Une approche de réingénierie des systèmes d'information basée sur la génération et l'appariement de spécifications de composants

An Information Systems Reengineering Approach based on Generation and Matching of Components Specifications

Oualid Khayati

Laboratoire LSR-IMAG ; 681, rue de la Passerelle, BP. 72, 38402 Saint Martin d'Hères Cedex. France

oualid.khayati@imag.fr

Agnès Front

Laboratoire LSR-IMAG ; 681, rue de la Passerelle, BP. 72,38402 Saint Martin d'Hères Cedex. France

agnes.front@imag.fr

Jean-Pierre Giraudin

Laboratoire LSR-IMAG ; 681, rue de la Passerelle, BP. 72,38402 Saint Martin d'Hères Cedex. France

jean-pierre.giraudin@imag.fr

Résumé

Dans un contexte de réutilisation, un système d'information est un assemblage de composants réutilisables. Généralement, un composant réutilisable est défini comme une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure spécifique et des directives de conception sous la forme de documentation pour supporter sa réutilisation. La documentation du composant illustre le contexte dans lequel le composant peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution. Dans ce cadre les diagrammes UML sont généralement utilisés pour documenter les différentes vues d'un composant (vue dynamique, vue structurelle, vue déploiement, etc.). Le point de départ de ce travail est la proposition d'une technique de génération de spécifications formelles en logique du premier ordre à partir de diagrammes de classes décrivant la structure des composants. Ensuite, le but est de proposer et d'exploiter une technique d'appariement de spécifications dans les domaines de la ré-ingénierie des systèmes d'information et dans le domaine de la recherche de composants.

Abstract

The full potential of components based approaches is reached when large components collections are available to programmers and designers. Reusing components improves software productivity and quality by decreasing development costs, bug risks and development time. A good reuse-oriented information system development environment must be composed of an appropriate components repository. UML diagrams are generally used to document the various sights of a component (dynamic, structural, etc.). This paper presents a technique for formal specifications generation using UML class diagrams. Then, we propose a specification matching technique applied in domains like information system re-engineering, and in components retrieval.

Mots-clés

réutilisation, composant, spécification formelle, ingénierie des systèmes d'information, réingénierie des systèmes d'information, recherche de composant

Keywords

reuse, component, formal specification, information system engineering, information system re-engineering, components retrieval

1. Introduction et problématique

Face à l'émergence de collections de composants réutilisables de différents types (conceptuels, logiciels, etc.), certains environnements professionnels de développement d'applications ont évolué vers une gestion sommaire de composants. Or, si ces outils permettent effectivement de gérer des collections de composants, ils n'en facilitent pas pour autant leur sélection et leur utilisation par une recherche adaptée. Les bases de composants sont des éléments clés dans les environnements de développement à base de composants et de réutilisation de composants. L'efficacité de tels environnements est optimale lorsque les développeurs et les concepteurs disposent de bases riches en composants testés et validés, ce qui augmente la fiabilité et diminue le temps de développement des systèmes d'information résultants. Malheureusement, mettre simplement à disposition des développeurs des bases de composants de taille importante n'augmente pas forcément la productivité. Le concepteur a besoin de techniques performantes de recherche et de classification de composants.
Dans cet article, nous ne nous intéressons pas aux bases de composants uniquement du point de vue recherche de composants mais nous les exploitons pour la construction d'une base de connaissances servant à la réingénierie de systèmes d'information. Nous considérons dans cet article le problème de réingénierie des systèmes d'information comme l'inverse du problème de la recherche de composants. En effet, les composants de la base sont transformés en requêtes et ils sont recherchés dans le code ou dans le modèle de conception du système d'information sur lequel le processus de réingénierie est appliqué.
La deuxième section présente un bref état de l'art des techniques structurelles de recherche de composants. La section 3 introduit le concept de spécification formelle en logique du premier ordre d'un diagramme de classes UML. Puis, les sections 4 et 5 présentent successivement les processus de génération d'une spécification source et d'une spécification cible. La section 6 propose un mécanisme d'appariement de spécifications à base des méta-connaissances. Enfin, la section 7 décrit un outil implantant cette technique ainsi que des exemples de son utilisation.

2. Les techniques structurelles de recherche de composants

Il existe principalement deux types d'approches structurelles de recherche de composants (TRC) : les TRC par appariement de signatures et les TRC par appariement de spécifications.

2.1 Les techniques de recherche de composants par appariement de signatures

Les TRC à base de signatures exploitent les techniques d'appariement et de transformation de types pour retrouver les composants. Dans un cadre standard de développement par objets, un composant est construit par composition de classes. Une classe déclare un ensemble de méthodes, de types et d'attributs. Un composant peut donc être représenté par l'ensemble des signatures des entités qu'il contient. Les signatures peuvent être basées sur des types simples, des types complexes voire des fonctions.
Dans (Ritti, 1989), Ritti propose une technique d'appariement de signatures et applique sa méthode à une collection de composants écrits dans un langage fonctionnel. L'algorithme d'appariement (c'est-à-dire la mesure de la distance) que Ritti adopte utilise un système de type polymorphique et garantit une invariance vis-à-vis de l'ordre des déclarations dans un

type. Dans (Ritti, 1992), Ritti améliore le rappel de sa technique en relaxant la condition d'appariement sur l'isomorphisme de types.
Runciman et Toyn (Runciman et Toyn, 1989) proposent une approche similaire à celle de Ritti en utilisant des types polymorphiques pour définir des critères d'appariement de signatures dans le contexte de la programmation fonctionnelle. L'apport de cette approche consiste dans l'indépendance du critère d'appariement vis à vis du nombre d'arguments des méthodes, ce qui améliore le critère de rappel.
Deux auteurs, Zaremski et Wing, (Zarimski et Wing, 1993) (Zarimski et Wing, 1995) proposent une technique de classification et de recherche de composants basée sur les signatures. Deux signatures sont considérées compatibles si elles sont identiques modulo un renommage et une permutation des noms et des paramètres des méthodes. Une relaxation des critères d'appariement (équivalence) permet d'améliorer le rappel de cette approche. Les auteurs proposent deux types de relaxation : par changement du critère d'appariement ou par transformation des signatures de la requête et des composants. Ces deux types de relaxation peuvent être combinés pour produire d'autres critères d'appariement.

