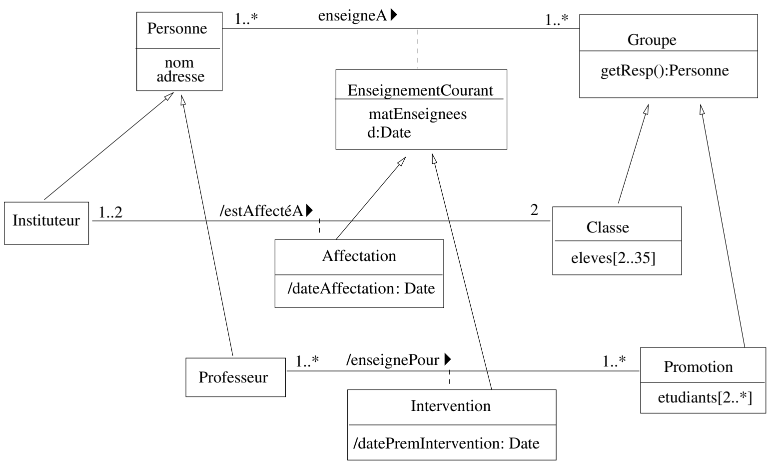


FIG. 4.55 – Etape finale

Aspects théoriques D'un point de vue théorique, la convergence de la méthode est assurée par le simple fait que les relations ne sont étendues que sur une dimension (seuls de nouveaux référents sont ajoutés mais jamais de nouveaux objets formels). Restent à énoncer et à établir avec plus de précision un certain nombre de propriétés telles que la complétude des généralisations (on peut attendre une telle propriété, la méthode procèdant par saturation) ou la complexité théorique du calcul dans les deux cas d'application possibles (construction d'un treillis ou construction d'une sous-hiérarchie de Galois).

Portée de la méthode Ce travail se pose en généralisation des travaux rapportés dans [DDHL96], en prenant les référents adéquats pour les langages de programmation considérés. Il contient les éléments qui permettent d'en étendre les résultats partiels et *ad hoc* et de se pencher sur le traitement de situations d'auto-référence quelconques. Cependant il a évidemment une portée beaucoup plus générale que la programmation et la modélisation par objets et pourrait s'appliquer par exemple à différents problèmes d'extraction de connaissances à partir de données relationnelles ainsi qu'aux graphes conceptuels [HRV02].

Nous pensons également que cette approche doit être mise en relation avec d'autres techniques de généralisation de graphes telles que celle qui est présentée dans [Liq02].

Chapitre 5

Une vue sur le domaine de la construction automatique de hiérarchies de classes

Dans cette partie nous proposons une vue générale sur le domaine de la construction automatique de hiérarchies de classes. Cette vue provient des travaux déjà effectués dans [HLD99] et plus récemment dans [GHRV02] et reprend les termes du chapitre 2 selon lesquels se définit une méthode de classification automatique. Quelques références nouvelles sont également ajoutées. Certains des schémas sont adaptés à partir de discussions avec R. Godin, P. Valtchev et C. Roume. Il est notable que de nombreux travaux précurseurs se trouvent dans le domaine des bases de données à objets [Cas98] et que divers courants (ne se divisant pas forcément entre le domaine des bases de données et celui de la programmation par objets) semblent s'ignorer parfaitement, ce qui montre l'importance de maintenir à jour une telle étude du domaine.

Nous dresserons tout d'abord le cadre (section 5.1) de cet état de l'art. Puis nous aborderons les critères de qualité dont se réclament les méthodes étudiées (section 5.2), le langage de description retenu (section 5.3), les structures de classification visées (section 5.4), les procédures et scénarii proposés (section 5.5) et enfin les applications pratiques connues (section 5.6). Nous concluons dans la section 5.7.

5.1 Cadre

Dans un premier temps, il est utile de préciser les travaux que nous considérons comme relevant pleinement de la construction automatique et ceux qui s'en éloignent. Pour cela, nous utilisons une cible (figure 5.1) dont le cœur est constitué des apports que nous jugeons les plus notables à l'automatisation proprement dite. Plus on s'écarte de ce centre, plus les apports sont annexes. Dans cette première section, seules quelques références du domaine sont données à titre d'exemple de cette classification.

Dans le cœur se trouvent les travaux qui effectuent des transformations importantes sur les hiérarchies.

- *Construction intégrale* à partir d'instances prototypiques [LBSL90, LBSL91], d'une description exhaustive des classes par leurs ensembles de propriétés ou une spécification de haut niveau [GM93, Dvo94], de l'expression du problème sous forme de dépendances fonctionnelles [ASZ92], ou plus audacieusement à partir de textes libres en langue naturelle [LA02].

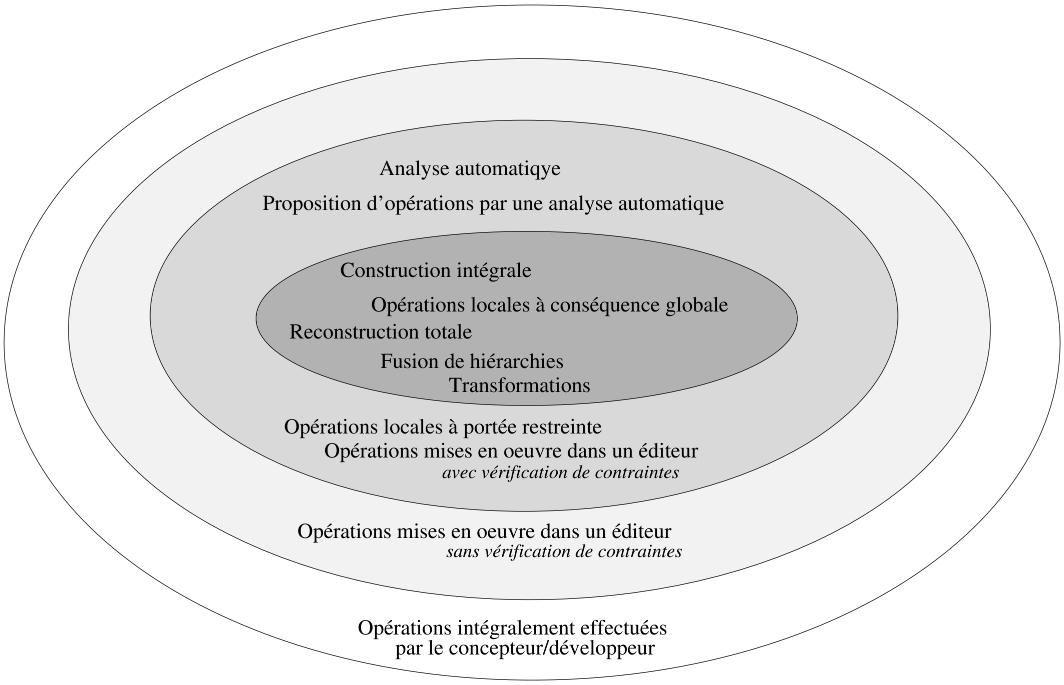


FIG. 5.1 – Cadre de l'état de l'art

- *Reconstruction totale* en utilisant certains aspects et une interprétation de la connaissance inscrite dans la hiérarchie. Nous en avons donné un exemple dans la section 4.2. Le plus courant consiste en effet à « remettre à plat » la hiérarchie en explicitant dans une relation binaire quelles propriétés possède une classe, puis à appliquer un algorithme global de reconstruction [Cas91, Coo92, GM93]. Une autre approche consiste à établir quelles sont les propriétés d'une classe qui sont réellement utilisées pour reconstruire une hiérarchie adaptée à un ensemble d'usages [ST98, ST00].
- *Fusion de hiérarchies* provenant de différents développements. A ma connaissance, ceci n'a été étudié que dans le cas très particulier du tissage d'aspects [ST02] et mériterait certainement plus d'attention. Evidemment la fusion pose plus nettement que toutes les autres opérations la question du nommage et des choix hétérogènes de critères de classification. C'est sans doute une question parmi les plus délicates. Elle peut se décliner sous des formes affaiblies telles que la greffe d'une hiérarchie dans une autre (l'une des deux hiérarchies ayant ainsi la primauté sur certains choix d'organisation).
- *Opérations locales de conséquence globale* telles que l'ajout, la suppression de classes, ou encore la modification de propriétés, de liens d'héritage. De telles opérations, si elles sont menées de manière à préserver certaines caractéristiques comme la factorisation maximale ou le respect de la spécialisation entre propriétés entraînent généralement des modifications très importantes de la hiérarchie. Ceci a été particulièrement mis en évidence pour l'ajout de classe, opération qui a été très étudiée [Cas92, GMM95, DDHL95, DDHL96], mais est également vrai des autres opérations [DVH$^+$02].
- *Transformations de structures* en particulier le passage d'une classification multiple (graphe orienté sans circuit) à une classification simple (arborescence). Cette opération a été encore peu étudiée. Elle est effleurée dans [GM93, ST00] où quelques indications sont données et un peu plus approfondie dans [CMR02, Rou02b].

Nous réunissons dans un deuxième cercle des travaux qui admettent une automatisation plus partielle.

– Les *opérations locales* à portée restreinte, mises en œuvre ou non par un éditeur (il peut s'agir seulement de définitions comme dans [Ber91]), mais associées à des vérifications de contraintes. On trouve de telles opérations dans le domaine du *refactoring* [OJ93, FBB+99] et de l'évolution des bases de données [Cas95, BKKK87].
– Les *opérations proposées par une analyse automatique* comme le calcul de métriques [Arf96, SGM00] ou la découverte de patrons d'utilisation [LM94]. De ce fait les travaux portant sur une automatisation de l'analyse (métriques [DHL+02] ou analyse formelle de concepts [ST00, AM02a, AM02b]) prennent tout leur intérêt.

Les cercles plus extérieurs correspondent à des opérations d'édition dans un environnement standard de développement qui laissent une très faible part à la vérification et enfin le traitement sans aucune automatisation.

La suite de cet état de l'art se concentre essentiellement sur le cœur que nous venons d'identifier.

