Construction dynamique d'annuaires de composants par classification de services

Gabriela Beatriz Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

Construction dynamique d'annuaires de composants par classification de services

Gabriela Arévalo *, Nicolas Desnos **, Marianne Huchard ***,
Christelle Urtado **, Sylvain Vauttier **

* LIFIA - Facultad de Informática (UNLP) – La Plata – Argentina
garevalo@sol.info.unlp.edu.ar
** LGI2P / Ecole des Mines d'Alès – Parc scientifique G. Besse – 30035 Nîmes cedex – France
{Nicolas.Desnos, Christelle.Urtado, Sylvain.Vauttier}@ema.fr
*** LIRMM – CNRS et Univ. Montpellier 2 – 161, rue Ada – 34392 Montpellier cedex 5 – France
Marianne.Huchard@lirmm.fr

Résumé. Les annuaires de composants permettent d'indexer et de localiser rapidement les composants selon les services qu'ils offrent. Ils donnent ainsi aux assemblages en cours d'exécution la possibilité d'évoluer dynamiquement par remplacement de composants, en cas de défaillance, ou par intégration de nouvelles fonctionnalités, en réponse à de nouveaux besoins. Dans ce travail, nous visons des méthodes semi-automatiques d'évolution. Nous posons les bases théoriques d'une utilisation de l'Analyse Formelle de Concepts pour une construction incrémentale des annuaires de composants basée sur les définitions syntaxiques des services requis et fournis. Dans ces annuaires, les composants sont organisés de manière plus intelligible et les descriptions externes de composants plus abstraits et plus réutilisables sont suggérées. Mais surtout, cette organisation rend plus efficaces les tâches automatisées d'assemblage et de remplacement.

1 Introduction

Le génie logiciel à base de composants permet de construire des applications par assemblage de composants sur étagère. Pour faciliter ce processus, les composants exposent leur description externe : les interfaces requises et fournies par le composant correspondent à la description syntaxique des services que le composant met à disposition des composants de son environnement ou que le composant s'attend à trouver chez les composants de son environnement pour fonctionner. De précédents travaux sur l'assemblage automatique de composants et sur l'évolution dynamique d'assemblages (Desnos et al., 2006, 2007) nous ont fait ressentir le besoin de disposer d'un annuaire de composants performant. En effet, l'étape de recherche, dans un annuaire, de composants disponibles en bibliothèque et compatibles avec ou substituables à un composant donné n'est pas triviale. En tout état de cause, les annuaires de type « pages blanches », qui sont les plus fréquemment utilisés, ne sont pas adaptés car ils ne sont pas structurés pour permettre le choix de composants compatibles ou substituables.

L'idée de cet article est de proposer les mécanismes de base d'une indexation semi-automatique des composants dans un annuaire de type « pages-jaunes » qui soit efficace pour

la recherche de composants compatibles avec, ou substituables à, un composant donné. Nous cherchons à exploiter l'information contenue dans la description syntaxique des composants afin d'automatiser entièrement la recherche et la connexion de composants, par opposition aux approches intégrant des informations sémantiques fournies manuellement pour indexer les composants. Notre approche s'appuie sur l'Analyse Formelle de Concepts qui permet de pré-calculer un treillis classifiant les interfaces de composants, c'est-à-dire organisant les descriptions de services de telle façon que la recherche en soit naturellement facilitée. Ce treillis fournit ainsi une classification intelligible pour le développeur ou l'architecte et sépare le calcul de compatibilité des services de la recherche qui est réalisée au cours du processus d'assemblage ou d'évolution. Ce treillis peut être utilisé comme un index, aussi bien pour la recherche d'un composant compatible avec un composant donné (en vue d'assemblage), que pour la recherche d'un composant comparable à un composant donné (en vue d'une substitution). Plus encore, l'Analyse Formelle de Concepts permet la découverte de descriptions externes de composants (nouveaux types de composants) qui n'existent pas dans la biblothèque mais sont plus abstraits et réutilisables que les composants existants. Ces nouvelles abstractions peuvent être un plus pour guider les développeurs dans leur processus d'ingénierie ou de réingénierie ainsi que pour enrichir la bibliothèque.

La suite de cet article est organisée comme suit. La Section 2 pose les fondements d'une extension de la théorie des types des langages orientés-objet au typage des composants. Ensuite, après avoir rappelé les bases de l'Analyse Formelle de Concepts et avoir décrit l'exemple utilisé, nous montrons, dans la Section 3 comment construire un treillis de signatures de fonctionnalités et comment l'exploiter pour l'assemblage de composants ou la substitution. Ensuite, la Section 4 généralise ces résultats pour proposer la classification d'interfaces complètes et montrer son utilisation. La Section 5 compare notre approche aux travaux existants et la Section 6 conclut et présente quelques perspectives de recherche.

2 Typage et substituabilité des fonctionnalités

Cette section s'intéresse à la notion de typage et de substituabilité de fonctionnalités dans le cadre des composants. Une partie de la validité d'un assemblage est en effet déterminée grâce à un typage statique, inspiré des langages à objets statiquement typés où la redéfinition d'une opération dans une sous-classe doit vérifier la contravariance des arguments et la covariance du type de retour (Cardelli, 1984). Dans le contexte des langages à objets, les types statiques sont utilisés pour guider une analyse efficace qui réduit considérablement les erreurs dynamiques de types que peut entraîner l'usage intensif du polymorphisme. Dans le contexte des composants, la compatibilité est, en première approche, définie comme la comparaison statique de leurs interfaces, ce qui consiste plus précisément à comparer deux à deux les signatures des fonctionnalités issues de ces interfaces. Nous ne considérons dans un premier temps que les plus petites abstractions (*i.e.,* les fonctionnalités) car le problème de la substituabilité des composants peut être réduit à déterminer si un composant qui fournit (*resp.* requiert) une fonctionnalité peut se substituer à un composant, déjà connecté au sein d'un assemblage, qui fournit (*resp.* requiert) une autre fonctionnalité.

Pour ajouter la notion de direction aux règles de typage statique des langages orientés-objet et ainsi pouvoir comparer entre elles des fonctionnalités requises (*resp.* fournies), le lecteur peut utiliser une analogie avec le mécanisme usuel d'appel de fonction : les fonctionnalités requises sont assimilables à des appels de fonctions alors que les fonctionnalités fournies en sont

plutôt les définitions. Pour pouvoir exécuter une fonctionnalité, l'appel de cette fonctionnalité doit contenir toute l'information attendue, telle que déclarée dans sa signature. La position à laquelle l'appel apparaît dans le code attend, en retour, un résultat dont le type est le type de retour déclaré pour la fonctionnalité.