2.2 Les techniques de recherche de composants par appariement de spécification

Les techniques de recherche de composants par appariement de spécifications tentent de retrouver les composants dont les spécifications correspondent (sont compatibles avec) à celle de la spécification requête. Zaremski et Wing ont étendu leurs travaux sur les signatures pour proposer une approche par appariement de spécifications de composants (Zaremski et Wing, 1995a). Ils représentent les requêtes et les composants par un ensemble de paires de pré- et post- conditions (une paire par méthode). Ils définissent aussi un critère général d'appariement défini par la formule suivante :

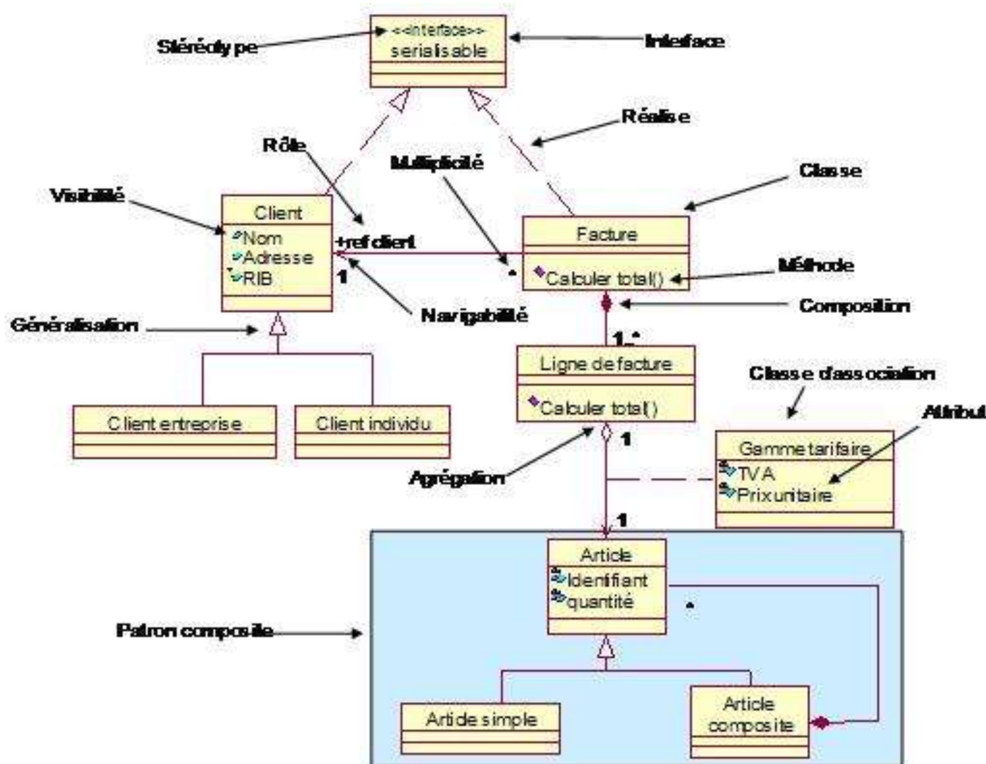
$Match(S,Q)=(Q_{Pré}\ R_1\ S_{Pré})\ R_2\ (Q_{Post}\ R_3\ S_{Post})$
où $S=(S_{pré},S_{Post})$ est la spécification d'un composant,
$Q=(Q_{pré},Q_{Post})$ est la spécification d'une requête
R_1,R_2,R_3 sont trois relations logiquement connectives.

En variant les relations R_1,R_2,R_3 Zaremski et Wing obtiennent une hiérarchie de critères d'appariement.
Rollins et Wing (Rollins et Wing, 1991) présentent une base de composants où les composants sont spécifiés en langage λ-Prolog. Cette technique exploite les mécanismes d'inférence du langage λ-Prolog pour l'appariement des spécifications.
Hemer et Lindsay proposent une base de modules (Hemer et Lindsay, 2001). Un module est un ensemble d'unités. Une unité peut être une procédure, une classe ou une structure de données. Si un développeur a besoin d'un module qui implante toutes les opérations sur les listes chaînées, alors il doit spécifier toutes les primitives dont il a besoin sous la forme d'unités, puis il cherche dans la base le module qui contient ses unités. Hemer et Lindsay proposent trois stratégies de recherche de modules : *ALL-match*, *SOME-match* et *ONE-match*. Le critère *ALL-match* vérifie que toutes les unités de la requête correspondent à des unités distinctes du composant. Le critère *SOME-match* relaxe le critère *ALL-match* en se contentant de vérifier qu'au moins un sous-ensemble non vide d'unités de la requête correspond à un sous-ensemble non vide d'unités du composant. Le critère *ONE-match* vérifie qu'au moins une des unités de la requête correspond à une des unités du composant.