5.2 Critères de qualité utilisés

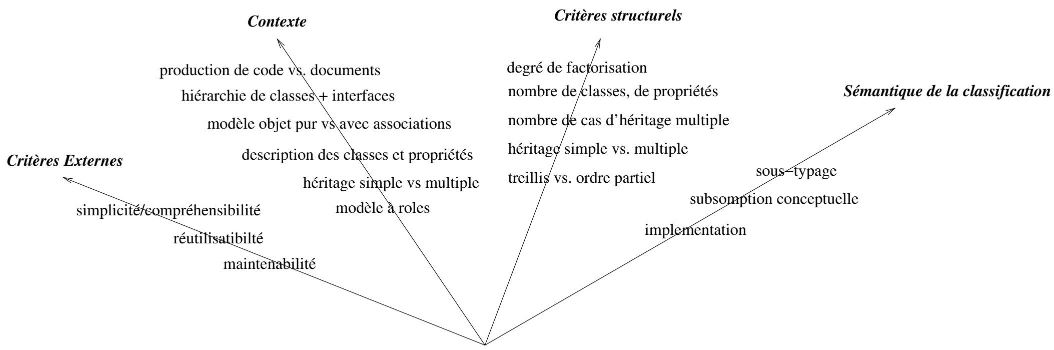


Fig. 5.2 – Critères de qualité

Comme le présente la figure 5.2, différents axes peuvent être considérés lorsque l'on cherche à relater les critères de qualité mis en avant par les diverses propositions.

Contexte Le contexte de manipulation des hiérarchies joue un rôle fondamental dans les propriétés qui vont être recherchées. Lorsque [Coo92] cherche à extraire des interfaces dans les hiérarchies Smalltalk, ce n'est pas pour produire du code Smalltalk exécutable, mais pour extraire une spécification des classes, base de documentation par exemple. C'est tout à fait différent de la perspective dans laquelle nous nous sommes placés (chapitre 4) pour construire des interfaces Java. La construction de classes dans un système composé de classes et d'interfaces peut prendre un aspect différent de la construction de classes dans un système qui ne sera constitué que de classes. Dans le premier cas, on peut plus facilement admettre, comme le fait [Ast96], que la classification des classes ne soit régie que par une sémantique d'implémentation. Ce que l'on peut exprimer dans la description des propriétés (attributs, méthodes, contraintes, associations, etc.) et dans la description des classes et des liens influe nécessairement sur l'expression des critères. Enfin on ne déploiera pas les mêmes arguments lorsque l'on est contraint à de la classification arborescente, lorsque l'on dispose de la grande liberté de la classification multiple ou enfin lorsque l'on évolue dans un modèle admettant des rôles [CL02b].

Critères externes L'objectif de l'automatisation est naturellement d'améliorer certaines des qualités externes d'une conception ou d'un développement [Mey97] : simplicité, compréhensibilité, facilité d'utilisation, niveau de réusabilité, extensibilité, maintenabilité, etc. Mais de telles qualités externes (parfois contradictoires) s'obtiennent en améliorant ce que nous appellerons des qualités techniques (par exemple la factorisation de code peut augmenter la réutilisation de code mais diminuer la compréhensibilité, la factorisation de spécifications peut augmenter la maintenabilité et le niveau de réutilisation, la cohésion peut augmenter la compréhensibilité, etc.). Nous divisons ici ces qualités techniques en critères sémantiques et critères structurels.

Critères sémantiques Nous avons déjà exposé précédemment que l'organisation des classes est sujette à trois interprétations majeures que nous allons donc retrouver dans les diverses propositions d'automatisation. Ces interprétations ne sont pas toujours clairement affichées dans les références étudiées ou sont parfois confuses dans les termes utilisés. La figure 5.3 est un essai de classification des approches suivant le critère sémantique.
- L'organisation permet de faire des économies de développement comme dans [Moo96, Ast97].
- L'organisation est conforme au sous-typage tel que le définit la théorie des types, et compris comme de la substituabilité [Cas97, CL96, Coo92, Cas95]. Les travaux en bases de données se placent souvent dans un contexte de types structurés [MS89, Run92]. Dans leur cas, sous-typage et spécialisation se rejoignent. Dans la partie automatisée du travail de [Coo92] c'est également le cas.
- L'organisation reflète la spécialisation du domaine d'application (ou « subsomption conceptuelle ») [Cas95, GM93, DDHL96, GMM$^+$98, HL00]. Sur le schéma (figure 5.3), nous assimilons la spécialisation à l'inclusion stricte des intensions ou à la spécialisation des intensions (spécialisation du domaine ou du type d'une caractéristique par exemple).

Il faut garder à l'esprit également qu'un même outil technique (en l'occurrence les treillis de Galois) peut servir plusieurs points de vue différents suivant la manière dont on l'alimente en informations : on peut ainsi factoriser du code ou des spécifications et le résultat en sera tout à fait différent.

E. Casais a pour sa part une approche originale pour traiter des différentes interprétations d'un lien entre des classes [Cas94]. Partant de l'idée qu'un lien peut véhiculer quatre sémantiques fondamentales (implémentation, sous-typage, restriction ou utilisation), il propose un algorithme qui décompose un lien de sémantique complexe en plusieurs liens dont la sémantique est unique.

Critères structurels Les critères structurels les plus marquants sont les suivants.
- factorisation maximale. Un niveau élevé de factorisation est supposé contribuer à la facilité d'utilisation car les éléments sont faciles à retrouver [GMM95], à la réutilisabilité car l'augmentation du niveau d'abstraction dans la hiérarchie doit conduire à des classes qui s'appliquent dans un contexte plus large [Cas95], dans certains cas à la compréhensibilité [HDL00].
- minimisation du nombre de classes ou de propriétés, en éliminant des redondances basées sur les usages connus [TS97, ST00], effacement de propriétés inutilisées [TCFR96].
- dans le cas où la classification multiple est admise
  - minimiser les situations d'héritage multiple [LBSL90, LBSL91] ou au contraire le favoriser (les approches qui produisent une factorisation maximale)

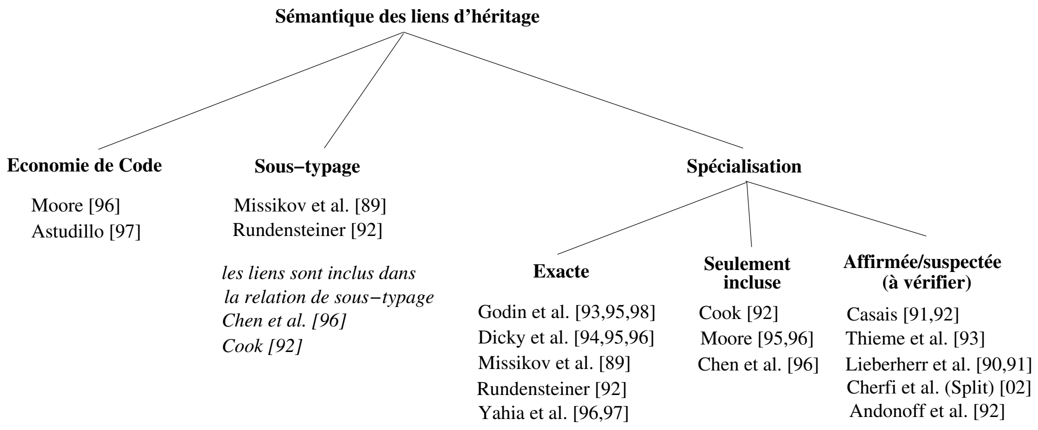


FIG. 5.3 – Sémantique de l'organisation des classes

  - promouvoir l'organisation de type treillis par rapport à celle d'un ordre partiel quelconque [MS89, Run92, YLCB96, YLB97].
- dans le cas où le rejet d'une propriété héritée est admis
  - l'utiliser de manière à limiter l'effort de codage [Ast97]
  - éviter au maximum les cas de rejets [Cas92].
- dans [CL02a], deux propositions sont faites. L'algorithme véritablement automatisé (Split) devrait produire une factorisation mais le problème n'est pas exprimé en ces termes donc ce point reste à étudier. L'opération semi-automatisée qui génère des contraintes appropriées lors d'une fusion de classes (Restructuring) ne devrait pas invalider cette factorisation, mais la classe produite a des attributs optionnels ce qui diffère du modèle classique.

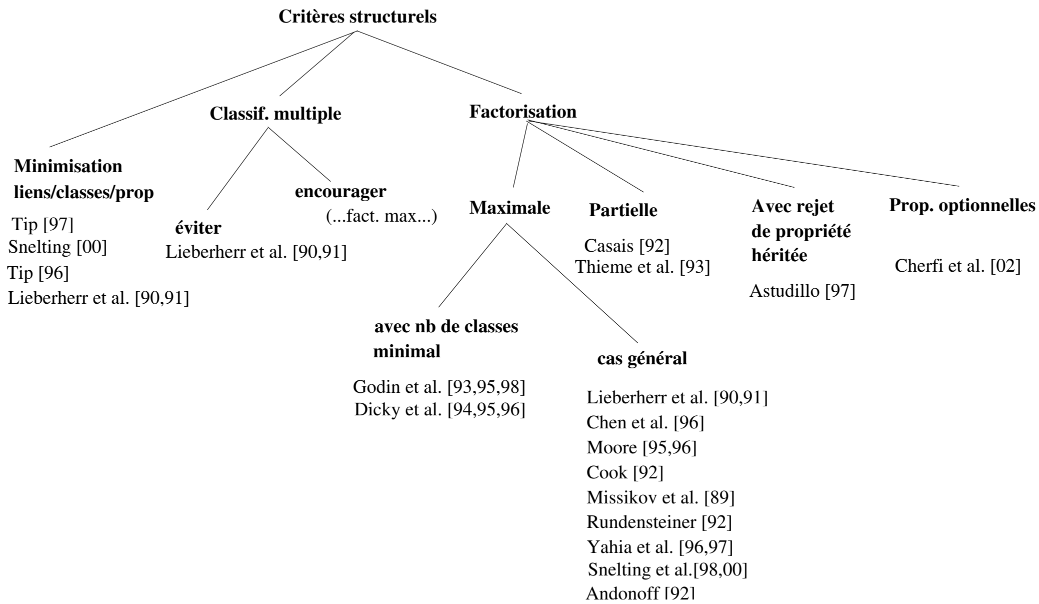


FIG. 5.4 – Critères structurels recherchés

5.3 Langage de description

La figure 5.5 classe les approches selon le formalisme de description utilisé. Le formalisme majoritaire est une description des classes par une liste de propriétés (autrement dit une relation binaire « owns » associant les classes aux propriétés qu'elles possèdent). Les propriétés sont des attributs ou des méthodes décrites avec plus ou moins de précision selon que l'on avance sur l'axe (type, signature, corps). A l'extrémité de l'axe se trouvent les approches qui considèrent une relation de spécialisation entre propriétés.

