Vers une implémentation flexible des activités de maintenance et d'évolution du logiciel

Y Maweed, M Bouneffa, H Basson, P M Oumoum

Vers une implémentation flexible des activités de maintenance et d'évolution du logiciel

Y. Maweed, M. Bouneffa, H. Basson et P.M. Oumoum sack

Laboratoire d'Informatique du Littoral
Maison de la recherche Blaise Pascal
50 Rue Ferdinand Buisson, BP 719
62228, Calais, France
{maweed, bouneffa, basson, oumoumsack}@lil.univ-littoral.fr

RÉSUMÉ. *Nous présentons une approche d'assistance à l'évolution du logiciel basée sur un modèle intégré de représentation des divers artefacts logiciels fondé sur les graphes typés et attribués ainsi qu'une représentation de ces graphes moyennant un langage de structuration des hyperdocuments GXL (Graph eXtensible Language). Ces hyperdocuments GXL sont utilisés pour faciliter l'interopérabilité des outils destinés à traiter les divers aspects de l'évolution du logiciel. Nous utilisons également les systèmes de réécriture de graphes pour une implémentation simple et flexible de mécanismes nécessitant des capacités de raisonnement pour la gestion de l'évolution. Notre approche est illustrée par deux expérimentations que nous avons réalisées sur l'extraction des architectures d'applications multi-tiers Java J2EE et sur la gestion de l'impact du changement de ces applications.*

ABSTRACT. *In this paper we present an approach of assistance at the evolution of the software based on an integrated model of representation of the various software artefacts founded on the typed and attributed graphs as well as a representation of these graphs using GXL (eXtensible Graph Language) a language for structuring hyperdocuments. The hyperdocuments GXL are used to facilitate the interoperability of the tools intended to represent and handle various aspects of the software evolution. We also use the graph rewriting systems for a simple and flexible implementation of mechanisms requiring of the capacities of reasoning for the management of the evolution. Our approach is illustrated by two experiments that we realized on the recovery of applications developed according to multi-tiered architecture Java J2EE and on the management of the change impact of these applications.*

MOTS-CLÉS : *Evolution du logiciel, recouvrement d'architecture, système de réécriture de graphes, Interopérabilité, GXL*

KEYWORDS: *software evolution, architecture recovery, graph rewriting systems, Interoperability, GXL.*

1. Introduction

La maintenance et l'évolution des systèmes informatiques sont des activités cruciales et critiques du fait que ces systèmes doivent constamment évoluer pour répondre aux changements continus des besoins fonctionnels, qualitatifs et s'adapter aux évolutions technologiques. Cependant, du fait de la complexité intrinsèque de la structure et du comportement du logiciel, les activités d'évolution et de maintenance sont dans la plupart des cas prohibitives en coûts et sources d'anomalies. Pour prévenir les effets de bord à l'issue de la mise en œuvre d'une évolution et réduire ses coûts, il est nécessaire de disposer d'outils assistant les chargés de la gestion de l'évolution dans l'accomplissement des activités d'analyse, de réalisation, et de validation induites par l'évolution. Ces outils automatisés sont destinés, à aider à la compréhension de la structure et du comportement souvent complexes des systèmes logiciels, et à identifier *a priori* les impacts du changement ou pour aider à la restructuration du logiciel, etc. Il existe actuellement des outils individuels ou intégrés dans des ateliers de génie logiciel remplissant différentes tâches relatives à l'évolution. Dans ce papier, nous essayons de répondre à deux questions ayant comme point commun la flexibilité du processus de gestion de l'évolution.

La première question concerne la flexibilité de la construction d'une famille d'outils constituants d'un atelier d'aide à la gestion de l'évolution, et la seconde concerne la flexibilité de certains processus comme la génération d'abstractions telles que les architectures logicielles à partir de codes source ou l'identification de la propagation des impacts du changement d'artefacts logiciels. En réponse à ces deux questions, nous proposons deux mécanismes principaux : le premier vise une interopérabilité simplifiée des outils du génie logiciel à travers l'expression des artefacts selon le standard GXL (Graph eXchange Language) (Holt *et al*, 2000) proposé pour l'échange de graphes modélisés et stockés sous la forme d'hyperdocuments XML (W3C, 2000). Le second s'appuyant sur la réécriture de graphes (Ehrig *et al*, 1999) est adopté comme un moyen d'implémentation combinant des capacités de raisonnement sur le changement des artefacts logiciels pour aider les tâches sous-jacentes à la gestion et la mise en œuvre de l'évolution du logiciel. Notre approche sera décrite en illustrant son application par des expérimentations que nous avons réalisées lors du développement d'un atelier réalisant les deux principales tâches qui sont le recouvrement, ou extraction d'architectures logicielles à partir de codes sources, et l'identification des chemins de propagation des impacts du changement du logiciel. L'expérimentation s'est faite sur des applications distribuées multi-tiers de type Java J2EE (Sun Microsystems, 2002).