2.3 Conclusion

Nous nous sommes intéressés dans cette section aux approches structurelles de recherche de composants. Les techniques par appariement de signatures sont en réalité un cas particulier des techniques par appariement de spécifications. En effet, les spécifications de techniques par appariement de signatures s'intéressent uniquement aux signatures des composants. Bien que ces deux types d'approches donnent de bons résultats et possèdent des fondements théoriques solides, leur utilisation reste assez difficile et destiné à des spécialistes ayant de bonnes connaissances des langages de spécifications formelles. Nous présentons dans la suite de cet article une technique structurelle de recherche de composants par appariement de spécifications élaborée par notre équipe. L'apport principal de cette technique vis-à-vis des techniques que nous avons présentées est sa transparence et sa simplicité de mis en œuvre. En effet, la technique proposée est basée sur deux processus de génération de spécifications formelles en logique du premier ordre à partir de diagrammes de classes UML ; elle permet à un utilisateur d'indexer ses composants et de spécifier ses requêtes avec des diagrammes de classes.

3. Spécification formelle d'un diagramme de classes UML



La
figure **1** présente un exemple de diagramme de classes types pouvant documenter un système d'information. La notion d'article composite a été modélisée en imitant la solution du patron Composite (Gamma et al., 1995). Si nous voulons faire de la réingénierie de ce système d'information pour détecter les composants qui ont servi à sa construction, le formalisme graphique semi formel d'UML n'est pas bien adapté à cette tâche. Notre but est de passer de la représentation graphique, semi formelle à une représentation formelle facilitant les tâches de comparaison et d'appariement de diagrammes de classes. La logique du premier ordre offre l'avantage de permettre la description de la structure d'un diagramme de classes sous la forme de formules logiques. Le calcul des prédicats en tant que théorie du premier ordre définit des axiomes et des règles d'inférence qui permettent de prouver qu'une formule est

une conséquence logique d'une autre formule. Nous utilisons donc cette technique pour détecter la structure d'un composant (défini par un diagramme de classes cible) dans un système d'information (défini par un diagramme de classes source). Cela est le cas si la formule logique spécifiant le composant est une conséquence logique de la formule spécifiant le système d'information.
Dans l'exemple proposé de diagramme de classes (figure 1), la spécification cible serait la spécification de la solution du patron Composite et la spécification source serait la spécification de tout le diagramme de classes du système d'information. Un prouveur de théorèmes tentera de prouver que la spécification cible est une conséquence logique de la spécification source. Ainsi, il détectera la structure du patron *composite* dans le diagramme de classes de la figure 1. La figure 1 présente les principaux concepts qu'un diagramme de classes UML peut contenir : classe, généralisation, association, réalise, opération, attribut, agrégation, composition, rôle, multiplicité, navigabilité, visibilité, stéréotype, etc.

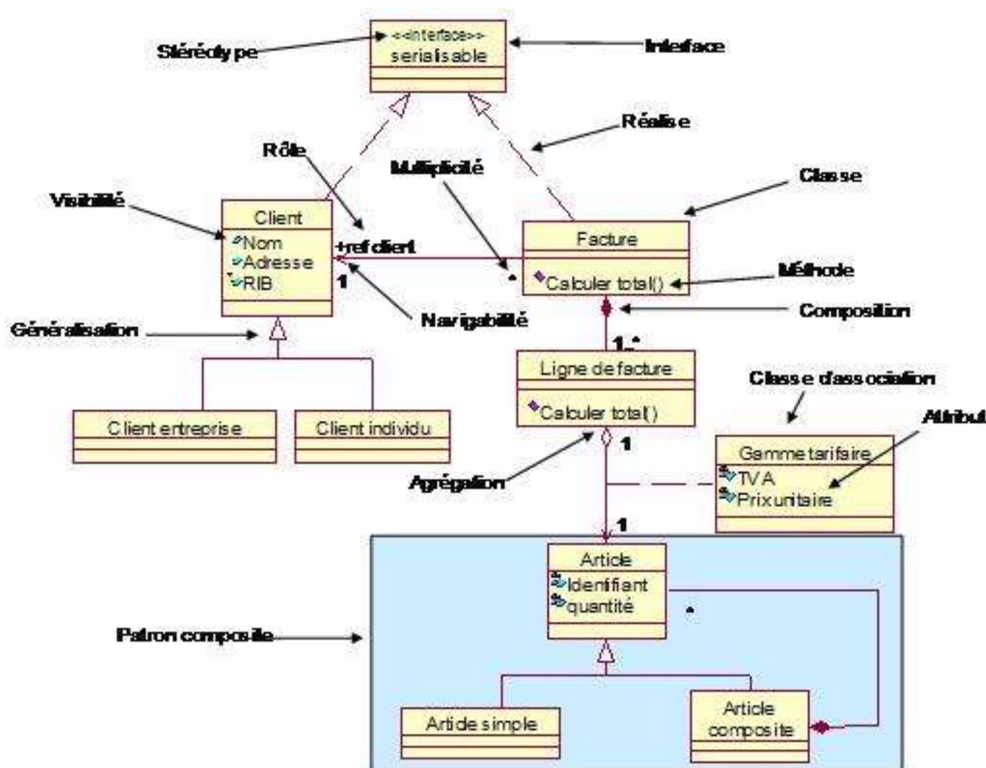


Figure 1. Un diagramme de classes type.