- La forme la plus simple est une liste de noms d'attributs [Ast96, Ast97, YLCB96, YLB97]
- Pour traiter un modèle d'origine qui est un schéma Entité-Relation, [LLJ99, CL02a] utilisent des attributs accompagnés de contraintes (optionnel/existence mutuelle/existence exclusive/implication). Un travail à mener serait de vérifier que ce modèle n'est pas traduisible en codant dans une relation binaire les différentes contraintes.
- Un type peut ensuite être adjoint à l'attribut [MS89, Run92]
- Au lieu de classes, on peut décrire des instances prototypiques par des attributs typés [LBSL90, LBSL91]
- Les signatures de méthodes sont parfois la seule description retenue [Coo92]
- ... ou bien on utilise à la fois attributs typés et signatures de méthodes [CL96, Moo95, MC96, Leb00, HL00]. Dans [HL00], puisqu'il s'agit de construire des interfaces en Java les seuls attributs considérés sont les attributs statiques, constants et publics.
- En plus d'attributs typés et de signatures de méthodes on peut utiliser le corps des méthodes [Cas92, Moo96].
- Enfin, on peut considérer un ordre de spécialisation sur les propriétés [GM93, GMM95, DDHL95, DDHL96]. La référence [Cas92] utilise en fait un tel ordre mais réduit à un seul niveau (une méthode concrète spécialise une méthode abstraite de même signature). Dans une étape non automatique de sa construction d'interfaces, [Coo92] s'intéresse à une opération de raffinement au sein de familles de méthodes. Ce raffinement se base sur les pré-conditions et post-conditions, ce qui est tout à fait dans l'esprit des propriétés génériques, avec un ordre issu de la relation de substituabilité.

A l'écart de ce classement se trouvent les approches de [TS93] qui considèrent des descriptions des classes par des arbres, les dépendances fonctionnelles de [ASZ92] ou la relation d'utilisation qu'exploitent [ST98, ST00]. Enfin on note nos travaux récents sur la reconstruction de classes et d'associations [HRV02, DHLR01b].

5.4 Structures de classification

Nous identifions essentiellement quatre structures de classification proposées (voir Figure 5.6) :

- *Arborescence*,
- *Graphe orienté sans circuit (quelconque)*,
- *Sous-hiérarchie de Galois*,
- *Treillis*.

Il est intéressant de constater que la plupart des propositions se concentrent sur une classification multiple. Ce résultat est naturel car il est dicté par une recherche de factorisation maximale qui ne peut dans le cas général s'obtenir qu'ainsi. La recherche d'une factorisation plus compacte amène aussi un certain nombre de travaux à calculer une structure proche d'une sous-hiérarchie de Galois comme cela est démontré dans [HDL00]. Les approches qui calculent une arborescence s'en explique en général par l'exigence technique

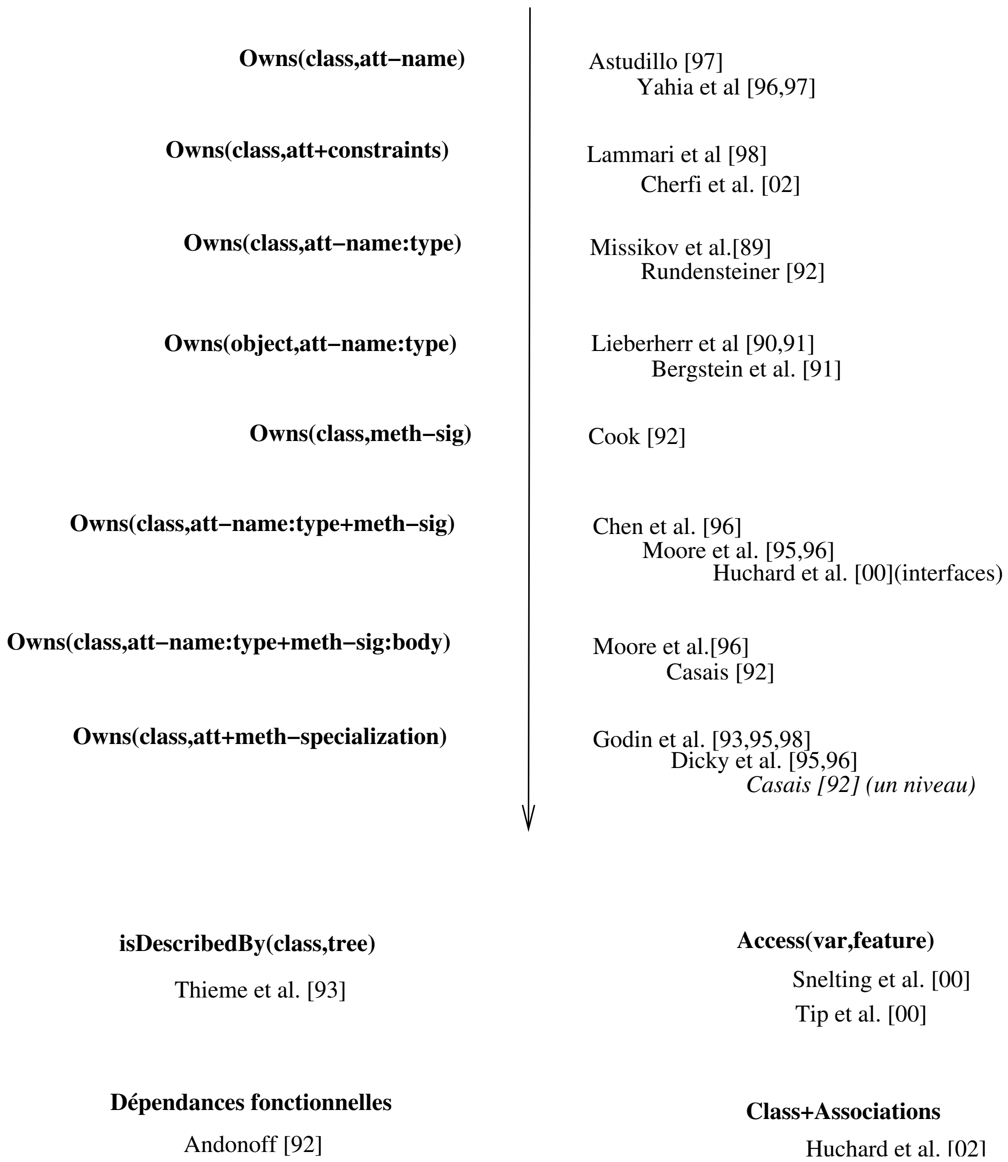


FIG. 5.5 – Formalismes de description des données

d'un langage cible.

De ce croquis nous nous permettons de conclure que la sous-hiérarchie de Galois présente un caractère intuitif puisque cette structure est souvent recherchée, même si elle n'est pas mentionnée explicitement dans les références. Elle ne présente pas toutes les généralisations possibles (ce que fait le treillis de Galois) mais elle en présente un nombre important et surtout propose une organisation viable techniquement pour une utilisation quasiment directe dans un langage de programmation car les classes d'origine sont représentées et les propriétés sont toutes déclarées. Nous conjecturons que l'algorithme principal de [LLJ99, CL02a] peut s'exprimer comme la construction d'une sous-hiérarchie de Galois, car les règles de base de l'algorithme semblent impliquer un calcul de fermeture. En ce qui concerne [ASZ92], l'emploi des techniques de construction de forme normale non redondante justifie le même type de conjecture.

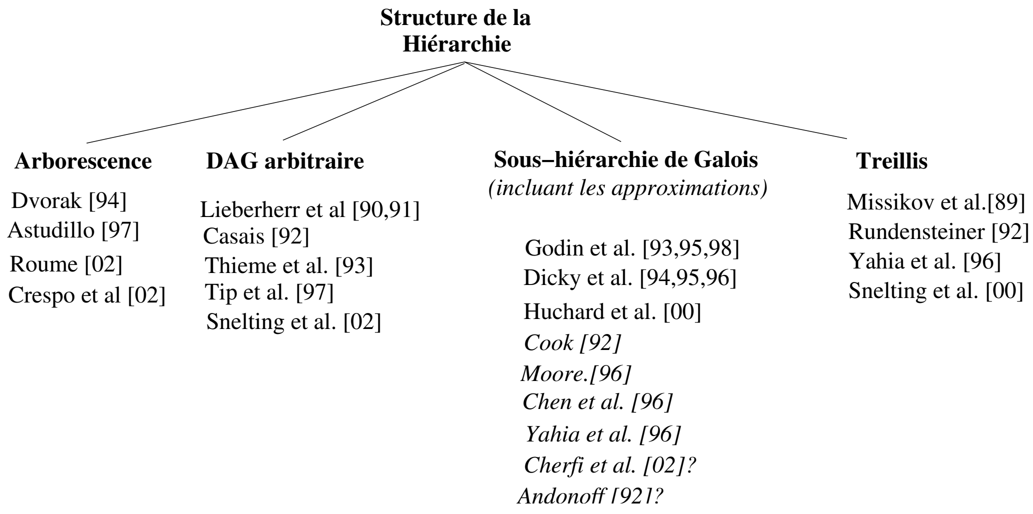


FIG. 5.6 – Structures de classifications

5.5 Procédures de classification

Le classement des procédures de classification existantes (Figure 5.7) montre une grande quantité de procédures globales. Reconstruire intégralement semble bien un point de départ logique. Dès qu'une hiérarchie est (re)construite de manière systématique il devient plus facile de la faire évoluer sainement. Peu de travaux se préoccupent encore des algorithmes d'évolution autres que les algorithmes d'ajout de classe. Pour le projet MACAO, nous étudions justement l'évolution d'une hiérarchie [DVH+02] qui soulève évidemment des problèmes plus larges que les problèmes algorithmiques proprement dits : il faut en effet mettre en perspective l'utilisation des algorithmes d'évolution et les scénarii d'utilisation par un concepteur. Ce peut être assez délicat à mettre en place si l'on veut proposer des cycles de modifications dans lesquels à la fois le résultat est théoriquement fondé (cohérence entre la spécialisation des référents et la hiérarchie) et le concepteur peut exprimer ses *desiderata* (refuser ou forcer certaines spécialisations entre référents, modifier le résultat des heuristiques de calcul de référents, etc.). Le problème de la spécification d'un scénario d'élaboration de hiérarchie est abordé dans [CL02a].