Les fonctionnalités sont décrites statiquement par leur signature qui comprend leur *nom*, leur *liste de types de paramètres* (types de paramètres IN) et leurs *types de retour* (types de paramètres OUT ; pour simplifier, on se limitera ici à un paramètre de cette sorte, mais cela n'introduit pas de limitation théorique). Ce travail se fixant comme limite que le remplacement entre fonctionnalités est uniquement admis entre fonctionnalités homonymes, nous faisons porter les règles de substituabilité sur les divers points de variation possibles : les variations de types de paramètres IN et OUT (spécialisation et généralisation) et l'ajout de paramètres.

La Figure 1 présente un exemple de différentes signatures possibles pour une fonctionnalité de nom `create`. De façon à pouvoir utiliser la notation usuelle des interfaces pour indiquer la direction, nous considérons sur le schéma des interfaces fictives comprenant chacune une seule signature. Compte tenu des hiérarchies de spécialisation de types (*cf.* Fig. 1(d)), trois cas sont illustrés : la spécialisation des types des paramètres IN (*cf.* Fig. 1(a)), la spécialisation des types des paramètres OUT (*cf.* Fig. 1(b)) et l'ajout de paramètres IN (*cf.* Fig. 1(c)).

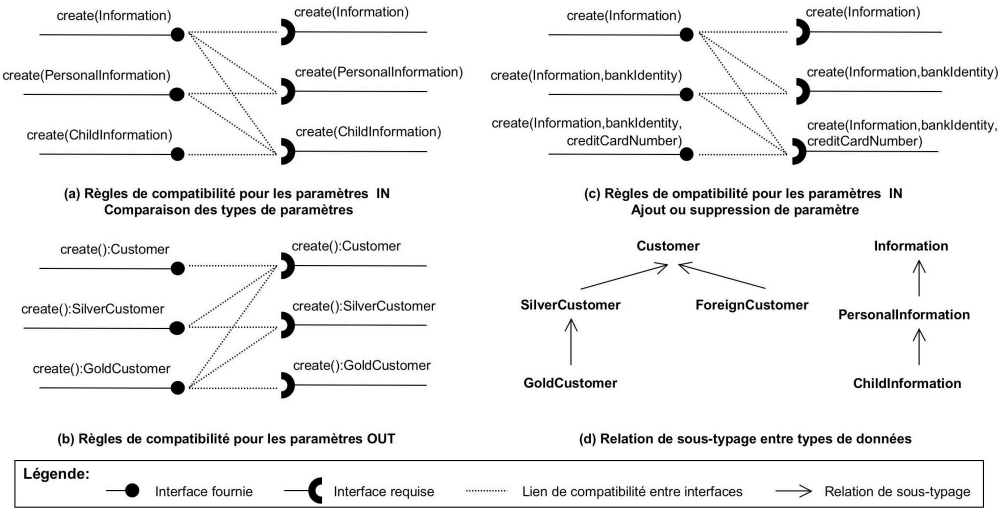


FIG. 1 – *Compatibilité des interfaces lorsque les paramètres varient en type et en nombre.*

De ces exemples d'assemblages possibles sont déduites les règles de substituabilité pour les fonctionnalités requises (à inverser pour les fonctionnalités fournies) :

– *Spécialisation des types des paramètres* IN. Si une fonctionnalité requise (qui s'aparente à un appel) est capable de faire passer un paramètre d'un certain type, elle peut remplacer une fonctionnalité requise qui fait passer un paramètre d'un type plus générique (car la fonctionnalité appelée peut ignorer l'information spécifique au type spécialisé). Dans l'exemple de la Fig. 1(a), la fonctionnalité requise `create(PersonalInformation)` peut remplacer la fonctionnalité requise `create(Information)` car, dans le contexte où la fonctionnalité requise `create(Information)` peut être connectée (c'est-à-

dire à la fonctionnalité fournie `create(Information)`), la fonctionnalité requise `create(PersonalInformation)` peut, elle aussi, être connectée.
Ceci est une application de la règle de substituabilité dans les langages orientés-objet et relève du principe de la contravariance (car, dans le point de vue fourni, tout s'inverse par rapport à l'explication précédente et les types de paramètres IN d'une fonctionnalité redéfinie doivent être généralisés).

– *Ajout de paramètres* IN. Si une fonctionnalité requise est capable de faire passer un paramètre d'un certain type, elle peut remplacer une fonctionnalité qui ne fait pas passer ce paramètre (car la fonctionnalité appelée peut ignorer ce paramètre). Par exemple (*cf.* Fig. 1(c)), la fonctionnalité requise `create(Information,bankIdentity)` peut remplacer la fonctionnalité requise `create(Information)`.
– *Spécialisation des types de paramètres* OUT. Si une fonctionnalité requise a besoin de recevoir un retour d'un certain type, elle peut remplacer une fonctionnalité requise qui a besoin de recevoir un retour d'un type plus spécialisé (puisque l'information excédentaire du type spécifique peut toujours être ignorée). Par exemple (*cf.* Fig. 1(b)), la fonctionnalité requise `create():Customer` peut remplacer la fonctionnalité requise `create():SilverCustomer`. Ceci relève du principe de covariance (si l'on considère à nouveau le point de vue fourni).

3 Treillis des signatures de fonctionnalités

Les règles de substituabilité présentées dans la section précédente peuvent former la base d'une relation de spécialisation entre fonctionnalités : une fonctionnalité qui peut se substituer à une autre peut être considérée comme la spécialisant. On peut donc simplement organiser les fonctionnalités existantes dans une hiérarchie fondée sur leurs relations de substituabilité, mais nous allons montrer que l'Analyse Formelle de Concepts (AFC) va nous permettre d'aller beaucoup plus finement dans leur classification. Nous commençons par rappeler les principes de l'AFC puis nous montrons son application dans la construction d'une hiérarchie de fonctionnalités et les bénéfices obtenus.

Petit kit de survie pour l'AFC. La classification que nous construisons est basée sur la structure partiellement ordonnée appelée *treillis de Galois* (Barbut et Monjardet, 1970) ou *treillis de concepts* (Wille, 1982), induite par un contexte K, composé d'une relation binaire R sur une paire d'ensembles O (*objets*) et A (*attributs*) (Fig. 2). Un concept formel C est défini comme le couple d'ensembles (E, I) tels que :

E = { e $\in O|$ $\forall$ i $\in$ I, (e, i) $\in R$} est appelé *l'extension* (objets couverts)

I= { i $\in A|$ $\forall$ e $\in$ E, (e, i) $\in R$} est appelé *l'intension* (propriétés partagées).

Ainsi, tout objet contenu dans E possède l'ensemble des attributs de I et inversement tout attribut de I est possédé par tous les objets de E ; E et I sont les ensembles maximaux vérifiant cette propriété.