La spécification qui suit décrit par exemple la relation de généralisation qui existe entre la classe Client et la classe Client_individu de la figure 1. Nous utilisons le prédicat *entité*() pour déclarer trois éléments de modélisation(1). Puis, nous utilisons le prédicat classe() pour déclarer les entités id1 et id2 comme étant des classes(2). Ensuite, nous utilisons le prédicat généralisation()(3)pour déclarer l'entité *id3* comme une relation de généralisation entre l'entité *id1* et l'entité *id2*. Enfin, nous assignons des noms aux deux classes identifiées par id1 et id2 en utilisant le prédicat nom_entité()(4).

entité(id1), entité(id2), entité(id3) (1),

classe(id1), classe(id2) (2), généralisation(id3,id1,id2) (3),

nom_entité (id1,'Client'), nom_entité (id2,'Client individu') (4)

Dans la suite de l'article nous adoptons les notations suivantes : nous attribuons aux variables des noms commençant avec des '_' et aux constantes des noms commençant avec des minuscules. Dans la section suivante, nous présentons le processus de génération des spécifications sources.

4. Processus de génération d'une spécification source

Une spécification source est générée à partir d'un diagramme de classes. Nous présentons dans cette section les règles de génération des spécifications sources. Pour chacun des concepts que nous pouvons rencontrer dans un diagramme de classes (OMG, 2003), nous présentons sa définition en termes de prédicats. Il est important de noter que toutes les clauses sont générées avec des constantes ; cela est dû au fait que la spécification source décrit des faits réels qui se trouvent dans le diagramme de classes source.

4.1 Elément de modélisation

Comme nous l'avons montré dans la spécification partielle de la section **Erreur ! Source du renvoi introuvable.**, nous déclarons chacun des éléments de modélisation présents dans un diagramme de classes avec le prédicat entité(). Les noms des entités sont déclarés avec le prédicat nom_entité().

4.2 Classe

Une classe est la description d'un ensemble d'objets qui partagent les mêmes attributs, opérations, associations et la même sémantique. Le prédicat classe() déclare(1) une classe. Une classe est définie par les attributs suivants : nom(2), abstraction(3) (valeur booléenne indiquant si une classe est abstraite).

entité(id_classe), classe(id_classe)(1),nom_entité(id_classe, nom_classe) (2),

classe_abstraite(id_classe) (3)

4.3 Attribut

Un attribut est une propriété structurelle d'une classe permettant à un objet de stocker des informations. Le prédicat attribut() déclare (4) un attribut et le prédicat classe_attribut() associe un attribut à la classe qui le déclare(5). Un attribut est défini (4) par les rubriques suivantes : nom(1), type(2) et valeur par défaut(3).

entité(id_attribut), nom_entité(id_attribut, nom_attribut) (1), attribut(id_attribut) (4)

classe_attribut(id_classe, id_attribut) (5), attribut_type(id_attribut, id_type) (2)

attribut_valeur_par_défaut(id_attribut, valeur_par_défaut) (3)

4.4 Opération

Une opération est une propriété comportementale d'une classe décrivant un service offert par une classe ou une interface. Une opération est déclarée par le prédicat opération()(4) et associée à la classe qui la déclare par le prédicat classe_opération()(5). Une opération possède une signature qui décrit ses paramètres et sa valeur de retour. Une opération est définie par les attributs suivants : nom de l'opération (1), abstraction (2) (une valeur booléenne qui indique si une opération est abstraite ou non) et signature (3).

entité(id_opération), entité(id_signature),

nom_entité(id_opération, nom_opération) (1),

signature(id_signature)(3),opération(id_opération) (4),

classe_opération(id_classe, id_opération, id_signature)(5),

opération_abstraite(id_opération)(2)

4.5 Paramètre

Un paramètre est une variable qui peut être passée à une opération. Le prédicat paramètre() permet de déclarer un paramètre(8) et le prédicat signature_paramètre() permet de l'associer à une signature (9). Il est défini par les attributs suivants : nom(1), type(2), type de communication (in (entrée)(3), out (sortie)(4), in/out (entrée sortie)(5), return (valeur de retour (6)) et valeur par défaut (7).

entité (id_paramètre), paramètre (id_paramètre) (8),

nom_entité (id_paramètre,nom_paramètre)(1),

signature_paramètre (id_signature, id_paramètre) (9),

paramètre_type(id_paramètre, id_type)(2),

paramètre_in(id_paramètre)(3),

paramètre_out(id_paramètre)(4),

paramètre_in_out(id_paramètre)(5) ,

paramètre_result(id_paramètre)6) ,

paramètre_valeur_par_défaut(id_paramètre, valeur_par_défaut)(7)

4.6 Association

Une association définit une relation sémantique entre *Classifiers* (interfaces, classes, etc.). Une association possède un nom(1) et au moins deux instances de *AssociationEnd*. Un seul *classifier* peut être connecté à plus d'un *AssociationEnd* dans une Association. Une association est déclarée avec le prédicat association()(2). Le prédicat *classe_associée*() est utilisé pour déclarer une éventuelle classe d'association.

nom_entité(id_association, nom_association) (1),

entité(id_association) association(id_association),

classe_associée(id_association,id_classe) (2)

4.7 Associationend

associationend spécifie la relation entre une association et un classifier. C'est l'extrémité d'une association. Elle indique les classifiers compatibles avec l'extrémité de l'association. Une associationend est déclarée par le prédicat associationend()(6). Une associationend est définie par les attributs suivants :

- Agrégation prend l'une des trois valeurs suivantes : none pour indiquer une association simple, *aggregate* (3) pour indiquer une agrégation, *composite* (4) pour indiquer une composition,
- Navigabilité(2) est une valeur booléenne indiquant si une *associationend* est navigable,
- Multiplicité (5) donne le nombre d'instances cibles qui peuvent être associées avec une instance source dans une association,
- Nom de rôle(1) représente le nom du rôle de l'*associationend*,
- *Classifier* identifie les *classifiers*, interfaces ou classes, qui sont acceptés par une *associationend*.