Le papier est organisé comme suit : la deuxième section introduit les notions de graphes et les systèmes de réécriture ou transformation de graphes. Elle introduit également le standard GXL actuellement adopté par les communautés scientifiques et industrielles comme standard d'échange de graphes entre outils du génie logiciel ou tout autre outil utilisant les graphes. La troisième section présente l'architecture générale de notre atelier. La quatrième section présente le processus que nous avons

mis en œuvre dans le cadre de l'atelier pour le recouvrement ou extraction des architectures logicielles et qui est basé sur l'utilisation d'un système de réécriture de graphes. La cinquième section décrit le processus de gestion de l'impact du changement et surtout le processus de propagation de ce type d'impact. La sixième section conclut ce travail et présente ses perspectives.

2. Les graphes et les systèmes de réécriture de graphes

Notre approche est basée essentiellement sur l'utilisation des graphes, de la représentation et l'échange de ces graphes en forme d'hyperdocuments XML (Manolesku, 2000) et des systèmes de réécriture ou transformation de graphes. Nous allons donc faire une brève présentation de ces trois concepts.

2.1. Définition d'un graphe

- Un graphe orienté est un couple G=<E, F> formé de deux ensembles. L'ensemble E représente les nœuds du graphe et F l'ensemble des arcs qui est un sous-ensemble du produit cartésien E X E.
- Les nœuds et les arcs d'un graphe G sont typés et attribués. Un type de nœud ou d'arc dénote une appartenance du nœud ou de l'arc à des sous-ensembles caractérisés de E ou de F. Un attribut est une valeur éventuellement complexe décrivant des caractéristiques ou propriétés d'un nœud ou d'un arc.

2.2 Les règles de réécriture de graphes

Les systèmes de réécriture de graphes sont basés sur des règles dites également de réécriture de graphes. Ces règles sont semblables aux règles de production mais dans lesquelles les termes sont des graphes. Une règle de réécriture de graphe est de la forme *L* <= *K* => *R* où :

- L et R sont des graphes représentant respectivement la partie gauche (pré-condition) et la partie droite (post-condition) de la règle. *K* est un graphe appelé graphe d'interface représentant la partie commune entre *L* et *R*.
- Soit un graphe *G* qu'on appellera graphe hôte (graphe sur lequel on va appliquer la règle). L'exécution d'une règle de réécriture sur G se fera suivant les étapes suivantes :
    1. *identifier dans G un sous-graphe qui correspond à la pré-condition L de la règle.*
    2. *Supprimer dans G tout ce qui appartient à L et n'appartient pas à l'interface K.*
    3. *Ajouter à G tout ce qui appartient la post-condition R et qui n'appartient pas à K.*

En général, les systèmes de réécriture de graphes permettent la définition et la mise en œuvre des règles de façon visuelle. La figure 1 illustre la définition visuelle des règles de réécriture de graphes. Dans cette figure la règle introduite par sa pré-condition *L* et sa post-condition *R* agit sur un graphe représentant une partie d'une application à objets. Cette règle élimine toutes les méthodes associées à une classe et crée un nœud de type *Component* (dans le sens des langages de description des architectures logicielles) qui sera relié à cette classe par un arc de type *map*. Ce type de règles que nous appelons règles de mise en correspondance ou mapping servent principalement à générer une architecture à partir d'un code source tout en maintenant un lien entre des composants d'un code source et la description de l'architecture ainsi obtenue. Dans cette règle l'interface K est constituée du nœud de type *Class* ce nœud représente la partie invariante entre *L* et *R*. Ainsi, lors de l'application de la règle au graphe hôte, le nœud de type *Method* qui appartient à *L* sans appartenir à *K* est supprimé du graphe et par conséquent tous les arcs entrants ou sortants de ce nœud sont également détruits. Le nœud de type *Component* qui appartient à *R* sans appartenir à *K* est ajouté au graphe. Le chiffre 1 qui préfixe le nœud de type *Class* signifie que la règle s'appliquera sur ce nœud. Sans cette désignation explicite du nœud sur lequel la règle va s'appliquer, la règle sera appliquée à tous les nœuds de type *Class*.