Par exemple, pour la relation représentée sur la Fig. 2, $(\{1, 2\}, \{b, c\})$ est un concept formel car les objets 1 et 2 partagent exactement les attributs b et c (et vice-versa). Par contre,

$(\{2\}, \{b, c\})$ n'est pas un concept formel car l'objet 1 possède, comme l'objet 2, les attributs b et c alors qu'il ne figure pas dans l'extension : l'ensemble des objets n'est donc pas maximal.

L'ensemble $\mathcal{C}$ de tous les concepts formels constitue un treillis $\mathcal{L}$ lorsqu'il est muni de la relation d'ordre (spécialisation) suivante, basée sur l'inclusion des intensions et des extensions :

$(E_1, I_1) \leq_{\mathcal{L}} (E_2, I_2) \Leftrightarrow E_1 \subseteq E_2$ (ou, de façon équivalente, $I_2 \subseteq I_1$).

La Figure 3 montre le diagramme de Hasse de $\leq_{\mathcal{L}}$.

	a	b	c	d	e	f	g	h
1		×	×	×	×			
2	×	×	×				×	×
3	×	×				×	×	×
4				×	×			
5			×	×				
6	×							×

FIG. 2 – *Relation binaire de* $K = (O, A, R)$ où O=$\{1, 2, 3, 4, 5, 6\}$ et A= $\{a, b, c, d, e, f, g, h\}$.

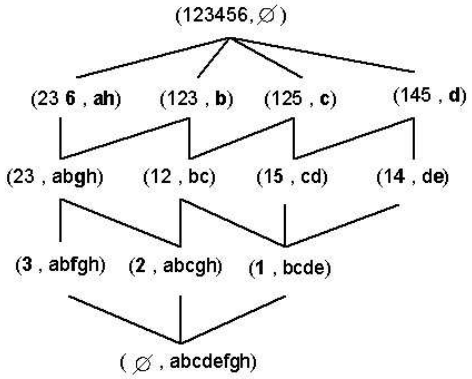
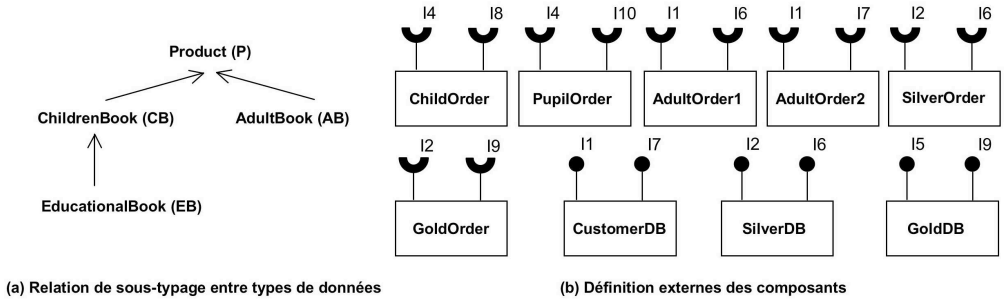


FIG. 3 – *Treillis de concepts* $\mathcal{L}$.

Exemple d'une application de vente en ligne de livres. Dans la suite, nous utiliserons l'exemple d'une application de vente en ligne, dans une boutique spécialisée dans la vente de livres, pour adulte ou pour enfant (*cf.* Fig. 4(a) pour la hiérarchie des types de produits vendus). Deux types de clients potentiels peuvent interagir avec cette application. Les adultes peuvent enregistrer des listes de livres favoris (liste de vœux) dans l'application ou réaliser des achats selon divers protocoles définis en fonction d'une typologie de clients (*cf.* Fig. 1(d)). Les enfants, si leurs parents appartiennent au moins à la catégorie des `SilverCustomer`, ont la possibilité de faire des commandes, sous la forme de listes de livres favoris que les adultes pourront leur offrir, en choisissant parmi les livres pour enfant. Pour cette application de vente en ligne, nous disposons donc d'une bibliothèque de composants (*cf.* Fig. 4(b)) parmi lesquels divers composants pour gérer les commandes (d'adultes ou d'enfants) et divers composants permettant de gérer la liste des clients. Ces composants exposent un certain nombre d'interfaces dont les types sont énumérés dans la Fig. 4(c).

Construction du treillis des fonctionnalités. Nous nous plaçons selon le point de vue des fonctionnalités requises. Comme les signatures de fonctionnalités fournies sont organisées dans l'ordre inverse, le treillis obtenu ici peut aussi être utilisé pour les fonctionnalités fournies s'il est lu à l'envers (de bas en haut). Nous illustrons notre explication en considérant la fonctionnalité `create(PI,BI,CCN):SC` telle que la décrit la Fig. 5 sous le point de vue requis. Dans un premier temps, et pour chaque fonctionnalité `create` dont la signature est contenue

Construction dynamique d'annuaires de composants par classification de services.



(a) Relation de sous-typage entre types de données

(b) Définition externes des composants

I1 | create(Information, bankIdentity, country) : Customer
modify(Information) : Customer

I2 | create(PersonalInformation, bankIdentity, creditCardNumber) : SilverCustomer
modify(PersonalInformation, bankIdentity) : SilverCustomer

I3 | create(PersonalInformation, bankIdentity, creditCardNumber) : GoldCustomer
modify(PersonalInformation, bankIdentity, creditCardNumber) : GoldCustomer

I4 | create(ChildInformation, bankIdentity) : SilverCustomer
modify(ChildInformation)

I5 | create(PersonalInformation) : GoldCustomer
modify(PersonalInformation) : GoldCustomer

I6 | add(Product)
remove(Product)

I7 | add(AdultBook)
remove(Product)

I8 | add(ChildrenBook)
remove(ChildrenBook)

I9 | add(AdultBook)
remove(Product)

I10 | add(EducationalBook)
remove(EducationalBook)

(c) Définitions des types d'interfaces

FIG. 4 – *Types de données, interfaces et composants d'une application de vente de livres.*

dans une des interfaces de la Fig. 4, les attributs sont déduits des types de paramètres d'entrée et de retour qui apparaissent explicitement dans chaque signature. Ces attributs sont marqués par le symbole $\times$ dans la Fig. 5 : `create(PI,BI,CCN):SC` est ainsi décrite explicitement par les attributs `IN:PI`, `IN:BI`, `IN:CCN`, `OUT:SC`. Dans un second temps, nous inférons des attributs (marqués par le symbole $\otimes$ dans la Fig. 5) lorsque leurs types sont compatibles, en fonction des règles de spécialisation des signatures. Voici nos règles d'inférence :

- paramètres IN. Comme expliqué précédemment, si une fonctionnalité requise fait passer un paramètre d'un certain type, elle fait aussi implicitement passer un paramètre de tout type plus générique. Par exemple, l'attribut `IN:I` est ainsi ajouté par inférence en présence de l'attribut `IN:PI`.
- paramètres OUT. Si une fonctionnalité requise attend en retour une valeur d'un certain type, une valeur de retour d'un quelconque type plus spécifique conviendra aussi. Par exemple, l'attribut `OUT:GC` est ainsi ajouté par inférence en présence de l'attribut `OUT:SC`.