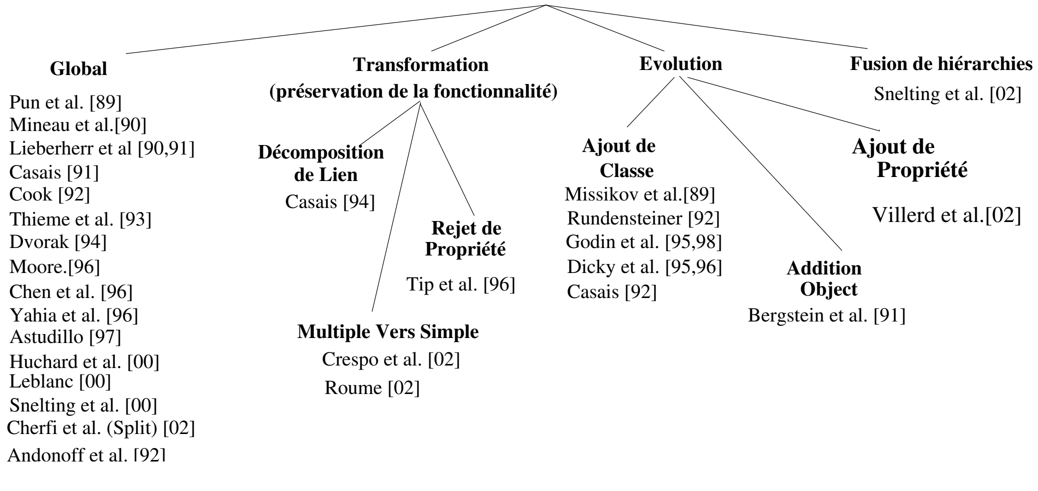


FIG. 5.7 – Procédures de classifications

5.6 Applications pratiques connues

La mise en pratique reste un point faible de la plupart des approches. Comme le montre le schéma (figure 5.8), il reste du chemin à faire ! La difficulté réside dans le fait de trouver des concepteurs qui ont du temps à consacrer à l'utilisation d'un outil et à l'analyse des résultats (parfois obscurs). Les API (*Application Programming Interface*) offrent un support intéressant, d'autant plus que l'on peut en étudier l'évolution. Mais il n'y a pas de retour sur expérience autre que celui que nous pouvons en faire, et cela nous place dans une position où nous sommes à la fois juge et partie. Dans le cadre des travaux avec France Télécom, Michel Dao nous a permis de tester la reconstruction et les métriques sur l'application Pétra et a pu en discuter les résultats. Les outils de l'équipe de R. Godin ont été testés sur des applications du domaine industriel. Une voie que nous avons commencé à explorer est le test sur les logiciels de laboratoire (tels que Resyn). Les chercheurs en charge du développement peuvent être plus sensibles à la demande d'autres chercheurs ainsi qu'aux suggestions issues des tests. Nous avons débuté ce type de travail avec les métriques [BG02] ainsi que dans le cadre des travaux sur la visualisation des graphes d'accès [GAH02]. Cela reste à faire pour la construction automatique proprement dite. La dernière phase du projet MACAO est une phase d'expérimentation au cours de laquelle nous devrions prolonger la mise en pratique.

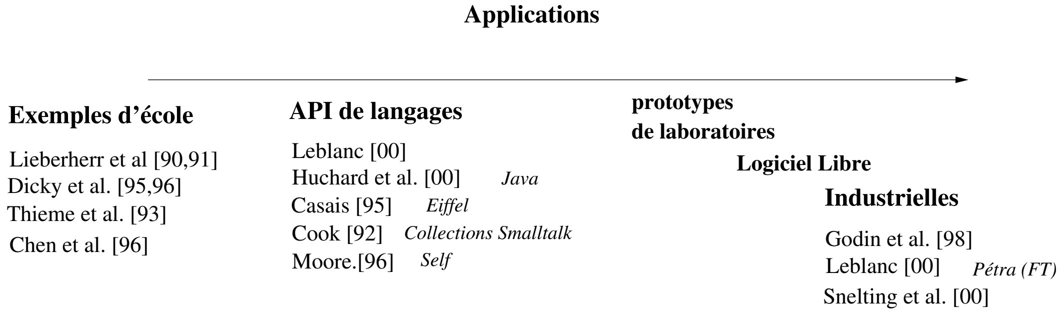


FIG. 5.8 – Applications pratiques connues

5.7 Conclusion

L'une des contributions d'un état de l'art est de clarifier l'existant et de montrer les voies possibles de recherche. Dans [ABC+02, GHRV02], nous avons identifié deux manques essentiels :

- *validation* : des modalités sérieuses de validation restent à définir. Différentes approches peuvent être étudiées telles que les métriques ou la définition d'un *benchmark* spécifique.
- *opérations et utilisation* : de nombreux types d'opérations sont incomplètement étudiés, tout comme l'est aussi peu l'interaction d'un concepteur avec des outils.

L'étude montre l'importance croissante de l'utilisation des treillis de Galois (ou analyse formelle de concepts) en génie logiciel. Cette utilisation est assez diversifiée [ABC+02], les propositions les plus marquantes dans notre contexte étant (seules les références fondatrices sont précisées) :

- *basées sur la possession* la relation binaire associe une classe et les propriétés (attributs/méthodes) qu'elle possède [GM93],

- *basées sur l'usage* la relation mémorise pour chaque variable d'un programme l'ensemble des propriétés (attributs/méthodes) utilisées [ST00],
- *basées sur le comportement* une paire (class,sélecteur de méthode) est associée à un *patron d'appel* [AM02b].

L'étude des références a fait émerger également un domaine connexe intéressant constitué par les techniques mises en œuvre pour la rétro-conception d'une application ou d'un schéma vers l'objet (programmation ou bases de données).

Chapitre 6

Conclusion et perspectives

Arrivé au terme de ce tour d'horizon sur la construction de hiérarchies de classes, il convient de dresser un bilan provisoire et de recenser quelques perspectives qui viennent rappeler ou compléter celles que nous avons présentées tout au long du document.

6.1 La sous-hiérarchie de Galois : une forme normale ?

Nous avons pris appui sur un modèle théorique qui nous apporte la garantie de pouvoir toujours spécifier formellement les opérations que nous effectuons. Il y a deux composantes dans ce modèle, la première consiste à considérer un ordre de spécialisation sur les propriétés (ou référents), la deuxième à utiliser le treillis de Galois (ou le sous-ordre que constitue la sous-hiérarchie de Galois). Cette position nous apporte des propriétés fortes sur le résultat : factorisation maximale, accord entre la spécialisation des classes et celle des référents, accord entre la spécialisation des classes et l'inclusion des intensions, nombre minimal de classes. Elle nous permet d'établir des résultats sur la classification des instances et d'évaluer une hiérarchie quelconque à l'aune de cet idéal. Enfin les modifications par ajout de classes ou de propriétés se comportent comme une croissance « monotone » de la structure, tandis que les opérations de retrait se traduisent par un repli, dans tous les cas sans remise en cause profonde, ce qui nous autorise à penser qu'elle favorise la maîtrise de l'évolution des applications.

Cette approche a bien entendu aussi ses limites. Un premier inconvénient est tout d'abord un regroupement de référents qui peut être excessif, regroupement qui produit des concepts (des classes) de faible cohésion sémantique. Le deuxième désavantage notable est la « sur » généralisation, c'est-à-dire la découverte de trop nombreux concepts parmi lesquels il faut séparer le bon grain de l'ivraie. Il faut également signaler que les résultats ne sont intéressants que si la description initiale des classes est effectuée de manière suffisamment précise et assez systématique notamment en ce qui concerne la terminologie et la structuration des méthodes. Lorsque ces conditions ne sont pas remplies, le résultat en est affecté et devient moins intéressant.

Si la sous-hiérarchie de Galois est donc surtout un idéal théorique, compatible avec l'intuition comme l'a montré notre étude des travaux de recherche existants, il faut la défendre en tant que tel et ne pas en abuser. Je souhaiterais ici reprendre à mon compte le rapprochement que fait R. Godin entre notre démarche et celle de la normalisation des bases de données. De même que les différentes formes normales sont utilisées dans le domaine des bases de données pour guider la production d'un schéma de qualité, la sous-hiérarchie de Galois peut être considérée comme une forme normale. En pratique il est utile de s'écarter de cette structure pour diminuer la complexité d'un schéma ou d'une hiérarchie, mais il me semble qu'elle doit rester le repère dans lequel on travaille.