nom_entité(id_association_end, nom_rôle) (1),

entité (id_association_end),

associationend(id_association, id_association_end, id_classifier)(6)

associationend_multiplicité(id_association_end, multiplicité)(5)

associationend_navigable(id_association_end) (2)

associationend_agrégation(id_association_end) (3)

associationend_composition(id_association_end) (4)

4.8 Réalise

Réalise est une relation reliant une classe à une interface. La classe doit posséder toutes les méthodes définies dans l'interface. Une relation de réalisation est déclarée par le prédicat *réalise*()(1) et associée à la classe et l'interface qu'elle relie par le prédicat *réalise_classe_interface*()(2).

entité(id_realise), réalise(id_realise) (1),

réalise_classe_interface(id_realise, id_classe, id_interface) (2)

4.9 Stéréotype

Le stéréotypage est un moyen de classification des éléments de modélisation. Un élément de modélisation stéréotypé se comporte comme s'il est une instance d'une méta-classe virtuelle. Un stéréotype est déclaré par le prédicat stéréotype() (3). Il est défini par son nom(2) et l'élément de modélisation qu'il stéréotype. Un stéréotype est associé à un élément de modélisation par le prédicat *stéréotype_élément_modélisation*()(1) .

nom_entité(id_stéréotype, nom_stéréotype)[(2)],

entité(id_stéréotype), stéréotype(id_stéréotype)[(3)],

stéréotype_élément_modélisation(id_stéréotype, id_model_element) [(1)]

4.10 Interface

Une interface est un ensemble d'opérations nommées qui caractérise le comportement d'une classe. Nous faisons le choix de représenter une interface comme étant une classe stéréotypée avec le stéréotype « interface ». Ce choix de modélisation permet d'éviter de redéfinir les méta-connaissances pour l'héritage des attributs, des opérations et des relations. Une interface est définie par son nom et ses opérations.

nom_entité(id_stéréotype_interface, 'interface') [(1)], classe(id_interface),

stéréotype_élément_modélisation (id_interface, id_stéréotype_interface),

nom_entité(id_interface, nom_interface)

(1) (1) l'identifiant du stéréotype « interface » est défini une seule fois dans un diagramme de classes qui définit des interfaces.

4.11 Généralisation

La généralisation est une relation taxonomique entre un élément général et un élément plus spécialisé. L'élément le plus spécialisé est complètement compatible avec l'élément général (il possède ses associations, ses réalisations, ses attributs et ses opérations) et peut contenir d'autres informations. Une généralisation est déclarée par le prédicat généralisation()[(1)]. Une généralisation est définie par les attributs suivants : *classifier* père (interface ou classe), *classifier* fils (interface ou classe).

entité(id_généralisation),

généralisation(id_généralisation, id_père, id_fils)[(1)]

5. Processus de génération d'une spécification cible

Une spécification cible décrit la structure d'un composant qui doit être retrouvé dans une ou plusieurs spécifications sources. Le processus de génération des spécifications cibles reprend exactement les mêmes règles définies dans le processus de génération des spécifications sources en leur appliquant les transformations suivantes :

- toutes les constantes sont remplacées par des variables. Ainsi, le prouveur de théorèmes va rechercher les instances des variables (identifiants, noms d'entité, etc.) qui satisfont la spécification cible. S'il en trouve alors cela signifie qu'il a détecté des instances des entités correspondant à la structure du composant spécifié par la spécification cible ;
- lorsque l'utilisateur désire spécifier des contraintes sur le nom d'une entité alors il place ce nom sous la forme d'une constante dans le prédicat *nom_entité()* qui lui est associé ;
- la clause « ∀ variable » est générée pour chaque variable définie dans la règle

Exemple : la règle de génération qui concerne les interfaces devient

∀ _id_stéréotype_interface , ∀ _id_interface,
nom_entité(_id_interface, 'serialisable')(1),
identifiant(_id_stéréotype_interface, 'interface'),
classe(_id_interface),
stéréotype(_id_interface, _id_stéréotype_interface)

(2) clause générée seulement lorsque l'utilisateur veut sélectionner les interfaces qui ont comme nom 'sérialisable'

6. Mécanisme d'appariement de spécifications à base de méta-connaissances

Le mécanisme d'appariement de spécifications est un algorithme de résolution de formules (spécifications cibles) en logique du premier ordre implanté par un prouveur de théorèmes. Les processus de génération de spécifications présentés dans les sections 4 et 5 spécifient un diagramme de classes sous une forme déclarative en utilisant la logique du premier ordre. Or, un concept comme la généralisation a une sémantique plus large que le simple fait de relier deux *classifiers* (interfaces ou classes). La relation de généralisation propage les informations contenues dans le *classifier* général (attributs, opérations, associations, réalisations) vers ses spécialisations. Lors de la phase d'appariement des spécifications, il faut tenir compte de cette méta-connaissance. L'idée est de permettre à l'utilisateur de choisir les méta-connaissances qu'il veut utiliser lors de la phase d'appariement dans le but d'améliorer les performances du système. Par exemple, s'il s'intéresse uniquement aux relations qui existent entre les classes, il n'a pas besoin de la propagation des attributs et des méthodes induites par les relations de spécialisation. Les méta-connaissances que nous présentons dans cette section traitent les

aspects sémantiques de la généralisation et des techniques de relaxation de l'appariement pour améliorer le rappel.

6.1 Mécanismes de propagation des informations par la généralisation

La relation de généralisation entre deux classes implique la propagation des informations (relations, attributs, opérations) de la classe la plus générale vers la classe spécialisée.