Ce codage prend naturellement en compte l'ajout de paramètres car une fonctionnalité, identique à une autre par ailleurs, qui aurait un paramètre supplémentaire en sera naturellement une spécialisation.

La Fig. 6 montre le treillis qui correspond à la relation binaire de la Fig. 5, construit avec l'outil GaLicia (Valtchev et al., 2003). Pour le treillis $\mathcal{L}_{create}$, les concepts sont représentés en utilisant les intensions (ensemble `I`) et les extensions (ensemble `E`) ainsi que les intensions réduites (ensemble `Reduced I`) et les extensions réduites (ensemble `Reduced E`). Ces ensembles réduits sont déduits des ensembles complets de la manière suivante : un objet (signa-

ture) qui appartient à l'extension réduite d'un concept est hérité par tous les concepts qui sont au dessus (de bas en haut) et disparaît de leur extension réduite ; un attribut (type de paramètre IN ou OUT) qui appartient à l'intension réduite d'un concept est hérité par tous les concepts qui sont au dessous (de haut en bas) et disparaît de leur intension réduite. Ainsi, pour associer à un concept la signature de fonctionnalité qu'il représente, le plus simple est de considérer les ensembles réduits. Lorsque l'extension réduite est non vide, elle contient directement la signature de la fonctionnalité que représente le concept. C'est le cas pour les signatures figurant dans l'ensemble O. Lorsque l'extension réduite est vide, cela signifie que le concept représente une signature abstraite qui ne figurait pas dans l'ensemble O. La signature abstraite se calcule en considérant le ou les concepts qui sont au-dessus du concept concerné et en considérant pour chaque attribut le type le plus spécifique figurant soit dans les signatures correspondant aux concepts du dessus, soit dans l'intension réduite du concept même. Les ensembles réduits étant ainsi suffisants pour la compréhension, nous ne montrerons que ceux-ci dans les autres figures représentant des treillis (Figures 8 et 9), pour des raisons de lisibilité.

	IN parameters						OUT param.		
	I	PI	CI	BI	CCN	Co	C	SC	GC
create(I,BI,Co) :C	×			×		×	×	⊗	⊗
create(PI,BI,CCN) :SC	⊗	×		×	×			×	⊗
create(PI,BI,CCN) :GC	⊗	×		×	×				×
create(CI,BI) :SC	⊗	⊗	×	×				×	⊗
create(PI) :GC	⊗	×							×

I	Information
PI	PersonalInfo.
CI	ChildInfo.
BI	BankIdentity.
CCN	CreditCardNb
Co	Country
C	Customer
SC	SilverCustomer
GC	GoldCustomer
FC	ForeignCustomer

FIG. 5 – *Contexte R_{create} décrivant les signatures de la fonctionnalité requise* `create` *par leurs paramètres. Les* × *identifient les paramètres qui figurent explicitement dans les signatures et les* ⊗*, les paramètres inférés.*

Utilisation du treillis des compatibilités entre fonctionnalités. Le treillis des signatures peut être utilisé dans différents types de situations concernant la substitution et la connexion des composants.

Considérons le treillis de la Fig. 6 avec le point de vue des fonctionnalités requises. Dans ce treillis, `create(PI):GC` est représentée par le Concept C_3 alors que `create(CI,BI):SC` est représentée par le Concept C_8. Le Concept C_3 est plus général que le Concept C_8 ce qui peut être interprété comme : le Concept C_8 peut remplacer le Concept C_3. Dans un assemblage de composants, une connection d'une fonctionnalité requise correspondant au Concept C_3 peut être remplacée par une connection d'une fonctionnalité requise correspondant au Concept C_8. Dans le cas général, lorsqu'il y a un chemin entre deux concepts, le plus spécifique (celui qui possède le plus d'attributs) peut remplacer le plus générique (celui qui possède un sous-ensemble d'attributs) lorsque le plus générique est connecté (*cf.* Fig. 7(a)). Le même treillis peut aussi être utilisé pour réaliser la substitution d'une fonctionnalité fournie lorsqu'il est lu à l'envers (*cf.* Fig. 7(b)). Ceci se généralise comme suit.

Substitution de fonctionnalités. *Soient C_{father} et C_{son} deux concepts du treillis $\mathcal{L}_f$ des signatures de fonctionnalités tels que $C_{son} \leq_{\mathcal{L}_f} C_{father}$. Les fonctionnalités requises de C_{son} peuvent remplacer les fonctionalités requises de C_{father}. La règle opposée s'applique dans le cas de fonctionnalités fournies.*