6.2 Orientations actuelles

6.2.1 Description des modèles

La qualité des résultats dépendant grandement de la précision dans la description des données du problème (classes, associations, attributs, opérations, etc.), il nous semble qu'une réflexion devrait être menée, dans le cadre du langage UML, afin de compléter les modèles par des annotations codifiées dans la notation visuelle. UML propose déjà des notations précisant les discriminants (attributs dont la valeur est la cause de la spécialisation) ou diverses contraintes sur les spécialisations d'une classe (sous-classes disjointes, spécialisation complète, incomplète, etc.). D'autres indications seraient profitables pour appuyer nos hypothèses : expliciter le fait qu'une méthode en spécialise une autre ; admettre des spécialisations covariantes sur le type des paramètres ; préciser les raffinements de domaine des attributs ; décrire les raisons pour lesquelles une classe est spécialisée (les sous-classes représentent différents rôles, correspondent à différents états [MC02]), etc.

D'autres composantes des modèles UML n'ont pas été utilisées dans ces travaux, essentiellement les modèles dynamiques, or certains contiennent des informations précieuses sur les classes. Les diagrammes d'états indiquent le comportement des objets et peuvent révéler des sous-classes intéressantes. Les diagrammes de séquence ou de collaboration décrivent le fonctionnement d'une méthode et peuvent aider à l'identification de spécialisations entre méthodes.

6.2.2 Utilisation des instances

Nous disposons d'une méthode qui permet sans difficulté théorique de construire les classes à partir de la description des objets. Cette capacité pourrait être d'une grande utilité pour étendre la construction de hiérarchies au contexte d'une mise à jour pendant la phase d'exploitation d'une application. Ceci nous rapproche de la problématique de l'émergence de classes à partir d'objets vue par D. Tamzalit et M. Oussalah [TO00].

6.2.3 Développements algorithmiques

Différents développements algorithmiques restent en suspens. Un travail de fond serait utile pour comparer les algorithmes travaillant sur le treillis *versus* sur la sous-hiérarchie et proposer des schémas génériques adaptables à la construction de l'un ou de l'autre. Des raisons d'efficacité en temps de calcul ou en espace, que nous rencontrons avec les implémentations actuelles, peuvent nous amener à rechercher de nouveaux algorithmes basés sur des structures annexes ou sur des principes différents, par exemple construction à la volée et sans mémorisation de parties de la structure. Notre objectif actuel est de proposer un jeu complet d'opérations de manipulation intégré à l'atelier Objecteering.

6.2.4 Intégration dans un atelier de développement

Certaines questions sont liées à l'utilisation des algorithmes au sein d'ateliers de conception ou de développement et demandent d'étudier des scénarios d'interaction. La construction des propriétés génériques est un point crucial dans notre proposition. Elle peut se faire précisément en interaction avec un concepteur, mais l'utilisation d'heuristiques contradictoires peut nous faire pencher vers un énoncé du problème comme une forme de « choix social » où différentes sources (heuristiques et/ou concepteur) doivent s'entendre pour produire un ordre de spécialisation sur les référents.

Nous avons évoqué aussi l'intérêt que nous aurions à intégrer à nos propositions un module linguistique qui mémorise des connaissances sur la synonymie ou l'hyperonymie des termes employés dans un modèle.

La visualisation et le repérage dans des hiérarchies de grandes taille s'avèrent devenir rapidement des problèmes lors des opérations de validation. Même si ce n'est pas notre thème de recherche principal, cela pose des problèmes de décomposition de la hiérarchie ou de construction de vues fort intéressants.

6.2.5 S'écarter du modèle de sous-hiérarchie de Galois

Comme nous l'avons dit précédemment, une utilisation pragmatique de la sous-hiérarchie de Galois consiste à la considérer comme un modèle de référence que l'on améliore, pour le simplifier ou pour augmenter la cohésion des concepts construits, suivant les objectifs précis d'une aplication.

Il me semble cependant important de maîtriser les modifications apportées et en particulier de pouvoir faire le chemin inverse. Différentes solutions sont possibles. Nous en citerons deux ici. Première voie, on peut chercher à définir la hiérarchie finalement utilisée comme une sous-hiérarchie de Galois d'une partie seulement de la relation qui décrit les classes, considérant ainsi que certains attributs sont directeurs dans la construction et les autres plus secondaires. Une deuxième voie possible consisterait à s'inspirer des propositions de [LLJ99] pour la restructuration de schémas en les adaptant au contexte des sous-hiérarchies de Galois. Dans une telle approche, nous accompagnerions la disparition ou le regroupement de classes par des règles (ou leur négation) d'implication, d'équivalence ou d'exclusion mutuelle entre caractéristiques (dans la référence citée ceci est présenté comme des contraintes d'existence), règles qui précisément expriment ce que la hiérarchie construite devrait respecter pour être une sous-hiérarchie de Galois.

6.3 Systèmes à objets et classification

Nous avons montré comment la construction de hiérarchies de classes se présente, du point de vue de la classification, comme un problème de classification de classes et fait quelques propositions à son sujet.

6.3.1 Classification au-delà du modèle à classes

Dans ce mémoire, nous avons suivi le courant du modèle à classes standard. Or la classification est un problème qui se pose dans le modèle alternatif des prototypes ou dans les approches basées sur les aspects. Traiter des prototypes rejoint parfois la problématique que nous avons eue ici comme le montrent les travaux de Y. Moore [Moo95, MC96, Moo96], mais il reste des différences [Bar00] telles que l'absence de différenciation entre variables et méthodes ou l'usage de la délégation plutôt que de l'héritage. Ceci est finalement au bénéfice des prototypes car une partie des problèmes présentés au sujet de l'impact de la classification des classes sur celle des instances disparaissent. Ainsi l'étude des deux modèles pourrait suggérer des processus de développement basés sur les prototypes pendant les phases initiales de conception puis sur le transfert du modèle obtenu dans une représentation avec des classes [Bar00].

Une autre voie très prospective consiste à considérer le problème dans le cadre de la construction ou de la conversion vers une approche par aspects [KLM+97]. L'une des questions soulevées au cours du workshop d'ECOOP 2002 sur l'héritage était justement la recherche ou la combinaison d'aspects *via* l'analyse formelle de concepts.

6.3.2 Classons tout !

Les approches à objets mettent en avant la classification dans d'autres aspects de la modélisation et du développement : métaclasses, patrons de conception (en particulier pour leur indexation), événements, exceptions, diagrammes d'états, scénarios de collaboration entre classes. Cependant les systèmes (langages/modèles) et les environnements ne mettent pas tout en œuvre pour assurer une bonne manipulation des classifications : les instances, nous l'avons dit, sont attachées à des classes sans vérification approfondie ; la cohérence entre la classification des classes et celle des diagrammes dynamiques qui s'y rattachent n'est pas nécessairement assurée ; diverses règles de transformation de classification devraient être clarifiées (par exemple héritage multiple vers héritage simple) ; la navigation dans des bibliothèques de classes ou de patrons reste un sujet de recherche ; la classification des exceptions relativement à celle des classes n'est pas abordée d'une manière satisfaisante [AH01].

6.3.3 Vers la coopération de classifications

Dans les grandes applications actuelles, par exemple les systèmes complexes issus des sciences du vivant ou les systèmes environnementaux, il devient nécessaire d'utiliser de manière coopérative des classifications développées selon des terminologies, des cultures et des objectifs différents. Cela pose différents problèmes relatifs à la description de points de passage entre ces classifications (mise en correspondance des concepts), à l'explicitation des contradictions et des convergences, ou encore à l'intégration dans des classifications virtuelles temporaires que l'on peut comprendre comme des vues.

Tout ceci laisse donc ouvertes de nombreuses et passionnantes questions relatives à la classification dans les approches à objets.

Bibliographie

[ABC+02] G. Arévalo, A. Black, Y. Crespo, M. Dao, E. Ernst, P. Grogono, M. Huchard, and M. Sakkinen. The Inheritance Workshop (W11). In J. Hernandez and A. Moreira, editors, *ECOOP 2002 Workshop Reader*, number 2548 in LNCS. Springer-Verlag, 2002.

[AG98] Ken Arnold and James Gosling. *The Java Programming Language Second Edition*, addison wesley edition, 1998.

[AH01] Y. Ahronovitz and M. Huchard. Exceptions in Object Modeling : Finding Exceptions from the Elements of the Static Object Model. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, number 2022 in LNCS. Springer-Verlag, 2001.

[AH02] G. Ardourel and M. Huchard. Access Graphs : Another View on Static Access Control for a Better Understanding and Use. *Journal of Object Technologies (JOT)*, 1(5) :95, 2002.

[AKBLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions On Programming Languages and Systems*, 11(1) :115–146, january 1989.

[AM02a] G. Arevalo and T. Mens. Analysing Object Oriented Framework Reuse using Concept Analysis. In *Proceedings of the Inheritance Workshop at ECOOP 2002, Publications of Information Technology Research Institute, 12/2002, University of Jyväskylä, Black, Enrst, Grogono, Sakkinen (editors)*, pages 3–9, 2002.

[AM02b] G. Arévalo and T. Mens. Analysing Object-Oriented Application Frameworks Using Concept Analysis. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 53–63. Springer, 2002.

[Arf96] A. Arfi. Un outil pour la conception de hiérarchies de classes dans un modéle orienté-objet. Master's thesis, Université de Montréal, 1996.

[Ast96] H. Astudillo. *Evaluation and realization of modeling alternatives : supporting derivation and enhancement.* PhD thesis, Georgia Institute of Technology, Atlanta, USA, March 1996.

[Ast97] H. Astudillo. Maximizing object reuse with biological metaphor. *Theory and Practice of Object Systems*, 3(4) :235–251, 1997.

[ASZ92] E. Andonnoff, C. Sallaberry, and G. Zurfluh. Interactive design of object oriented databases. In *Proceedings of CAISE'92*, volume 593 of *LNCS*, pages 128–146. Springer-Verlag, 1992.

[Bac01] Roland Backhouse. Galois Connections. slides, http ://www.cs.nott.ac.uk/˜rcb/G53PAL/slides/Lecture6.pdf, 2001.

[Bak91] H. G. Baker. CLOStrophobia : Its Etiology and Treatment. *OOPS Messenger*, 2(4) :4–15, 1991.