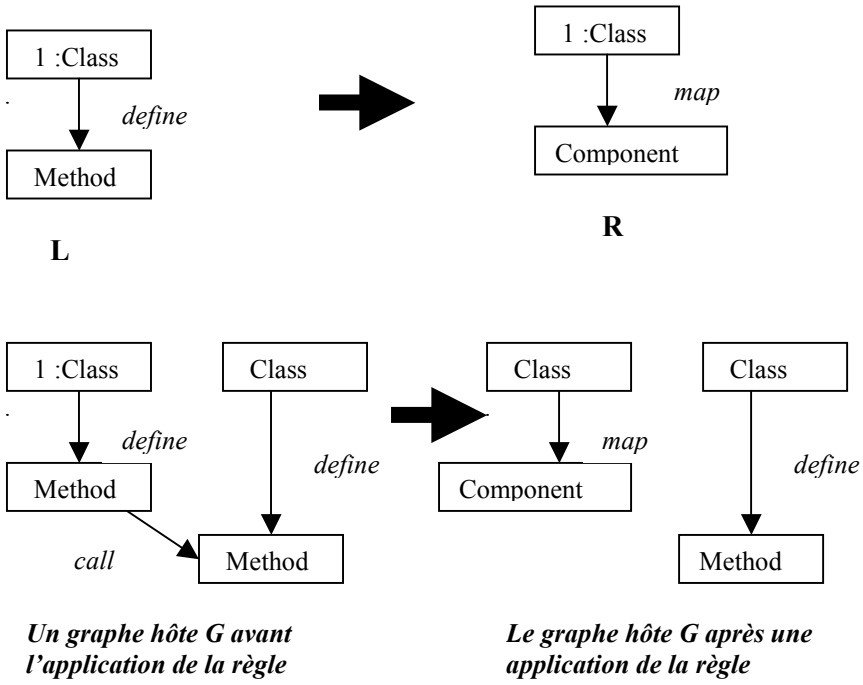

Figure 1. Exemple d'une règle de réécriture de graphes

2.2 Le système de réécriture de graphes AGG

Pour la mise en œuvre de notre approche nous avons utilisé le système AGG (Attribute Graph Grammar System) (Ermel et al, 1999). AGG est un langage visuel implémentant une approche algébrique d'expression des règles de réécriture ou de transformation de graphes. Dans AGG, les attributs associés aux nœuds et aux arcs peuvent être des expressions quelconques du langage Java. Ces expressions sont également des arguments du système de mise en correspondance entre les nœuds définis dans une règle et ceux d'un graphe hôte. Ainsi, une règle peut être paramétrée par un attribut d'un nœud (comme le nom d'une classe, etc.) et ne s'appliquera que sur ce nœud. AGG introduit également un troisième élément pour une règle appelé la NAC ou pré-condition négative. La pré-condition négative est un graphe défini de la même manière que la pré-condition et la post-condition d'une règle. Le rôle de la pré-condition négative est d'empêcher l'exécution d'une règle si un morphisme ou une correspondance existe entre un sous graphe du graphe hôte et cette pré-condition négative. Cela sert notamment pour éviter une exécution infinie d'une règle. La figure 2 illustre l'utilisation des NAC pour éviter une exécution infinie d'une règle de réécriture de graphes. En effet, cette règle associe à chaque classe un composant. Or la seule pré-condition est l'existence d'un nœud de type *Class* dans le graphe hôte et comme ce nœud reste toujours existant après l'exécution de la règle, cette dernière peut alors s'exécuter de façon infinie. La pré-condition négative ou NAC associée à cette règle empêche une réexécution de la règle pour toute classe qui est déjà reliée à un nœud de type *Component*.

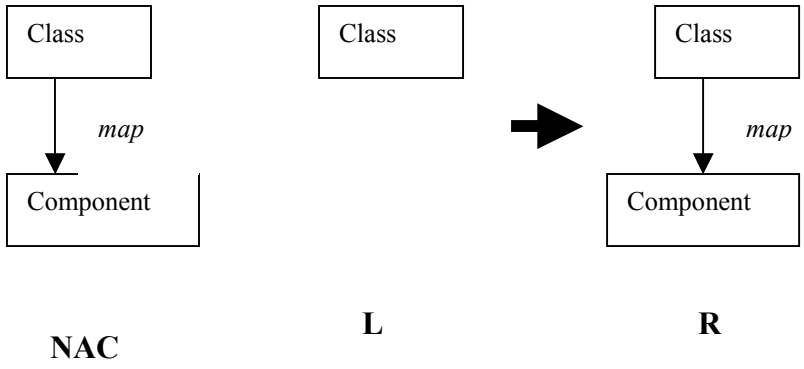

Figure 2. Exemple d'une règle de réécriture avec une pré-condition négative (NAC)

2. 3 Le standard GXL

GXL (Graph eXchange Language) a été défini et adopté par la communauté scientifique (surtout la communauté du génie logiciel) comme un format d'échange

de données représentées sous la forme de graphes. En effet, la majorité des outils intervenant dans le développement et la maintenance du logiciel manipulent des artefacts qui sont souvent représentés par des graphes tels que les graphes de flots de données, les diagrammes de conception ou de déploiement, etc. GXL peut être vu comme un des nombreux dialectes XML (eXtensible Markup Language) dédié à la représentation et l'échange de graphes. GXL permet de représenter des graphes attribués et typés. Il permet également la représentation des hyper-graphes et des graphes hiérarchisés. Ce format est utilisé pour faciliter l'interopérabilité des outils du génie logiciel. Ainsi, un analyseur de code source peut extraire des arbres syntaxiques qui seront transmis sous forme de documents GXL à un utilitaire de visualisation de graphes ce qui permettra ainsi d'implémenter un browser de programmes en faisant coopérer deux utilitaires existants. La figure 3 schématise le document GXL associé à une représentation en termes de graphes d'un schéma d'une base de données relationnelle. On remarquera l'existence de deux sortes de balises principales : des balises *node* pour représenter les nœuds et des balises *edge* pour représenter des arcs. Dans cette figure, on constatera la présence de deux nœuds. Un nœud de type *Table* identifié par *table1* et un nœud de type *Attribut* identifié par *att1*. Ces deux nœuds sont reliés par un arc de type *hasattribut* allant de *table1* (*from*) vers *attr1* (*to*) et identifié par *hasatt1*.