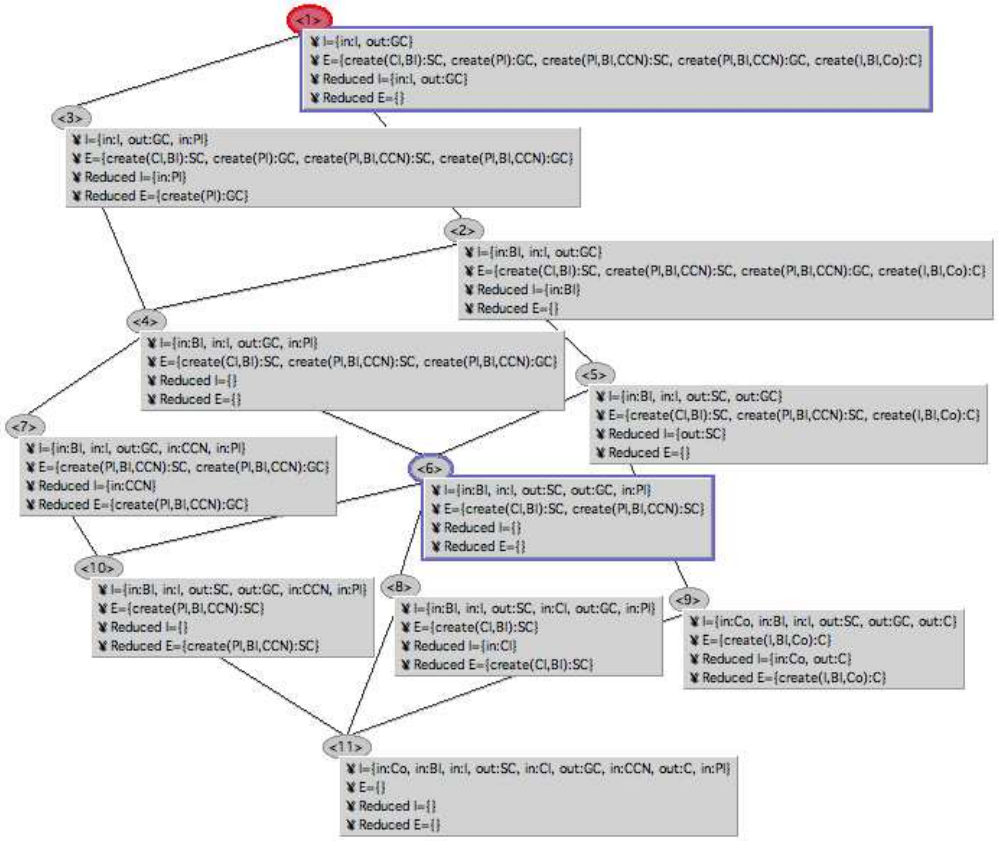


FIG. 6 – *Treillis $\mathcal{L}_{create}$ des signatures de* `create`.

Par ailleurs, les points de vue requis et fourni peuvent être combinés pour traiter la connexion de composants. Si on considère la signature `create(PI,BI,CCN):GC` (concept C_7), la fonctionnalité requise correspondante peut, de toute évidence, se connecter à la fonctionnalité fournie de même signature `create(PI,BI,CCN):GC`. Du fait de la règle de substitution, les fonctionnalités fournies qui sont au-dessus dans le treillis, par exemple la fonctionnalité requise `create(PI):GC`, peuvent être connectées à la fonctionnalité requise `create(PI,BI,CCN) : GC` (*cf.* Fig. 7(c)). En utilisant la même règle mais dans le cas symétrique, les fonctionnalités requises qui sont plus bas dans le treillis, par exemple la fonctionnalité requise `create(PI,BI,CCN):SC`, peuvent être connectées à la fonctionnalité fournie `create(PI, BI,CCN):GC`. La transitivité nous permet de déduire que la fonctionnalité requise `create(PI, BI,CCN):SC` peut se connecter à la fonctionnalité fournie `create(PI):GC`. Cela se généralise de la manière suivante.

Connexion de fonctionnalités. *Soient C, C_{father}, C_{son} trois concepts du treillis des signatures de fonctionnalités tels que $C_{son} \leq_{\mathcal{L}_f} C \leq_{\mathcal{L}_f} C_{father}$, les fonctionnalités requises de C_{son} peuvent être connectées aux fonctionnalités fournies de C_{father}.*

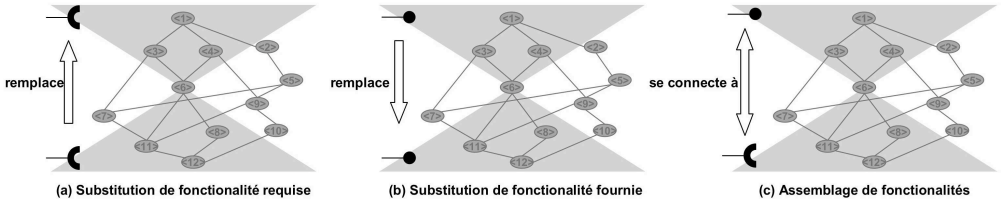


FIG. 7 – *Interprétation du treillis des signatures de fonctionnalités.*

L'utilisation du treillis et des règles d'interprétation est intéressante non seulement pour fournir un index des composants supportant la recherche automatique mais aussi comme visualisation structurée et intelligible / interprétable du contenu d'une bibliothèque de composants.

4 Treillis des interfaces et annuaires de composants

Les composants sont des entités logicielles réutilisables qui peuvent être choisies sur étagère et satisfont un objectif de haut niveau (composant base de données, composant compteur, composant de planification, etc.). Les interfaces jouent un rôle majeur pour atteindre cet objectif, regroupant des fonctionnalités à la sémantique proche et participant ensemble à des collaborations potentielles. L'assemblage des composants se base majoritairement sur une connexion d'interfaces compatibles, à un niveau d'abstraction plus élevé que les simples fonctionnalités.

Les interfaces pourraient être organisées par spécialisation assez naturellement en considérant les fonctionnalités incluses. Cette classification « naturelle » utiliserait simplement les relations d'inclusion entre ensembles de fonctionnalités des interfaces et pourrait également bénéficier de l'AFC en recherchant les fonctionnalités factorisables (ce serait le cas pour `remove(P)` dans notre exemple).

On peut pourtant faire encore mieux et découvrir des abstractions plus pertinentes au regard de la substituabilité ou de la connexion en utilisant les abstractions découvertes lors de la construction des treillis des signatures de fonctionnalités. Les treillis des fonctionnalités `modify`, `add` et `remove` de notre exemple ont été construits de la même façon que le treillis des fonctionnalités `create`. Les résultats sont visibles Fig. 8. Comme nous l'avons vu, ces abstractions sur les signatures sont des concepts dont l'extension contient un ensemble de signatures (les signatures couvertes par le concept) et l'intension contient un ensemble d'attributs décrivant la signature (les paramètres IN et OUT). De chaque concept nous pouvons tirer une signature caractéristique du concept. Nous l'illustrons sur un exemple avant d'en donner la définition générale.

La Figure 6 présente les concepts construits d'après la relation de la figure 5. Un concept dont l'extension réduite contient une signature d'origine (par exemple le concept C_9) représente exactement cette signature (`create(I,BI,Co):C`). Un concept dont l'extension réduite est vide peut s'interpréter comme une nouvelle signature que l'on peut inférer à partir des attributs hérités par le concept, en ne gardant que les plus spécifiques. Par exemple le concept C_6 hérite des attributs `in:I, in:PI, in:BI, out:GC, out:SC`. Dans le cas des signatures requises, `in:PI` est plus spécifique que `in:I` tandis que `out:SC` est plus spécifique que `out:GC`. Le concept C_6 peut donc s'interpréter comme la signature `create(PI,BI):SC` (ce sera la signature caractéristique du concept). Cela permet de constituer une description des