[Bar00] Daniel Bardou. Inheritance Hierarchy Automatic (Re)organization and Prototype-Based Languages. In M. Huchard, R. Godin, and A. Napoli, editors, *Proceedings of the workshop "Objects and Classification : a natural convergence (ECOOP 2000) - RR LIRMM 2000-095*, 2000.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, pages 303–311, october 1990.

[BCH+96] K. Barret, B. Cassels, P. Haahrs, D.A. Moon, K. Playford, and P.T. Withington. A Monotonic Superclass Linearization for Dylan. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96.*, 31(10) :69–82, 1996.

[BCK89] H. Brettauer, T. Christaller, and J. Kopp. Multiple versus Single Inheritance in Object Oriented Programming. *Arbeistpapiere der GMD 415*, 1989.

[Ben73] J-P. Benzecri. *L'analyse des données (Tomes 1 et 2)*. Dunod, 1973.

[Ber91] P. Bergstein. Object Preserving Class Transformations. In *Proceedings of OOPSLA'91, Phoenix (AZ), USA*, special issue of ACM SIGPLAN Notices, 26(11), pages 299–313, 1991.

[Ber97] V. Berry. *Méthodes et algorithmes pour reconstruire les arbres de l'évolution.* PhD thesis, Université Montpellier 2, 1997.

[BG95] E. Bertino and G. Guerrini. Objects with multiple most specific classes. In *Proceedings of ECOOP'95*, pages 102–126, Aarhus, Denmark, 1995.

[BG02] Luc Bernard-Gazagne. Implémentation de métriques dans l'atelier Objecteering. Technical report, Rapport de Stage 2ième année, Institut des Sciences de l'Ingénieur de Montpellier, Dpt Informatique et Gestion, 2002.

[BGMM99] E. Bertino, G. Guerrini, I. Merlo, and M. Mesiti. An approach to classify semi-structured objects. In R. Guerraoui, editor, *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 416–440, Lisboa, Portugal, 1999. Springer.

[Bir40] G. Birkhoff. *Lattice theory.* AMS Colloquium Publication, vol. XXV, 1940.

[Bir67] G. Birkhoff. *Lattice theory (third edition).* AMS Colloquium Publication, vol. XXV, 1967.

[BKKK87] J. Banerjee, W. Kim, K.J. Kim, and H. Korth. Semantics and Implementation of Schema evolution object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA, 1987.

[BM70] M. Barbut and M. Monjardet. *Ordre et Classification. Algèbre et Combinatoire*, volume 2. Hachette, 1970.

[Bor86] J. P. Bordat. Calcul pratique du treillis de Galois d'une correspondance. *Mathématiques et sciences humaines*, 96 :31–47, 1986.

[Bor92] J. P. Bordat. *Sur l'algorithmique combinatoire d'ordres finis. Thèse d'état.* Université Montpellier 2, 1992.

[Bou96] Isabelle Bournaud. *Regroupement conceptuel pour l'organisation de connaissances.* PhD thesis, LAFORIA - Université Paris VI, octobre 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, 1999.

[BS02] A. Berry and A. Sigayret. Maintaining Class Membership Information. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 13–23. Springer, 2002.

[BWDP98] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. Technical Report 07, ISERN, Kaiserslautern, Germany, 1998.

[Cap93] C. Capponi. Classification des classes par des types. In *Actes du colloque Représentations Par Objets (RPO93)*, 1993.

[Cas91] E. Casais. *Managing Evolution in Object Oriented Environments : An Algorithmic Approach*. PhD thesis, Université de Genève, 1991.

[Cas92] E. Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP'92, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, Berlin, 1992.

[Cas93] Y. Caseau. Efficient handling of multiple inheritance hierarchies. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10), 1993.

[Cas94] E. Casais. Automatic Reorganization of Object-Oriented Hierarchies. *Object-Oriented Systems*, 1(2) :95–115, 1994.

[Cas95] E. Casais. Managing class evolution in object-oriented systems. In O.Nierstrasz and D.Tsichritzis, editors, *Object-Oriented Software Composition*, pages 201–244. Prentice Hall, 1995.

[Cas97] G. Castagna. *Object-Oriented Programming, a Unified Foundation*. Birkhauser, 1997.

[Cas98] E. Casais. Re-Engineering Object-Oriented Legacy Systems. *Journal of Object-Oriented Programming*, 10(8) :45–52, January 1998.

[CG90] B. Carré and J. Geib. The Point of View notion for Multiple Inheritance. *Special issue of Sigplan Notice - Proceedings of ACM ECOOP/OOPSLA'90*, 25(10) :312–321, 1990.

[CG00] C. Capponi and J. Gensel. Classification of Objects and Tuples : the AROM's approach. In M. Huchard, R. Godin, and A. Napoli, editors, *Proceedings of the workshop "Objects and Classification : a natural convergence (ECOOP 2000) - RR LIRMM 2000-095*, 2000.

[CK91] S.R. Chidamber and C.F. Kemerer. Towards a metric suite for object-oriented design. In *SIGPLAN Notices*, volume 26, pages 197–211, Phoenix (AZ), USA, 1991.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on software engineering*, 20(6) :476–493, 1994.

[CL96] J.-B. Chen and S. C. Lee. Generation and reorganization of subtype hierarchies. *Journal of Object Oriented Programming*, 8(8), 1996.

[CL02a] Samira Si-Said Cherfi and Nadira Lammari. Towards and Assisted Reorganization of Is-A Hierarchies. In Z. Bellahsène, D. Patel, and C. Rolland, editors, *Proceedings of Object-Oriented Information Systems*, volume 2425 of *LNCS*, pages 536–548. Springer-Verlag, 2002.

[CL02b] S. Coulondre and T. Libourel. Towards a New Role Paradigm for Object-Oriented Modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426, pages 44–52. Springer, 2002.

[CMR02] Y. Crespo, J.-M. Marquès, and JJ Rodriguez. On the translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In *Proceedings of the Inheritance Workshop at ECOOP 2002, Publications of Information Technology Research Institute, 12/2002, University of Jyväskylä, Black, Enrst, Grogono, Sakkinen (editors)*, pages 30–37, 2002.

[Coo92] W.R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA'92, Vancouver, Canada*, special issue of ACM SIGPLAN Notices, 27(10), pages 1–15, 1992.

[Dao99] Michel Dao. Evolution of layered object-oriented models. In *Object-Oriented Architectural Evolution.* I.Borne, S.Demeyer and G.H.Galal. ECOOP'99 Workshop Reader, Ana Maria Moreira (eds), Springer-Verlag, LNCS 1743, 1999.

[DDHL94a] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, un algorithme d'Ajout avec REStructuration dans les hiérarchies de classes. Technical report, LIRMM, 1994.

[DDHL94b] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, un algorithme d'Ajout avec REStructuration dans les hiérarchies de classes. *Actes de Langages et Modèles à Objets 94*, pages 125–136, 1994.

[DDHL95] H. Dicky, C. Dony, M. Huchard, and T. Libourel. ARES, Adding a class and REStructuring Inheritance Hierarchies. In *11 ièmes journées Bases de Données Avancées, Nancy*, pages 25–42, 1995.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 251–267, 1996.

[Dec93] C. Decaestecker. Apprentissage et outils statistiques en classification conceptuelle incrémentale. *Revue d'Intelligence Artificielle*, 7(1) :33–71, 1993.

[DHHM94] R. Ducournau, M. Habib, M. Huchard, and ML. Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'94*, 29(10) :164–175, 1994.

[DHL97a] C. Dony, M. Huchard, and T. Libourel. Automatic hierarchies reorganization : an algorithm and case studies with overloading. Technical Report 97279, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, 1997.

[DHL97b] C. Dony, M. Huchard, and T. Libourel. Automatic hierarchy reorganization : an algorithm and case studies with overloading. Technical report, LIRMM, 1997.

[DHL+01a] M. Dao, M. Huchard, H. Leblanc, T. Libourel, and C. Roume. Towards a metric suite for evaluating factorization and generalization in class hierarchies. In M. Piattini G. Poels H. A. Sahraoui F. Brito e Abreu, B. Handerson-Sellers, editor, *Quantitatives approaches in Object-Oriented Software Engineering (workshop ECOOP 2001), Wettelijk Depot D/2001/Geert Poels, uitgever, ISBN 90-806472-1-7*, pages 85–97, Lisbonne, Portugal, 2001.

[DHL+01b] Michel Dao, Marianne Huchard, Hervé Leblanc, Thérèse Libourel, and Cyril Roume. Spécification des critères propres à l'héritage. sous-projet 1.2. Rapport du projet macao (modélisation et audit de composants à objets), Rapport du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, june 2001.

[DHL+02] M. Dao, M. Huchard, H. Leblanc, T. Libourel, and C. Roume. A New Approach to Factorization : Introducing Metrics. In *Proceedings of the International Symposium on Software Metrics*, 2002.

[DHLN99] R. Ducournau, M. Huchard, T. Libourel, and A. Napoli. Aspects classificatoires des systèmes à objets. In A. Napoli F. Le Ber, J.-F. Mari and A. Simon, editors, *Actes des Journées de la Société Francophone de classification (SFC'99)*, pages 45–52, 1999.

[DHLR01a] Michel Dao, Marianne Huchard, Thérèse Libourel, and Cyril Roume. Etat de l'art des critères. sous-projet 1.1. Rapport du projet macao (modélisation et audit de composants à objets), Rapport du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, june 2001.

[DHLR01b] Michel Dao, Marianne Huchard, Thérèse Libourel, and Cyril Roume. Spécification de la prise en compte plus détaillée des éléments du modèle objet uml. sous-projet 4.2. Rapport du projet macao (modélisation et audit de composants à objets), Rapport du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, décembre 2001.

[Did71] E. Diday. La méthode des nuées dynamiques. *Revue de statistique appliquée*, XIX(1), 1971.

[DLNS96] F.-M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Standford (CA), USA, 1996.