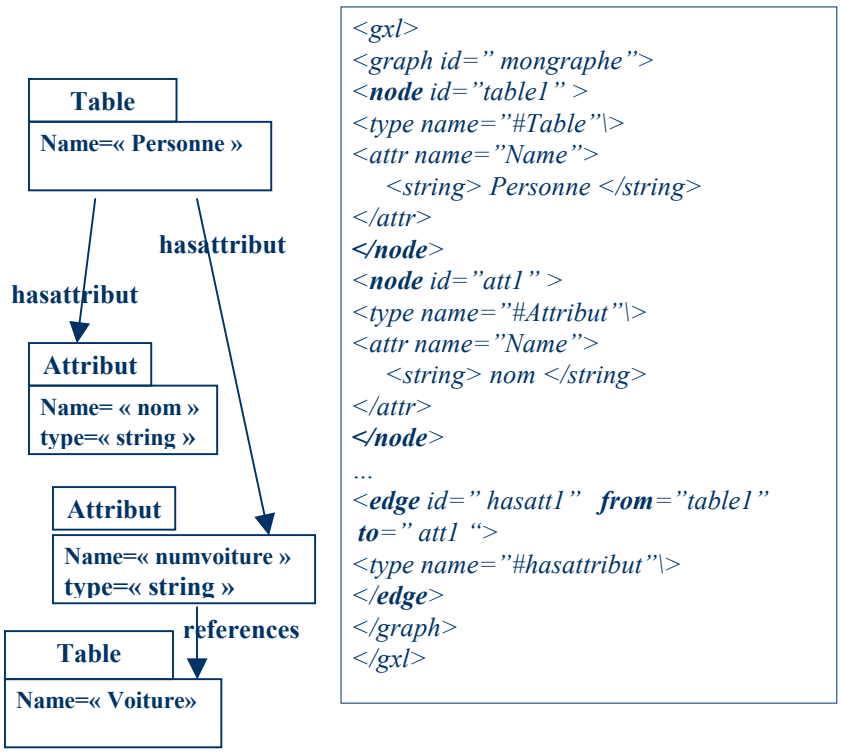

Figure 3. Exemple de graphe GXL

3. Structure générale du processus proposé

Pour implémenter notre processus nous avons développé un atelier expérimental de recouvrement d'architectures et de gestion de l'impact du changement du logiciel Cet atelier a été développé pour valider expérimentalement l'utilisation de GXL et des systèmes de réécriture de graphes dans le cadre d'une implémentation flexible des processus relatifs à la gestion de l'évolution du logiciel. L'architecture générale du processus et de l'atelier le supportant peut alors être schématisée par la figure 4.

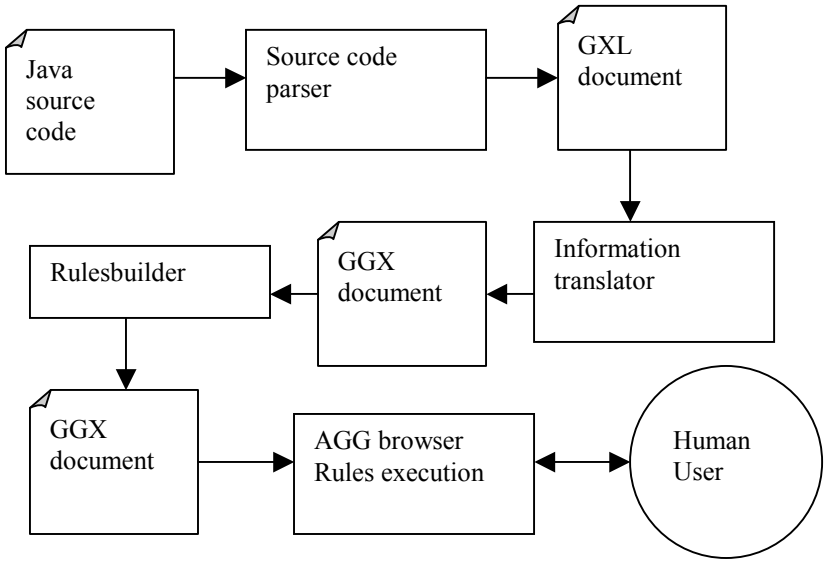

Figure 4. Structure générale du processus proposé

Les composants du processus actuellement implémenté dans l'atelier sont les suivants:

***L'analyseur du code source (source code parser)*:** il a pour fonction majeure de transformer les codes sources en des documents GXL. Nous avons mis en œuvre un analyseur pour des applications Java J2EE. L'analyseur commence d'abord par transformer le code source des applications en documents JavaML (Badros, 2000). JavaML est une représentation XML des codes sources Java. Les documents JavaML sont alors traités par un programme XSLT (W3C, 2001) pour produire un document GXL qui est une représentation des arbres syntaxiques ou AST (Kontogiannis, 2000) (Abstract Syntactic Trees) de programmes Java en GXL.

Le traducteur GXL => GGX (information translator) : les documents GXL produits par l'analyseur sont traduits en un autre format de documents XML appelé GGX. GGX est le format XML utilisé par le système AGG pour représenter les graphes ainsi que les règles de réécriture de graphes. Le traducteur ainsi défini est

implémenté par un script XSLT spécifique. Nous avons également réalisé un traducteur de GXL vers XMI pour assurer une interopérabilité entre notre approche et les modèles issus des travaux de l'OMG (MOF).