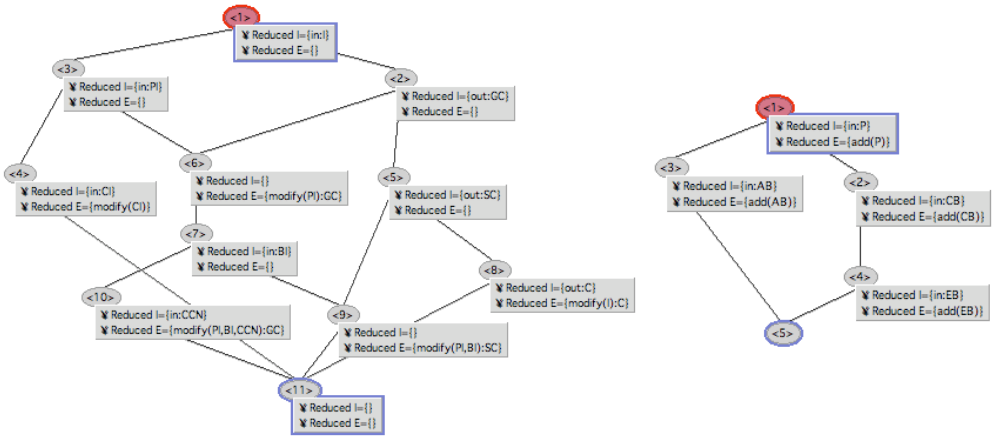


FIG. 8 – *Treillis $\mathcal{L}_{modify}$ et $\mathcal{L}_{add}$. Le treillis $\mathcal{L}_{remove}$, isomorphe à $\mathcal{L}_{add}$, n'est pas représenté.*

interfaces par l'ensemble des signatures d'origine complété par toutes les signatures créées par généralisation (*cf.* Tab. 1).

Signature de fonctionnalité caractéristique d'un concept. *Soit un concept C dans un treillis de signatures $\mathcal{L}_f$ pour la fonctionnalité f et $\leq_{Types}$ l'ordre partiel de spécialisation sur les types des paramètres. Notons $\sigma(C)$ la signature caractéristique de C.*

- *Si $ReducedE(C) = \{s\}$, alors $\sigma(C)= s$.*
- *Si $ReducedE(C) = \varnothing$, alors $\sigma(C)= f(i) : o$ où i est une liste composée des éléments de $min_{\leq_{Types}}\{T|in : T \in Intent(C)\}$ et où $o = max_{\leq_{Types}}\{T|out : T \in Intent(C)\}$.*

Cette description précise permet de construire des généralisations d'interfaces plus pertinentes que celles que nous aurions obtenues avec la classification « naturelle » des interfaces. Elle est utilisée comme suit pour construire la description des interfaces par les signatures de fonctionnalités dans le cadre d'un nouveau contexte $R_{IntSigCar}$.

- Les signatures caractéristiques sont utilisées comme attributs dans le contexte formel.
- Lorsqu'une interface I possède une signature s d'une fonctionnalité f dans sa description d'origine, si on note C le concept tel que $\sigma(C)= s$, on associe à l'interface l'attribut s et toutes les signatures caractéristiques des concepts qui sont au-dessus de C dans le treillis :
  $R_{IntSigCar} = \{(I, sc)|s$ *appartient à la définition de* $I, sc = \sigma(C_{father}), C_{father} \geq_{\mathcal{L}_f} C$ *avec* $s = \sigma(C)\}$.

Par exemple, l'interface `I1` possède la signature `create(I,BI,Co):C`. Cette signature est caractéristique du concept C_9 dans le treillis $\mathcal{L}_{create}$. Dans la Table 1, nous continuons à associer `I1` à `create(I,BI,Co):C` (signalé par le symbole $\times$) et nous associons également à `I1` les signatures caractéristiques de tous les concepts de $\mathcal{L}_{create}$ qui se trouvent au-dessus de C_9. Cela donne les signatures suivantes (signalées par le symbole $\otimes$) : `create(I,BI):SC` (concept C_5), `create(I,BI):GC` (concept C_2), et `create(I):GC` (concept C_1). Du point de vue des fonctionnalités requises, ces signatures sont des généralisations de la signature d'origine `create(I,BI,Co):C` (avec la sémantique présentée précédemment selon laquelle

elle peut les remplacer).

	create(I) :GC — C_1	create(I,BI) :GC — C_2	create(PI) :GC — C_3	create(PI,BI) :GC — C_4	create(I,BI) :SC — C_5	create(PI,BI) :SC — C_6	create(PI,BI,CCN) :GC — C_7	create(CI,BI) :SC — C_8	create(I,BI,Co) :C — C_9	create(PI,BI,CCN) :SC — C_{10}	modify(I) — C_1	modify(I) :GC — C_2	modify(PI) — C_3	modify(CI) — C_4
I1	⊗	⊗			⊗				×		⊗	⊗		
I2	⊗	⊗	⊗	⊗	⊗	⊗	⊗			×	⊗	⊗	⊗	
I3	⊗	⊗	⊗	⊗			×				⊗	⊗	⊗	
I4	⊗	⊗	⊗	⊗	⊗	⊗		×			⊗		⊗	×
I5	⊗		×								⊗	⊗	⊗	
I6														
I7														
I8														
I9														
I10														

	modify(I) :SC — C_5	modify(PI) :GC — C_6	modify(PI,BI) :GC — C_7	modify(I) :C — C_8	modify(PI,BI) :SC — C_9	modify(PI,BI,CCN) :GC — C_{10}	add(P) — C_1	add(CB) — C_2	add(AB) — C_3	add(EB) — C_4	remove(P) — C_1	remove(CB) — C_2	remove(AB) — C_3	remove(EB) — C_4
I1	⊗			×										
I2	⊗	⊗	⊗		×									
I3		⊗	⊗			×								
I4														
I5		×												
I6							×				×			
I7							⊗		×		×			
I8							⊗	×			⊗	×		
I9							⊗		×		×			
I10							⊗	⊗		×	⊗	⊗		×

TAB. 1 – $R_{IntSigCar}$. *Codage des interfaces requises utilisant les généralisations découvertes sur les signatures. En lignes : les interfaces. En colonnes : les signatures caractéristiques et leur concept correspondant dans le treillis associé à la fonctionnalité. Pour simplifier, les signatures caractéristiques correspondant au concept minimum de chaque treillis ne figurent pas dans le tableau car aucune interface ne les possède.*

Le treillis construit (*cf.* Fig. 9) met en évidence des relations de spécialisation entre interfaces. Ces relations indiquent des connexions ou des substitutions possibles par extension, aux interfaces, des règles précédemment indiquées sur les signatures. Par exemple, l'interface requise `I10` (concept C_{11}) peut se connecter avec l'interface fournie `I6` (concept C_2). Ou encore, l'interface requise `I10` peut remplacer l'interface requise `I6`. Une recherche manuelle ou automatique de composants est accélérée par ce treillis qui constitue un index de recherche et évite le parcours total d'un conteneur de composants puisque seules des branches sont parcourues. Imaginons dans le cadre de notre exemple le cas où le composant `GoldOrder` cherche, logiquement, à se connecter au composant `GoldDB` habituellement présent dans le système et que celui-ci soit devenu momentanément indisponible. La relation dans le treillis, en partant de l'interface requise attendue I_9 (C_5), permet de trouver immédiatement (parcours d'un arc vers le concept C_2, contenant l'interface I_6) que le composant `SilverDB` pourra être utilisé en remplacement (au moins temporairement). Cela revient à une dégradation du service car