[DLPT82] E. Diday, J. Lemaire, J. Pouget, and F. Testu. *Eléments d'analyse des données*. Dunod, Paris, 1982.

[DMB98] C. Dony, J. Malenfant, and D. Bardou. Langages et modèles à objets. etat des recherches et perspectives. In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, *Les langages à prototypes*. INRIA. Collection didactique, 1998.

[DMT01] P. Desfray, T. Marchand, and J. Tulmets. Spécification d'un mécanisme de synthèse et de "reporting" issu de l'analyse de critères. sous-projet 4.1. Rapport du projet macao (modélisation et audit de composants à objets), Rapport du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, juin 2001.

[DP91] B. A. Davey and H. A. Priestley. *Introduction to lattices and orders*. Cambridge University Press, second edition, 1991.

[dt97a] Pétra : Groupe de travail. Modèle de conception de réseaux commutés pour l'ingénierie du trafic. Technical report, CNET, Décembre 1997. DE/DAC/GTR/49.97.

[dt97b] Pétra : Groupe de travail. Modèle générique de réseaux commutés pour l'ingénierie du trafic. Technical report, CNET, Septembre 1997. DE/DAC/GTR/29.97.

[Duc88] R. Ducournau. *Y3, version 3.* Sema Group, 1988.

[Duc96] R. Ducournau. Des langages à objets aux logiques terminologiques : les systèmes classificatoires. Rapport de Recherche 96-030, L.I.R.M.M., 1996.

[Duc02] R. Ducournau. "Real World" as an Argument for Covariant Specialization in Programming and Modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 3–12. Springer, 2002.

[DVH+02] Michel Dao, Jean Villerd, Marianne Huchard, Thérèse Libourel, Hervé Leblanc, and Cyril Roume. Opérations de manipulation de hiérarchies. sous-projet 4.3. Rapport du projet macao (modélisation et audit de composants à objets), Rapport intermédiaire du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, juin 2002.

[Dvo94] J. Dvorak. Conceptual Entropy and Its Effect on Class Hierarchies. *Computer*, 27(6) :59–63, 1994.

[eAM96] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Proc. METRICS 96, Berlin, Germany*. IEEE Computer Society, march 1996.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring : Improving the Design of Existing Code.* Addison-wesley, 1999. Object Technologies Series.

[Fis87] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, (2) :139–172, 1987.

[FP91] D. Fisher and M. Pazzani. Computational Models of Concept Learning. In *Concept formation : Knowledge and Experience in Unsupervised Learning.* Morgan Kaufmann, 1991.

[GAH02] Olivier Gout, G. Ardourel, and M. Huchard. Access Graph Visualization : A step Towards better understanding of static access control. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002. http ://www.elsevier.nl/locate/entcs/volume72.html.

[GC00] R. Godin and T.T. Chau. Comparaison d'algorithmes de construction de hiérarchies de classes. *L'Objet*, 5(3) :321–338, 2000.

[GHRV02] R. Godin, M. Huchard, C. Roume, and P. Valtchev. Inheritance and Automation : Where Are We Now ? In *Proceedings of the Inheritance Workshop at ECOOP 2002, Publications of Information Technology Research Institute, 12/2002, University of Jyväskylä, Black, Enrst, Grogono, Sakkinen (editors)*, pages 58–64, 2002.

[GLF89] J.H. Gennari, P. Langley, and D. Fisher. Models of Incremental Concept Formation. *Artificial Intelligence*, 40 :11–61, 1989.

[GM93] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proceedings of OOPSLA'93, Washington (DC), USA*, special issue of ACM SIGPLAN Notices, 28(10), pages 394–410, 1993.

[GMM95] R. Godin, G. Mineau, and R. Missaoui. Incremental structuring of knowledge bases. In G. Ellis, R.A. Levinson, A. Fall, and V. Dahl, editors, *Proceedings of the 1st International Symposium on Knowledge Retrieval, Use, and Storage for Efficiency (KRUSE'95), Santa Cruz (CA), USA*, pages 179–193. Department of Computer Science, University of California at Santa Cruz, 1995.

[GMM+98] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T.T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Practice of Object Systems*, 4(2), 1998.

[GMMM95] R. Godin, H. Mili, G. Mineau, and R. Missaoui. Conceptual Clustering Methods Based on Galois Lattices and Applications. *Revue d'Intelligence Artificielle*, 9(2) :105–137, 1995.

[Gué90] A. Guénoche. Construction du Treillis de galois d'une Relation Binaire. *Mathématiques et Sciences humaines*, (109) :41–53, 1990.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations.* Springer-Verlag, 1999.

[HCN98] R. Harrison, S. J. Counsell, and T. V. Nithi. An evaluation of the mood set of object-oriented software. Technical report, Department of Electronics and Computer Science, University of Southampton, Southampton, UK, 1998.

[HDL00] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Theoretical Informatics and Applications*, 34 :521–548, January 2000.

[Hil01] Chris Hillman. What is a concept. http ://www.math.washington.edu/~hillman/papers.html, 2001.

[HL00] M. Huchard and H. Leblanc. Computing Interfaces in Java. In *Proc. IEE International conference on Automated Software Engineering (ASE'2000)*, pages 317–320, 11-15 September, Grenoble, France., 2000.

[HLD99] M. Huchard, T. Libourel, and C. Dony. Evolution de hiérarchies de classes par approches algorithmiques. In M. Oussalah, editor, *Génie objet : Analyse et Conception de l'Évolution*, chapter 6, pages 215–255. Hermès Science, 1999.

[HN94] M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In *Proceedings of ORDAL'94*, LNCS. Springer-Verlag, 1994.

[HRV02] Marianne Huchard, Cyril Roume, and Petko Valtchev. When concepts point at other concepts : the case of UML diagram reconstruction. In V. Duquenne, B. Ganter, M. Liquière, E. Mephu Nguifo, and G. Stumme, editors, *FCAKDD 2002, Advances in Formal Concept Analysis for Knowledge Discovery in Databases, Int. workshop ECAI 2002*, pages 32–43, Lyon, juillet 2002.

[HSZKP02] B. Henderson-Sellers, D. Zowghi, T. Klemona, and S. Parasuram. Sizing use cases : How to create a standard metrical approach. In Z. Bellahsène, D. Patel, and C. Rolland, editors, *Proceedings of the 8th conference Object-Oriented Information Systems (OOIS 2002), Montpellier, France, sept 2002*, volume 2425 of *LNCS*, pages 409–421. Springer-Verlag, 2002.

[Huc99] M. Huchard. Classification de classes contre classification d'instances. Evolution incrémentale dans les systèmes à objets basés sur des treillis de Galois. In *Actes de LMO'99 : Langages et Modèles à Objets*, pages 179–196. Hermès, 1999.

[Huc00] M. Huchard. Another Problematic Multiple Inheritance Mechanism : Construction and Desctruction in C++ in the Virtual Inheritance Case. *Journal of Object Oriented Programming (JOOP)*, 13(4) :6–12, july/august 2000.

[IS88] Ingrid and Peter Schönfelder. *Guide de la flore méditerranéenne*. Hatier, 1988.

[Jac70] F. Jacob. *La logique du vivant*. Bibliothèque des sciences humaines, nrf. Gallimard, 1970.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwing. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, 1997.

[KO01] S. Kuznetsov and S. Obekov. Algorithms for the Construction of the Set of All Concepts and Their Line Diagram. In *2001 International Workshop on Concept Lattice-based theory, methods and tools for Knowledge Discovery in Databases*, 2001. http ://www.lattices.org/.

[LA02] J. Llorens and H. Astudillo. Automatic Generation of Hierarchical Taxonomies from Free Text Using Linguistic Algorithms. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 75–83. Springer, 2002.

[Lan96] P. Langley. *Elements of Machine Learning*. Morgan Kaufmann Publishers, San Francisco, California, 1996.

[LBSL90] K.J. Lieberherr, P.L. Bergstein, and I. Silva-Lepe. Abstraction of object-oriented data models. In H. Kangassalo, editor, *Proceedings of the 9th International Conference on the Entity-Relationship Approach (ER'90), Lausanne, Switzerland*, pages 81–94, 1990.

[LBSL91] K.J. Lieberherr, P. Bergstein, and I. Silva-Lepe. From objects to classes : Algorithms for optimal object-oriented design. *Software Engineering Journal*, 6(4) :205–228, 1991.

[LDH00] H. Leblanc, M. Dao, and M. Huchard. Hiérarchies d'interfaces et API du JDK 1.2. Technical report, Rapport d'avancement 4 du contrat 97 1B 602 "Assemblage de composants" (CNET - France Télécom R&D), 2000.

[LDH01] Hervé Leblanc, Michel Dao, and Marianne Huchard. Prototype de générateur automatique d'interfaces. sous-projet 4.1. Rapport du projet macao (modélisation et audit de composants à objets), Rapport du Projet MACAO (Modélisation et Audit de Composants à Objets), France Télécom R&D, LIRMM (CNRS et Université Montpellier 2), SOFTEAM, juin 2001.

[Leb98] Jacques Lebbe. Représentations par objets et classifications biologiques. In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, *Langages et modèles à objets. Etat des recherches et perspectives.* INRIA. Collection didactique, 1998.

[Leb00] Hervé Leblanc. *Sous-hiérarchies de Galois : un modèle pour la construction et l'évolution des hiérarchies d'objets (Galois sub-hierarchies : a model for construction and evolution of object hierarchies).* PhD thesis, Université Montpellier 2, 2000.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. K. Meyrowitz, editor, *Proceedings of the first conference on Object-Oriented Programming Systems, Languages ans Applications (OOPSLA'86), Portland, Oregon*, volume 21 of *ACM SIGPLAN Notices*, pages 214–223, 1986.

[Liq02] M. Liquière. Generalization and Specialization for Object Oriented Languages. In H. Astudillo, M. Huchard, and P. Valtchev, editors, *Late submissions of the workshop MASPEGHI*, number 02087. Springer, 2002.

[LK94] M. Lorentz and J. Kidd. *Object-Oriented Software Metrics, a Practical Guide.* Prentice Hall, 1994.