Le constructeur de règles (rule builder): Cet outil produit des ensembles de règles de transformation de graphes utilisées dans le cadre de la gestion de l'évolution des applications Java. L'utilisateur peut choisir de produire un ensemble de règles prédéfinies comme celles qui produisent des abstractions de haut niveau des codes sources (comme les architectures), ou des règles prédéfinies destinées à l'analyse de l'impact du changement de ces applications (Basson, 1998), etc. L'utilisateur peut plus tard raffiner les règles pour les adapter à des situations spécifiques.

Le browser de graphes et l'exécution des règles (AGG browser / rules execution) : c'est le système AGG (Ermel et al, 1999) qui permet quelques fonctionnalités d'affichage et de manipulation de graphes ainsi que l'exécution des règles d'une façon simple et visuelle. L'utilisateur peut se servir du browser pour mettre en application de nouvelles règles ou raffiner les règles prédéfinies.

Pour bien montrer l'utilisation de notre système et son développement, nous montrerons deux fonctionnalités que nous mettons en application et qui sont : le recouvrement des architectures logicielles et la propagation d'impact du changement des codes sources et/ou des constituants de l'architecture.

4. L'outil de recouvrement des architectures

Cet outil a pour but la production des descriptions abstraites de haut niveau des applications Java J2EE en analysant leurs codes sources. Les abstractions sont exprimées au moyen d'un ADL (Architecture Description Language) (Medvidovic, 2000) appelé ACME (Garlan et al, 2000). En fait, ACME n'est pas seulement un ADL mais il a été défini pour jouer le rôle de standard du fait qu'il inclut la majorité des concepts les plus utilisés ou définis dans les autres ADLs. Nous décrivons d'abord les entrées et les sorties du processus de recouvrement des architectures et qui sont : les applications Java J2EE et les architectures ACME (Garlan et al, 2000).

4.1 Structure des applications Java J2EE

La plate-forme Java J2EE fournit des composants dédiés au développement et au déploiement d'applications réparties selon une architecture client/serveur multi-tiers. Les applications J2EE sont basées sur un environnement permettant la définition et le déploiement d'objets métier appelés Enterprise Java Beans ou EJB. Cet environnement fournit un cadre neutre (indépendant des fournisseurs) pour la gestion de diverses tâches importantes comme la persistance, les transactions et la sécurité. Un profil type d'application Java J2EE peut être schématisé selon la figure

5. Dans cette figure, le client peut être une application Java ou un client Web comme les pages JSP. Le client demande des services mis en œuvre par des objets spéciaux de Java appelés *Beans*.
Un environnement appelé Conteneur gère le cycle de vie des *Beans*. Ce conteneur met en oeuvre quelques tâches système internes et transparentes concernant le déplacement, l'activation et de la passivation des *Beans*. Il fournit également quelques services traitant la persistance, les transactions et l'invocation des *Beans*. Un *Bean* met en application deux genres d'interfaces : les interfaces *Home* et *Remote*. L'interface *Home* concerne les méthodes qui créent ou suppriment le *Bean* et l'interface *Remote* fournit des méthodes dites métiers (qui implémentent les fonctionnalités que le *Bean* doit réaliser). Ainsi, le client demande d'abord l'exécution des méthodes *Home* pour créer le *Bean* et dialogue ensuite avec le *Bean* au moyen de méthodes *Remote* définies dans l'interface *Remote*. Quelques *Beans* appelés les *Beans* entités matérialisent sous la forme d'objets Java des données persistantes. En effet, chacun de ces *Beans* représente une ligne d'une table relationnelle. Nous avons expérimenter notre approche sur une application qui se compose de deux interfaces EJB (*Home* et *Remote*), d'une classe EJB appelée *ConverterBean* et d'une classe cliente appelée *ConverterClient*. L'ejb *ConverterBean* implémente quelques méthodes pour la conversion monétaire (devises) et la classe client utilise ces méthodes qui sont définies dans l'interface *Remote* de l'ejb. La classe cliente commence d'abord par demander la création d'une instance de l'ejb à l'interface *Home* et invoque par la suite les méthodes définies dans l'interface *Remote*.