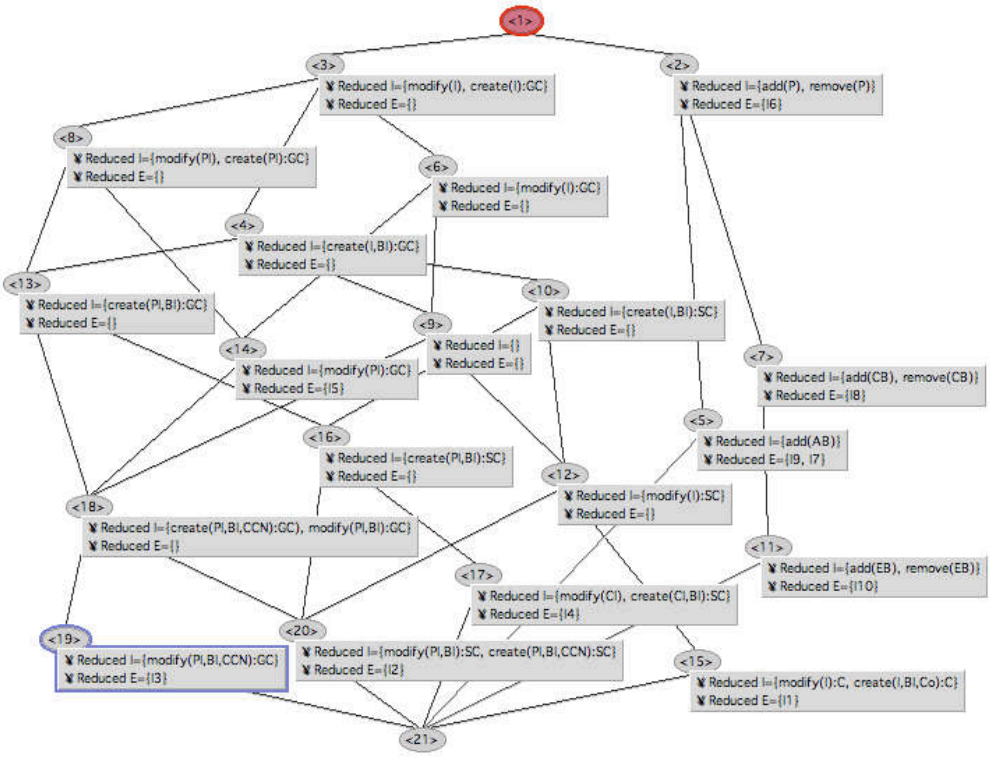


FIG. 9 – *Treillis des interfaces utilisant les treillis de signatures.*

le consommateur ne bénéficiera pas de toute l'offre faite aux *gold customers*, mais il pourra cependant effectuer une commande si elle est urgente.

Dans ce treillis, on trouve également de nouvelles interfaces, obtenues par généralisation des interfaces existantes. A partir d'une fonctionnalité découverte dans le premier treillis, la technique peut donc inférer une nouvelle interface, incluant au moins cette fonctionnalité partagée. Là réside un des avantages principaux des techniques de l'AFC par rapport au simple calcul de comparaison des signatures : de nouvelles signatures émergent, ce qui provoque l'émergence de nouvelles interfaces plus abstraites que les interfaces existantes, etc. L'étape de généralisation suivante serait d'utiliser ce treillis pour construire un treillis de composants et fera l'objet de travaux futurs.

5 Travaux connexes

Peu de travaux utilisent une hiérarchie de types syntaxiques pour structurer l'indexation et la recherche des composants. Zaremski et Wing (1995) suggèrent un mécanisme semblable, mais dans le contexte plus général de l'appariement de signatures de fonctionnalités. La hiérarchie des fonctionnalités est induite par la relation d'ordre partiel définie par l'opérateur d'appariement de signatures, exact ou approché, utilisé. L'appariement entre modules (com-

posants) est défini à partir de l'appariement des fonctionnalités : un composant est comparable à un autre si chacune de ses fonctionnalités peut être appariée avec une fonctionnalité de l'autre. Nos travaux apportent une traduction concrète de ces principes dans le contexte des langages orientés objets fortement typés, où, à l'inverse, les mécanismes d'appariement de signatures sont induits par la hiérarchie des types d'objets. Par ailleurs, l'utilisation d'un treillis de concepts est distinctive : au-delà de l'organisation d'un index des types d'interfaces ou de composants existants, la génération de concepts intermédiaires, suggérant de nouveaux types, offre des perspectives originales à explorer en matière de réingénierie des composants.

Dans le domaine des annuaires de services à pages jaunes, ou service traders (Iribarne et al., 2004), les propositions existantes, comme CORBA Trading Object Service (OMG, 2000), suivent les principes du standard ODP (ODP, 1996). Un composant exporte dans l'annuaire une annonce afin d'y être enregistré comme fournisseur d'un service. L'annonce du service se conforme à un type qui précise les propriétés et les interfaces syntaxiques qu'un composant doit posséder pour fournir ce service. Les types de services sont organisés en hiérarchie de spécialisation mais l'utilisation pratique de cette hiérarchie de types dans l'indexation et la recherche des composants n'est pas spécifiée.

Les travaux qui utilisent l'AFC proposent une méthode d'indexation semi-automatique (Lindig, 1995) pour aider le développeur à identifier les composants adéquats parmi ceux d'une bibliothèque existante. La recherche est fondée sur des groupements de noms et de mots clé et des requêtes incrémentales qui aident à raffiner la recherche. Dans le contexte de la recherche de services web, ces approches sont basées sur des techniques d'apprentissage, permettant la classification et l'annotation de services. (Bruno et al., 2005; Corella et Castells, 2006). A partir d'une documentation textuelle, les services y sont automatiquement regroupés en classes en utilisant des Support Vector Machines ou des Ontologies. L'AFC est ensuite utilisée pour faire correspondre l'information textuelle aux composants à identifier. Contrairement à notre approche, ces travaux utilisent une hiérarchie de types de services prédéfinie et construite statiquement (Marvie et al., 2001). De plus, seule l'information sur les services et interfaces fournis est utilisée. Ce type d'indexation limite les usages de ces annuaires dans les environnements dynamiques, évolutifs et ouverts.