6.1.1. Propagation des attributs

La règle suivante décrit la propagation des attributs dans une généralisation :

∀ _id_généralisation, ∀ _id_classe_générale, ∀ _id_classe_spécialisée, ∀ _id_attribut,
généralisation(_id_généralisation, _id_classe_générale, _id_classe_spécialisée)
^ classe_attribut(_id_classe_générale, _id_attribut)
⇒classe_attribut(_id_classe_spécialisée, _id_attribut)

6.1.2. Propagation des opérations

La règle suivante décrit la propagation des opérations dans une généralisation :

∀ _id_généralisation, ∀ _id_classe_générale, ∀ _id_classe_spécialisée, ∀ _id_opération,
généralisation(_id_généralisation, _id_classe_générale, _id_classe_spécialisée)
^ classe_opération(_id_classe_générale, _id_opération, id_signature_opération)
⇒classe_opération(_id_classe_spécialisée, _id_opération, id_signature_opération)

6.1.3. Propagation des relations

Réalise : Si la classe générale réalise une interface alors la classe spécialisée réalise aussi cette interface.

∀ _id_généralisation, ∀ _id_classe_générale, ∀ _id_classe_spécialisée, ∀ _id_realise,
∀ _id_interface, réalise(_id_realise, _id_classe_générale, _id_interface)
^généralisation(_id_généralisation, _id_classe_générale, _id_classe_spécialisée)
⇒réalise(_id_realise, _id_classe_spécialisée, _id_interface)

Si une classe réalise une interface spécialisée alors la classe réalise forcément l'interface générale.

∀ _id_généralisation, ∀ _id_interface_générale, ∀ _id_interface_spécialisée,

∀ id_realise, ∀ id_classe, ∀ id_interface_spécialisée,

généralisation(_id_généralisation, _id_interface_générale, _id_interface_spécialisée)

^ réalise(id_realise, id_classe, id_interface_spécialisée)

⇒ réalise(id_realise, id_classe, id_interface_générale)

Association : Si un *classifier* (interface ou classe) est relié par une association à un autre *classifier*, alors sa spécialisation l'est aussi.

∀ _id_généralisation, ∀ _id_classifier_générale, ∀ _id_classifier_spécialisé,

∀ _id_association, ∀ _id_association_end,

généralisation(_id_généralisation, _id_classifier_générale, _id_classifier_spécialisé)

^ associationend(_id_association, _id_association_end, _id_classifier_générale)

⇒ associationend(_id_association, _id_association_end, _id_classifier_spécialisé)

La clause de propagation des associations (association simple, agrégation et composition) doit être utilisée avec précaution car elle effectue une fermeture de toutes les associations. Elle doit donc être employée uniquement lorsque l'utilisateur désire retrouver tous les *classifiers* qui sont reliés à un *classifier* source. Dans un contexte de réingénierie simple, il est fortement conseillé de la désactiver.

6.2 Mécanisme de relaxation

Un processus de réingénierie a pour but de détecter les spécifications des composants de la base de composants dans la spécification d'un système d'information existant. Sans le mécanisme de relaxation, le processus d'appariement renvoie un résultat du type « tout ou rien » (composants détectés ou non). Parfois, il est intéressant de retrouver des parties qui ressemblent à quelques détails près à des composants de la base. Nous appelons ce cas de figure un appariement relaxé.

6.2.1. Relaxation des associations

Une agrégation est une association avec la sémantique supplémentaire qu'une classe peut appartenir (par agrégation) à une autre classe ; une composition est une agrégation avec un critère d'exclusivité supplémentaire.
Le fait qu'une composition et une agrégation soient des associations est exprimé par les clauses de génération de spécifications déjà présentées. La clause suivante exprime qu'une composition est un type particulier d'agrégation.

∀ id_association_end,

associationend_composition(id_association_end)

⇒ associationend_agrégation(id_association_end)

6.2.2. Relaxation des types de paramètres des opérations

Lors de la phase d'appariement, il est parfois intéressant de retourner une opération dont la signature est une spécialisation ou une généralisation de la signature recherchée par la

spécification cible. Le prédicat suivant effectue la relaxation par spécialisation et on peut faire de même pour la relaxation par généralisation.

∀ _id_paramètre, _id_type, _id_type_spécialisé,

paramètre_type(_id_paramètre, _id_type)

∧ descendant(id_type, _id_type_spécialisé)

⇒ paramètre_type(_id_paramètre, _id_type_spécialisé)

Le prédicat descendant (id1,id2) exprime que le type id2 est une spécialisation directe ou indirecte du type *id1*.

7. Réalisation et expérimentation

7.1 Contexte pratique

Nos travaux de recherche concernent actuellement l'élaboration d'un environnement de développement de systèmes d'information par réutilisation de composants. Cet environnement est conçu pour permettre aux utilisateurs de puiser des composants dans des bases de composants, de gérer des bases de composants et de gérer le cycle de vie des composants. Il inclut quatre sous-systèmes : un système de gestion de bases descriptives de composants (SGBDC) (Khayati et al., 2004) un système de recherche de composants (SRC), un gestionnaire de cycle de vie des composants (GCVC) et une interface d'administration. La technique que nous proposons s'intègre dans un cinquième outil qui complète cet environnement de développement de systèmes d'information par réutilisation de composants. Cet outil permet de construire une base de spécifications à partir de la base de composants que nous utilisons pour réaliser la réingénierie de systèmes d'information existants.

7.2 Réalisation

La fonctionnalité principale de notre outil est la détection de la structure d'un composant cible (diagramme de classes cible) dans un diagramme de classes source. Le processus de réingénierie que nous proposons (figure **2**) est composé principalement de trois sous-processus : le processus de génération de spécifications cibles, le processus de génération de spécifications sources et le processus d'appariement de spécifications. Chacun de ces processus a été présenté dans les sections 4, 5 et 6.