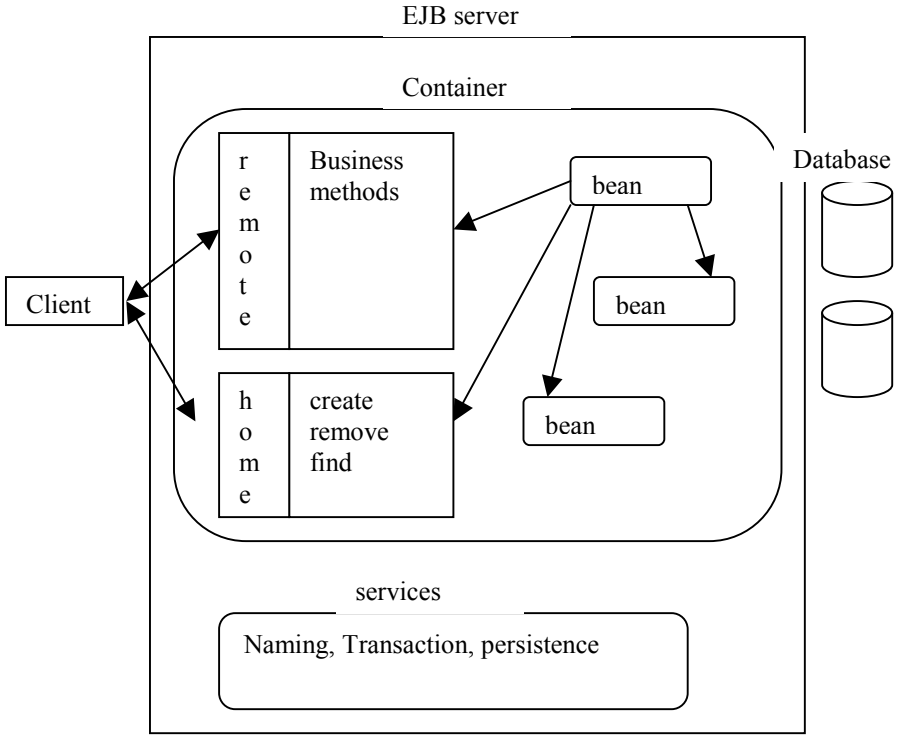

Figure 5. Architecture d'une application Java J2EE

4.2 ACME un ADL standard

ACME est un langage de description d'architectures combinant les fonctionnalités et les constructions d'une grande variété d'ADLs. Il fournit une structure générique et extensible pour représenter, produire et analyser des descriptions d'architectures. Les constructions principales proposées par ACME sont les composants, les connecteurs, les systèmes et les représentations. Les composants représentent des entités de calcul (programmes) décrites par des propriétés et des ports. Les ports identifient un point d'interaction entre un composant et son environnement. Un connecteur indique les interactions entre les composants. Il est décrit par des rôles indiquant le comportement des composants participant à l'interaction. Un système représente une configuration de l'architecture.
Le système peut être considéré comme un graphe de composants et de connecteurs. Les composants peuvent être représentés d'une façon hiérarchique. En fait, une représentation est une décomposition de composants de haut niveau d'abstraction en systèmes constitués de composants et de connecteurs de bas niveau. La relation verticale entre un composant de haut niveau d'abstraction et ceux appartenant à sa représentation s'appelle *Representation-Map*. Comme exemple, la figure 6 schématise une représentation graphique d'une architecture ACME d'une application Java J2EE, elle est basée sur la formalisation des applications J2EE présentée par Garlan et Sousa (Sousa, *et al*, 1999).

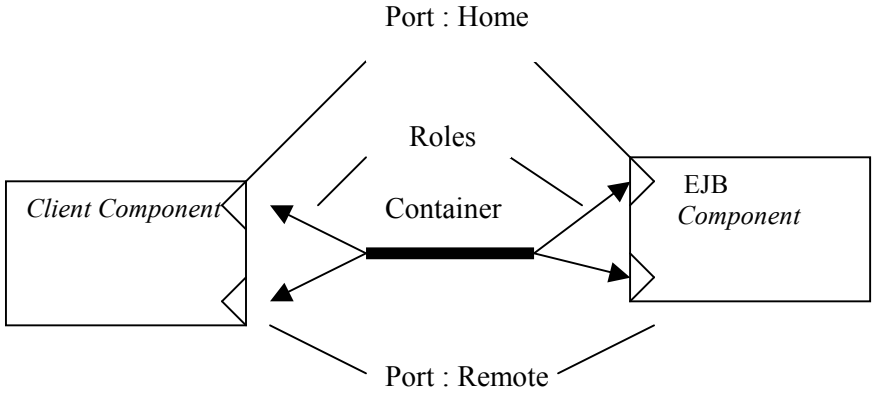

Figure 6. Une représentation ACME des EJB

Les composants représentent le client et l'EJB est représenté par un connecteur. Cette représentation n'est pas la seule manière de formaliser des applications J2EE au moyen de constructions ADL. Notre but est de pouvoir extraire la description architecturale d'une application d'une manière flexible. En d'autres termes, définir des règles qui correspondent à différents types de formalisation. Ainsi, il est possible

dans certains cas de représenter un EJB par un connecteur alors que dans d'autres cas il serait préférable de le représenter par un composant pouvant à son tour être décomposé en système, etc.

4.3 Le processus de recouvrement d'architectures

Le processus de recouvrement d'architectures est mis en oeuvre par deux groupes de règles de réécriture de graphes. Le premier groupe, appelé *règles d'extraction d'abstraction de haut niveau*, est utilisé pour transformer l'AST d'une application Java J2EE pour produire un graphe plus abstrait contenant les nœuds et les arcs qui sont significatifs au niveau architectural. Le deuxième groupe appelée *mappings d'architecture* définit l'ensemble des règles qui transforment les graphes obtenus par l'application des règles du premier groupe pour produire des descriptions d'architectures construites par des concepts de ACME.

4.3.1 Les règles d'extraction d'abstraction de haut niveau

Ces règles peuvent être divisées en deux sous-ensembles: les règles d'extraction de relations et les règles de nettoyage.