6 Conclusions et perspectives

Nous avons proposé dans cet article une technique de construction d'annuaires de composants basée sur l'AFC. L'annuaire est un treillis dans lequel les interfaces sont organisées de manière à rendre efficace la recherche de composants pour l'assemblage ou le remplacement à la volée. Ce treillis s'appuie sur la construction préalable de classifications des signatures de fonctionnalités également fondée sur l'AFC. Outre ses qualités pour l'aide à la construction et à l'évolution dynamique des assemblages, cette classification des interfaces fait émerger de nouvelles abstractions (généralisations), donnant ainsi la possibilité au développeur de composants d'imaginer de nouveaux composants plus réutilisables.

L'article pose les bases théoriques de cette technique et nous comptons à présent étudier son implémentation sous forme d'un outil semi-automatique d'assistance à la gestion des applications orientées composants. Pour la mise en pratique, nous envisageons quatre étapes :

– *Extraction de l'information sur les interfaces des composants*. Les capacités d'introspection des composants seront exploitées pour collecter *a priori* et maintenir dynamique-

ment l'information sur les interfaces des différents composants en fonction des entrées et des sorties de composants.
- *Codage de l'information dans les contextes formels*. La hiérarchie des types pourra être obtenue par introspection et sera exploitée pour coder les contextes formels (selon les règles d'inférence identifiées ci-dessus).
- *Calcul des treillis*. Parmi les nombreux algorithmes de construction automatique de treillis de concepts figurent des algorithmes incrémentaux (Kuznetsov et Obiedkov, 2002), qui peuvent intégrer de nouveaux concepts dans un treillis existant. De tels algorithmes sont notamment implémentés dans l'outil GaLicia (Valtchev et al., 2003). Ils pourront être utilisés pour calculer *a priori* les treillis et les maintenir dynamiquement en fonction des entrées et des sorties de composants.
- *Exploitation des treillis*. Les treillis obtenus pourront être exploités comme des index sur les composants pour en faciliter la recherche mais aussi, au travers de l'interface graphique de GaLicia, comme visualisation du contenu de la bibliothèque de composants.

Le codage de cet outil nous permettra d'envisager des tests sur des ensembles de composants, réels ou simulés, variés (notamment en nombre ou en granularité des composants et des interfaces et en complexité des signatures).

Des caractéristiques complémentaires des composants, des interfaces et des signatures seront étudiées, telles que les ports, les protocoles et les variations sur le nommage des fonctionnalités ou le passage de paramètres ou la présence d'exceptions. En effet, l'utilisation des informations spécifiant le comportement dynamique des composants dans le mécanisme d'indexation, comme par exemple des définitions de ports tels qu'envisagés dans (Desnos et al., 2006, 2007) pourrait permettre d'effectuer des recherches de composants correspondant à des protocoles de collaboration complets et non à des interactions élémentaires au travers d'interfaces.

D'autres perspectives sont inspirées par les annuaires de Services Web (Klusch, 2008). Ces annuaires de services se distinguent des annuaires de composants par le rôle prédominant donné aux informations sémantiques (noms, descriptions) dans les mécanismes de recherche. Certaines techniques pourraient être expérimentées pour raffiner la classification en prenant en compte le nom des paramètres dans les signatures de fonctions. Inversement, il est intéressant d'étudier comment notre proposition pourrait être utilisée pour améliorer les mécanismes de calculs de compatibilité syntaxique sur des Services Web.

Références

Barbut, M. et B. Monjardet (1970). *Ordre et Classification*. Hachette.

Bruno, M., G. Canfora, M. D. Penta, et R. Scognamiglio (2005). An approach to support web service classification and annotation. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, Washington, DC, USA, pp. 138–143. IEEE Computer Society.

Cardelli, L. (1984). A semantics of multiple inheritance. Volume 173 of *LNCS*, Berlin, pp. 51–64. Springer-Verlag.

Corella, M. A. et P. Castells (2006). Semi-automatic semantic-based web service classification. In *International Workshop on Advances in Semantics for Web Services*, pp. 459–470. Springer Verlag.

Desnos, N., M. Huchard, C. Urtado, S. Vauttier, et G. Tremblay (2007). Automated and unanticipated flexible component substitution. In H. W. Schmidt et al. (Eds.), *Proceedings of the 10^{th} ACM SIGSOFT CBSE*, Volume 4608 of *LNCS*, Medford, MA, USA, pp. 33–48. Springer.

Desnos, N., S. Vauttier, C. Urtado, et M. Huchard (2006). Automating the building of software component architectures. In V. Gruhn et F. Oquendo (Eds.), *Software Architecture : 3^{rd} European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA), Revised selected papers*, Volume 4344 of *LNCS*, Nantes, France, pp. 228–235. Springer.

Iribarne, L., J. M. Troya, et A. Vallecillo (2004). A trading service for COTS components. *The Computer Journal 47*(3), 342–357.

Klusch, M. (2008). Semantic service coordination. In M. Schumacher, H. Helin, et H. Schuldt (Eds.), *CASCOM - Intelligent Service Coordination in the Semantic Web*, Chapter 4. Birkhaeuser Verlag, Springer.

Kuznetsov, S. O. et S. A. Obiedkov (2002). Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence 14*(2-3), 189–216.

Lindig, C. (1995). Concept-based component retrieval. In J. Köhler et al. (Eds.), *Working Notes of the IJCAI-95 Workshop : Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pp. 21–25.

Marvie, R., P. Merle, J.-M. Geib, et S. Leblanc (2001). Type-safe trading proxies using TORBA. In *ISADS*, pp. 303–310.

ODP (1996). *ISO/IEC 13235, ITU-T X.9tr, Information Technology Open Distributed Processing ODP Trading Function*. International Organization for Standardization and International Telecommunication Union.

OMG (2000). *Trading Object Service Specification v1.0*. OMG.

Valtchev, P., D. Grosser, C. Roume, et M. R. Hacene (2003). GALICIA : an open platform for lattices. In *Using Conceptual Structures : Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03)*, pp. 241–254. Shaker Verlag. http ://www.iro.umontreal.ca/∼galicia.

Wille, R. (1982). Restructuring lattice theory : an approach based on hierarchies of concepts. *Ordered Sets 83*, 445–470.

Zaremski, A. M. et J. M. Wing (1995). Specification matching of software components. In *SIGSOFT '95 : Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, New York, NY, USA, pp. 6–17. ACM Press.

Summary

Component directories index components by the services they offer thus enabling to rapidly access them. Component directories are also the cornerstone of dynamic component assembly evolution when components fail or when new functionalities have to be added to meet new requirements. This work targets semi-automatic evolution processes. It states the theoretical

Construction dynamique d'annuaires de composants par classification de services.

basis of an on-the-fly building of component directories using Formal Concept Analysis, based on the syntactic description of the components' required and provided services. In these directories, components are more intelligibly organized and new abstract and highly reusable component external descriptions are suggested. But over all, this organization speeds up both automatic component assembly and automatic component substitution.