[LLJ99] N. Lammari, R. Laleau, and M. Jouve. Processus d'optimisation conceptuelle d'un schéma orienté-objet. *L'objet*, 5(1), 1999.

[LM94] Q. Li and D. McLeod. Conceptual Database Evolution Through Learning in Object Databases. *IEEE, Transactions on Knowledge and Data Engineering*, pages 205–224, 1994.

[LS98] M. Liquiere and J. Sallantin. Structural Machine Learning with Galois Lattice and Graphs. In *Proceedings of the 15th International Conference on Machine Learning*, pages 305–313. Morgan Kaufmann Publishers, 1998.

[Mar98] Michele Marchesi. Ooa metrics for the unified modeling langage. In *Proc. 2nd Euromicro Conf. on Software Maintenance and Reengineering, Florence, Italy, 1998*, pages 67–73. IEEE CS Press, 1998.

[MC96] I. Moore and T. Clement. A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies. In *Proceedings of TOOLS-Europe'96, Paris, France*, 1996.

[MC02] E. Marcos and J. M. Cavero. Hierarchies in Object Oriented Conceptual Modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 24–33. Springer, 2002.

[Mey97] B. Meyer. *Object-Oriented Software Construction, Second Edition.* The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[MGG90] Guy Mineau, Jan Gecsei, and Robert Godin. Structuring Knowledge Bases Using Automatic Learning. *Proceedings of the sixth International Conference on Data Engineering*, pages 274–280, 1990.

[MHK99] B. K. Miller, P. Hsia, and C. Kung. Object-oriented architecture measures. In *32nd Annual Hawaii International Conference on Systems Sciences*. IEEE Computer Soc., 1999.

[Mica] Microsoft Corporation. Distributed Component Object Model Protocol-DCOM/1.0, draft, November 1996. http ://www.microsoft.com/oledev/.

[Micb] Microsoft Corporation. The Component Object Model Specification, Version 0.9, October 24, 1995. http ://www.microsoft.com/oledev/.

[Mic99] Sun Microsystems. Java^{TM}Remote Method Invocation : Specifications and tutorials, 1999. http ://java.sun.com/products/jdk/1.2/docs/guide/rmi/.

[Mic00] Michel Liquière. Graal : a machine learning model for some Object Language problem? *Workshop Objects and classification, ECOOP'2000. http ://ecoop2000.unice.fr/Program/Technical/Workshops/w16.html*, 2000.

[Moo95] I. Moore. Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies. In *Proceedings of TOOLS-Europe'95, Versailles, France*, 1995.

[Moo96] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 235–250, 1996.

[MRU90] O. Mariño, F. Rechenmann, and P. Uvietta. Multiple perspectives and classification mechanism in object oriented classification. In *Proc. ECAI'90*, pages 425–430, 1990.

[MS83] R.S. Michalski and R.E. Stepp. Learning from observation : Conceptual clustering. In *Machine Learning, An Artificial Intelligence Approach*, volume I, pages 331–363. Morgan Kaufmann, 1983.

[MS89] M. Missikoff and M. Scholl. An Algorithm for Insertion into a Lattice : Application to Type Classification. *Proceedings of the 3rd Int. Conf. FODD'89*, pages 64–82, 1989.

[Nap00] A. Napoli. Classification and disjunction in object-based representation systems. In M. Huchard, R. Godin, and A. Napoli, editors, *Proceedings of the workshop "Objects and Classification : a natural convergence (ECOOP 2000) - RR LIRMM 2000-095*, 2000.

[Neb90] B. Nebel. *Reasoning and revision in hybrid representation systems.* Number 422 in LNCS. Springer-Verlag, Berlin, 1990.

[NTM99] J. Noble, A. Taivalsaari, and I. Moore. *Prototype Based Programming.* Springer, 1999.

[Obj01] Object Management Group. The Common Object Request Broker : Architecture and Specification. release 2.6, december 2001.

[OJ93] W.F. Opdyke and R.E. Jonhson. Creating abstract superclasses by refactoring. In S.C. Kswasny and J. F. Buck, editors, *Proceedings of the 21st Annual Conference on Computer Science, Indianapolis (IN), USA*, pages 66–72. ACM Press, New York (NY), USA, 1993.

[Ore44] Oystein Ore. Galois connexions. *Trans. Amer. Math. Soc.*, 55 :493–513, 1944.

[PBTL99] N. Pasquier, J. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 19(4) :33–54, 1999.

[Per98] Jean François Perrot. Objets, classes, héritage : définitions. In Roland Ducournau, Jérôme Euzenat, Gérald Masini, and Amedeo Napoli, editors, *Langages et Modèles d'Objets*, chapter 1. INRIA, collection didactique, 1998.

[Pol98] G. Polaillon. *Organisation et interprétation par les treillis de Galois de données de type multivalué, intervalle ou histogramme.* PhD thesis, Université Paris IX-Dauphine, 1998.

[Rat97] Rational Software Corporation. *UML v 1.1, Object Constraint Language Specification*, version 1.1 edition, septembre 1997. ad/97-08-08.

[Rat99] Rational Software Corporation. *UML v 1.3, Notation Guide*, version 1.3 edition, juin 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1991.

[RC00] D. Rayside and G. T. Campbell. An aristotelian understanding of object-oriented programming. In Doug Lea, editor, *Proceedings of OOPSLA'00, Minneapolis (Minesota), USA*, pages 337–353, october 2000.

[Rou01] C. Roume. Mesures expérimentales sur les propriétés génériques. Technical report, Rapport de Recherche LIRMM - n.01141, octobre 2001.

[Rou02a] C. Roume. Évaluation structurelle de la factorisation et la généralisation au sein des hiérarchies de classes : Introduction de métriques. *L'Objet*, pages 151–166, 2002.

[Rou02b] C. Roume. Going from multiple to single inheritance with metrics. In F. Brito e Abreu, M. Piattini, G. Poels, and H. A. Sahraoui, editors, *Proceedings of the sixth ECOOP workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE 2002)*, pages 30–37, 2002.

[RT01] Olivier Raynaud and Eric Thierry. A Quasi Optimal Bit-Vector Encoding of Tree Hierarchies. Application to Efficient Type Inclusion Tests. In *Proceedings of ECOOP'01*, volume 2072 of *LNCS*, pages 165–180, 2001.

[Run92] E. A. Rundensteiner. A Class Classification Algorithm For Supporting Consistent Object Views. Technical report, University of Michigan, 1992.

[Sch01] Phill Schultz. Explaining the term of Galois connection. Mail communication to The Math Forum, http ://mathforum.org/epigone/historia_matematica/jilmendsing, 2001.

[SGM00] H. Sahraoui, R. Godin, and T. Miceli. Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and its Automations ? In *Proceedings of the International Conference on Software Maintenance*, pages 154–162, 2000.

[sI98] Int. standard ISO/IEC 14882. *Programming Languages - C++.* Information Technology Council, 1998. http ://www.ncits.org/cplusplus.htm.

[SN99] A. Simon and A. Napoli. Building Viewpoints in an Object-Based Representation System for Knowledge Discovery in Databases. In *1st International Conference on Information Reuse and Integration, IRI-99*, 1999.

[ST98] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of ACM SIGPLAN/SIGSOFT Symposium on Foundations of Software Engineering*, pages 99–110, Orlando, FL, 1998.

[ST00] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3) :540–582, May 2000.

[ST02] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, Lecture Notes in Computer Science, Malaga, Spain, June 2002. Springer-Verlag.

[Str98] B. Stroustrup. *The C++ Programming Language, 3rd Edition.* ADDISON-WESLEY, 1998.

[Tai93] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming.* PhD thesis, University of Jyväskylä, Finland, 1993.

[TCFR96] F. Tip, J.-D Choi, J. Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, pages 179–197, 1996.

[TO00] D. Tamzalit and M. Oussalah. Allowing Conceptual Emergence through Instance Evolutionary processes. *Int. Journal of Engineering Intelligent Systems*, 8(3) :177–192, 2000.

[Tor02] M. Torgensen. Inheritance is specialization. In *Proceedings of the Inheritance Workshop at ECOOP 2002, Publications of Information Technology Research*

Institute, 12/2002, University of Jyväskylä, Black, Enrst, Grogono, Sakkinen (editors), pages 95–101, 2002.

[TS93] C. Thieme and A. Siebes. Schema Integration in Object-Oriented Databases. In *Proceedings of CAISE'93*, volume 685 of *Lecture Notes in Computer Science*, pages 54–70. Springer-Verlag, 1993.

[TS97] F. Tip and P. Sweeney. Class hierarchy specialization. In *Proceedings of OOPSLA'97, Atlanta (GA), USA*, pages 271–285, 1997.

[UCCH91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3) :223–242, june 1991.

[US87] D. Ungar and R. B. Smith. SELF : The Power of Simplicity. In N.K. Meyrowitz, editor, *Proceedings of the 2nd conference on Object-Oriented Languages, Systems and Applications (OOPSLA'87), Orlando, Floride*, volume 22, pages 227–241. ACM SIGPLAN Notices, december 1987.

[Vil02] J. Villerd. Spécification et implémentation d'opérations d'édition de hiérarchies de classes. Technical report, France Télécom R&D, september 2002. Rapport de stage de l'Institut des Sciences pour l'Ingénieur de Montpellier (ISIM).

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1) :7–87, 1990.

[Wey80] E. Weyuker. The oracle assumption of program testing. 1980.

[YLB97] A. Yahia, L. Lakhal, and J.P. Bordat. Designing Class Hierarchies of Object Database Schemas. In *13 ièmes journées Bases de Données Avancées*, pages 371–390, 1997.

[YLCB96] A. Yahia, L. Lakhal, R. Cicchetti, and J.P. Bordat. iO2 - An Algorithmic Method for Building Inheritance Graphs in Object Database Design. In *Proceedings of the 15th International Conference on Conceptual Modeling ER'96*, volume 1157 of *Lecture Notes in Computer Science*, pages 422–437. Springer-Verlag, 1996.