- Les règles d'extraction de relations ont pour but d'extraire des relations de haut niveau d'abstraction entre des artefacts présents comme des nœuds dans l'arbre syntaxique. Ces relations peuvent être des relations d'appel entre méthodes, l'héritage ou la composition entre classes, etc. En fait un AST ne préconise pas d'arc explicite représentant l'appel d'une méthode vers une autre. Ces relations sont représentées dans un AST par des chemins et cela est dû au fait que l'AST représente tous les détails d'un programme. Ainsi, pour la relation d'appel, par exemple, ces règles consistent à transformer des chemins reliant des nœuds à grain fin (comme les instructions, les variables, etc.) en des arcs reliant deux nœuds plus abstraits (comme des méthodes). Par exemple, si une méthode m1 contient un bloc d'instructions et une de ces instructions est un message (invocation d'une méthode m2), les règles d'extraction de relations créent un arc entre m1 et m2.
- Les règles de nettoyage du graphe suppriment les nœuds et les arcs à grain fin pour rendre sa structure plus compréhensible. En général les nœuds et les arcs supprimés correspondent à des détails et ne sont pas significatifs pour le processus de recouvrement d'architectures. L'application des règles d'extraction d'abstractions de haut niveau produit le graphe schématisé par la figure 7 qui est plus compréhensible que la visualisation de l'AST correspondant.

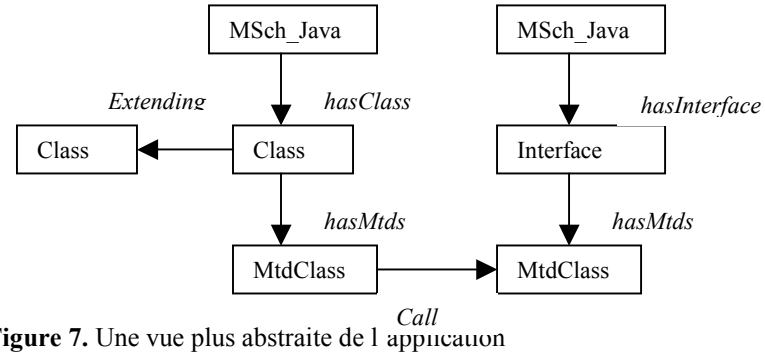

Figure 7. Une vue plus abstraite de l application

4.3.2. Les règles mappings d'architecture

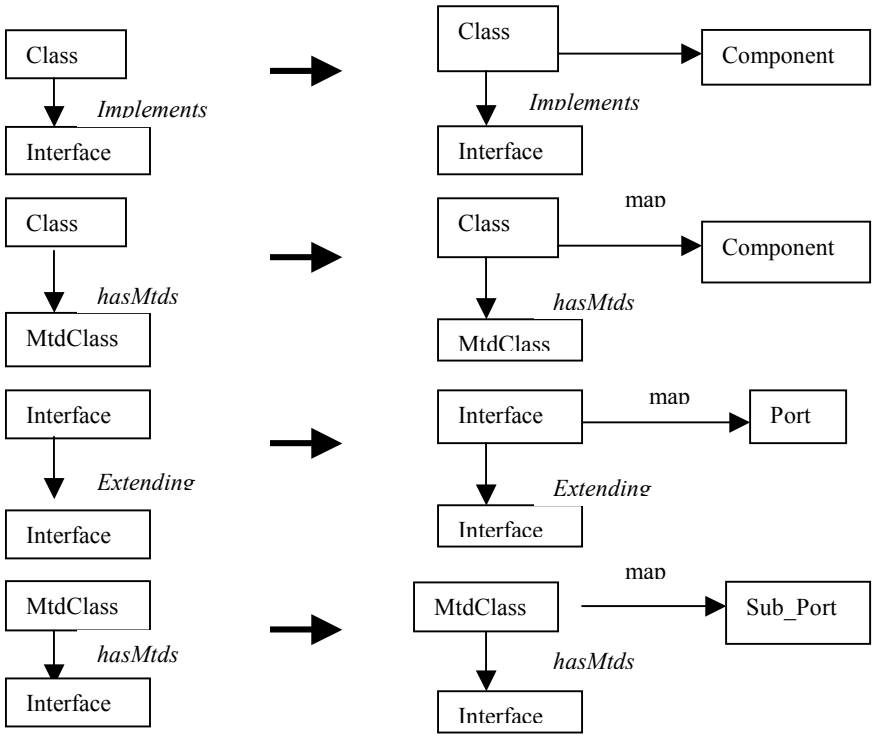

Figure 8. Quelques règles de mappings d'architecture

Les règles de mappings d'architecture visent à produire des modèles architecturaux à partir d'abstractions de haut niveau des codes sources. Ces règles mettent en

œuvre des mappings ou des correspondances entre les artefacts des codes source et le vocabulaire des langages de description d'architectures comme ACME . Dans notre approche nous essayons d'avoir une sorte de flexibilité dans la mise en œuvre de ce type de correspondances. Ainsi, les architectes peuvent choisir différentes stratégies pour implémenter les mappings et ainsi extraire des architectures. Chaque stratégie est mise en oeuvre par un ensemble de règles de réécriture de graphes. Nous montrons dans ce qui suit, la mise en œuvre de la stratégie présentée par Garlan et Sousa (Sousa *et al*, 1999). Cette stratégie peut être récapitulée comme suit:

- les interfaces Remote et Home sont représentées par des ports
- les méthodes sont représentées par des subports
- les classes et les beans sont représentés par des composants.