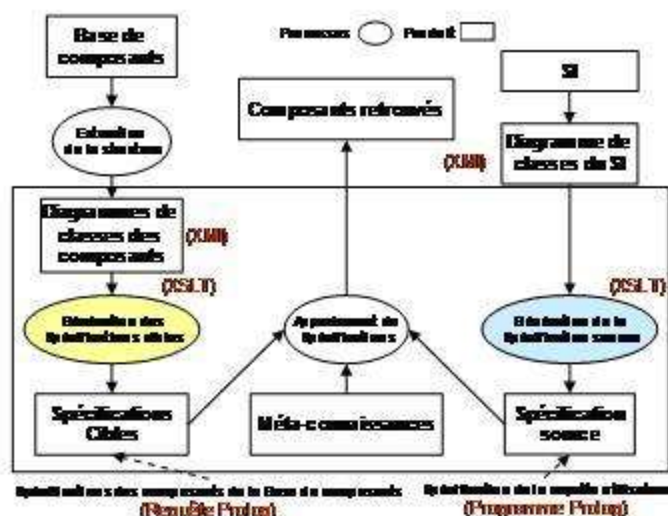


Figure 2. Processus de réingénierie par bases de connaissances.

L'implantation de notre outil utilise la technologie XSL pour transformer les diagrammes de classes (fichiers XMI) en programmes Prolog. Nous utilisons les spécifications sources pour générer les programmes et les spécifications cibles pour générer des cibles. Les règles de génération précédemment présentées sont transformées avec les règles suivantes pour générer des clauses Prolog :

- les clauses d'une spécification source ne sont pas transformées,
- les clauses d'une spécification cible sont transformées à l'aide de la règle

  $\forall$ _x P(_x,b) $\Rightarrow$ Q(_x) se transforme en Q(_x) :-P(_x,b)

Dans un première étape, l'utilisateur l'utilisateur saisit des diagrammes de classes à l'aide d'un éditeur UML (l'AGL ArgoUML). La deuxième étape (figure 3) consiste à transformer ce diagramme de classes en une spécification formelle en Prolog.

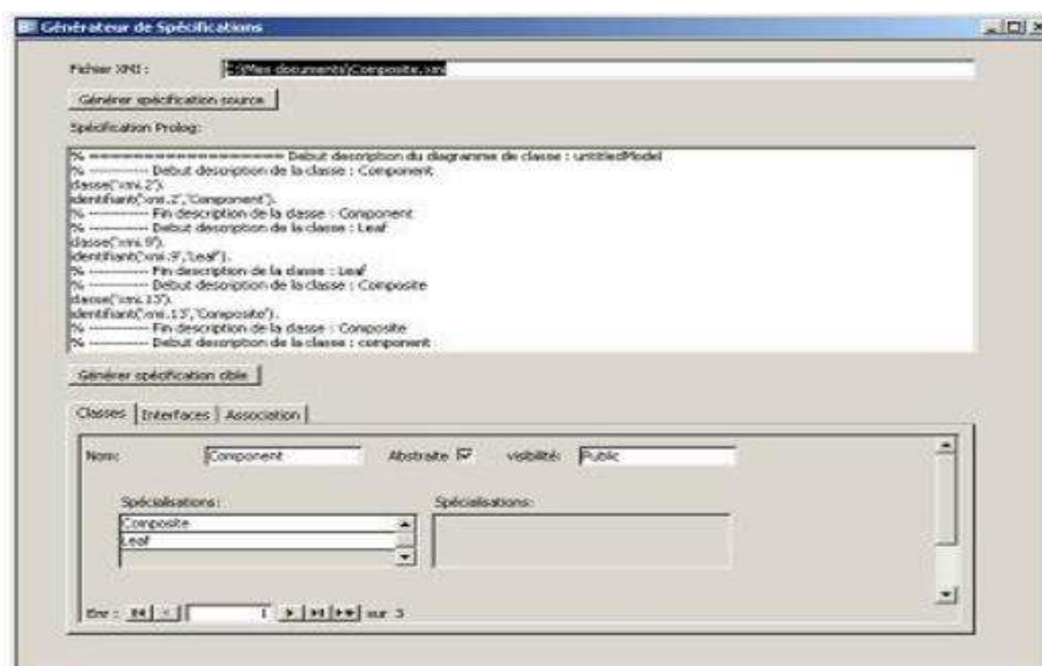


Figure 3. Le générateur de spécifications.

Notre technique peut être exploitée pour réaliser d'autres tâches que la réingénierie d'un système d'information ; par exemple pour assister l'ingénieur de composants pour construire de nouveaux composants en réutilisant des composants existants dans la base de composants, pour faire de la recherche de composants et pour promouvoir la réutilisation. Nous présentons dans les sous-sections suivantes ces trois cas d'utilisation.

7.2.1. Construction de composants

Si nous remplaçons les diagrammes de classes extraits à partir du système d'information par un diagramme de classes exprimant les besoins d'un ingénieur de composants (figure **2**), le processus d'appariement proposera une collection de composants qui aidera l'ingénieur de composants à la construction d'un composant répondant à ses besoins, partiellement ou totalement. Ainsi, l'outil contribue à promouvoir la construction de nouveaux composants et donc la réutilisation de composants.

7.2.2. Recherche de composants

En permutant les processus de « génération de spécifications cibles » et de « génération de spécifications sources » (figure **2**), nous obtenons un système de recherche de composants par appariement de spécifications (figure 4). L'outil utilisera le scénario classique d'interrogation de bases de composants à base de cibles. La structure du diagramme de classes utilisateur sera recherchée dans la base de composants.

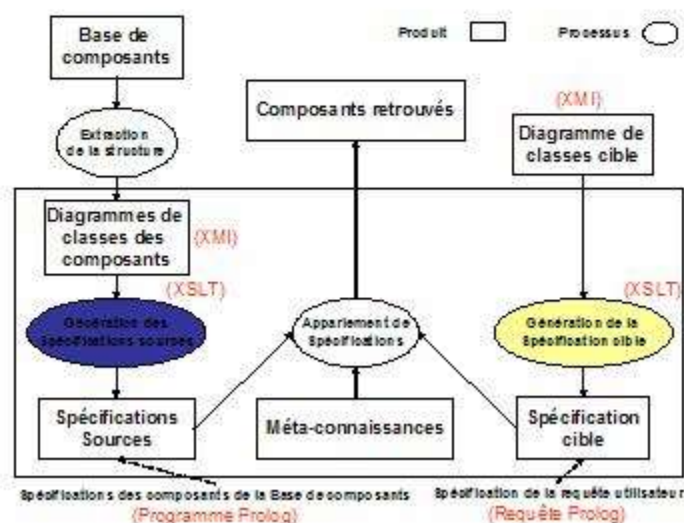


Figure 4. Processus de recherche de composants.

7.2.3. Promotion de la réutilisation

La technique proposée permet de promouvoir la réutilisation auprès des ingénieurs d'application. Son intégration dans un éditeur UML procure à l'éditeur la faculté d'analyser le diagramme de classes en cours de construction. Une fois une structure proche de celle d'un composant de la base de composant détectée (grâce à des techniques de relaxation), l'éditeur pourra proposer à l'ingénieur d'application de réutiliser le composant.

8. Conclusion

Nous avons présenté dans cet article une technique de spécification structurelle de diagrammes de classes UML en logique du premier ordre. Dans les sections 4 et 5, nous avons décrit des processus de génération des spécifications cibles et sources. Dans la section 6, une technique d'appariement de spécifications a été développée pour exploiter le calcul des prédicats pour prouver que les spécifications cibles sont une conséquence logique des spécifications sources. La technique d'appariement utilise un ensemble de méta-connaissances pour diriger le processus de preuve. Enfin, la section 7 décrit l'outil qui implante notre technique et les différents champs d'application de cette technique.
L'intérêt de notre approche par rapport à celles citées réside dans le fait que nous nous intéressons à la structure et aux signatures des composants. De plus, notre approche est applicable à des composants conceptuels et logiciels. Le processus de réingénierie est donc applicable aux niveaux conceptuel et logiciel. La technique proposée est souple et plus facile à utiliser puisque l'utilisateur ne manipule pas (ou presque pas) les spécifications logiques. En effet, il décrit ses cibles sous forme de diagrammes de classes et l'outil génère la spécification automatiquement. Nous devons désormais poursuivre l'étude de cette technique de génération et d'appariement de spécifications structurelles et nous nous sommes fixées des objectifs à court et à long terme. À court terme, il convient de tester l'efficacité de notre approche sur une large collection de composants. Cette expérimentation permettra d'identifier de nouvelles méta-connaissances spécifiques à la tâche dans laquelle notre technique est utilisée. À long terme, il est nécessaire d'étendre cette technique à tout le méta-modèle UML (OMG, 2003) et de traiter les autres types de diagrammes UML.

Références

Gamma E., Helm R., Johnson R., Vlissides J., (1995), *Design patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
Hemer D., Lindsay P., (2001), Specification-based Retrieval Strategies for Module Reuse, in *proceedings 2001 Australian Software Engineering Conference-IEEE Computer Society*, Canberra, Autriche, 27-28 Août 2001, 235-243.
Khayati O., Front A., Giraudin J.P., (2004), A metamodel for a metatool to describe and manage components, in *proceedings of IEEE International Conference on Information & Communication Technologies: from Theory to Applications* (ICTTA'04), Damscus, Syria 19-23 Avril 2004, 403-405.
OMG, 2003, Object Management Group (OMG), Unified Modeling Language (UML) Specification 1.5.
Ritti M., (1989), Using types as search keys in function libraries, In Proceedings of Conference on Functional Programming Languages and Computer Architectures, Addison Wesley.
Ritti M., (1992), *Retrieving library identifiers via equational matching of types*, Technical Report 65, Programming Methodology Group, Dept of Computer Science, Chalmers University of Technology and University of Goteborg, Goteborg, Sweden.
Rollins E.J., Wing J.(1991), specifications as search keys for software libraries, *in Proceedings of the Eighth International Conference on Logic Programming*, 173-187, http://citeseer.ist.psu.edu/434412.html.
Runciman C., Toyn I., (1989), Retrieving reusable software components by polymorphic type, *In proceedings of Conference on Functional Programming Languages and Computer Architectures*, Addison Wesley.
Zaremski A.M., J.M. Wing, (1993), Signature matching: A key to reuse, in *Proceedings of SIGSOFT'93: ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Redondo Beach, Etats-Unis, Décembre.
Zaremski A.M., J.M. Wing, (1995), Signature matching: A tool for using software libraries, *ACM Transactions on Software Engineering and Methodology*, 4(2):146-170, Avril
Zaremski A.M., J.M. Wing, (1995a), Specification matching of software components, *In Proceedings of SIGSOFT'95: third ACM SIGSOFT Symposium on the Foundations of Software engineering*, New York, Etats-Unis: ACM Press, Octobre