La figure 8 montre les règles de réécriture implémentant cette stratégie. Ces règles sont décrites chacune par sa pré-condition et sa post-condition. On remarquera la simplicité de définition de ce type de règles.

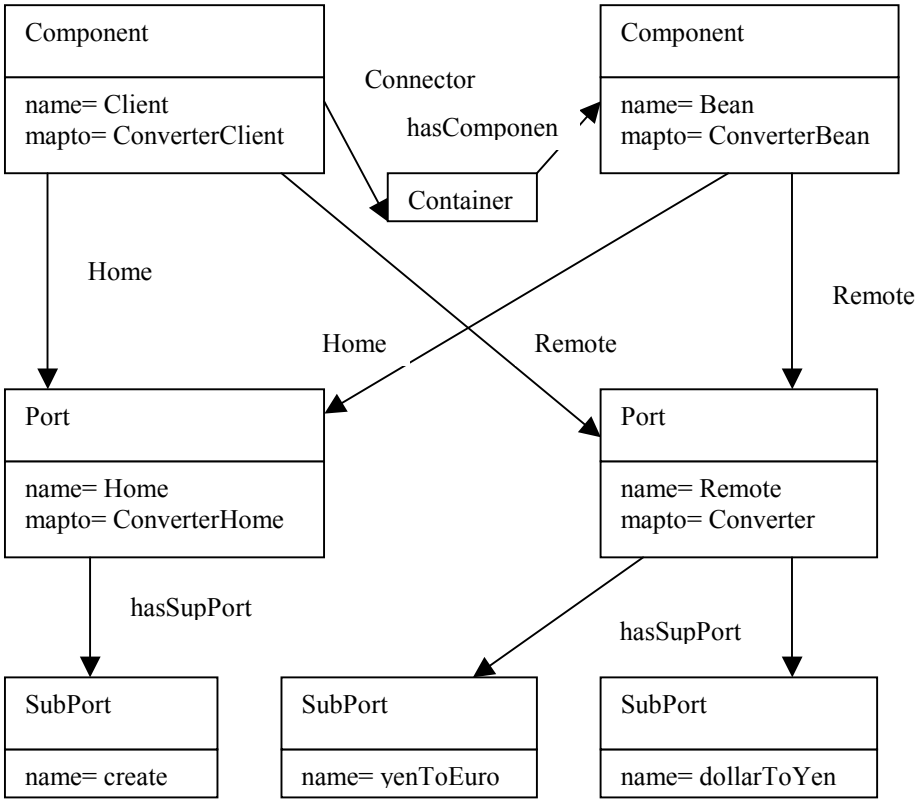

Figure 9. Architecture de l'application *Convertisseur de Monnaie*

La figure 9 représente l'architecture correspondante à l'application *Convertisseur de monnaie* après les applications successives des règles d'extraction des abstractions de haut niveau et des règles de mappings d'architecture.

5. Le processus de propagation d'impact du changement

La propagation d'impact du changement a été également développée au moyen de règles de réécriture de graphes. Ces règles implémentent la propagation de l'impact du changement d'un artefact du code source ou de l'architecture sur le reste de l'application. Dans un premier temps, nous avons établi une taxinomie des changements en considérant les opérations de base appliquées aux nœuds et aux arcs et qui sont l'insertion, la suppression et la modification. Ainsi, pour chaque type de nœuds et d'arcs nous avons considéré les trois opérations de base et nous avons implémenté trois techniques pour la propagation de l'impact. Ces techniques sont : la propagation totale, la propagation partielle et le change-and-fix.

- La technique de propagation totale consiste à propager l'impact du changement d'un nœud ou d'un arc à tous ses voisins potentiellement affectés par ce changement et réitérer ainsi la propagation pour chacun de ces voisins. Cette technique est utile pour avoir une idée générale sur l'impact du changement d'un artefact donné.
- La technique de propagation partielle consiste à propager l'impact en ne considérant qu'un nombre restreint de types d'arcs par lesquels l'impact peut être propagé. Cette technique est utile notamment quand on veut étudier l'impact d'un changement sur une hiérarchie d'héritage ou un graphe d'appel, etc.
- La technique dite du change-and-fix a été introduite par Rajlch (Rajlich, 1997). Elle consiste à opérer le changement sur un nœud ou un arc et à propager l'impact aux voisins immédiats. Ces voisins sont traités un à un et le chargé de l'évolution décidera pour chaque voisin de corriger les effets de l'impact et de s'arrêter ou de continuer à propager l'impact aux autres voisins et ainsi de suite.

En plus de l'implémentation de ces trois techniques de propagation, nous avons défini trois ensembles de règles de réécriture de graphes implémentant la propagation de l'impact du changement. Ces règles concernent :

- La propagation de l'impact du changement d'un artefact du code source sur le reste du code (impact horizontal entre constituants du code source).
- La propagation de l'impact du changement d'un artefact de l'architecture sur le reste de l'architecture (impact horizontal entre constituants de l'architecture)
- La propagation de l'impact du changement d'un artefact du code source sur l'architecture et *vice versa* (impact vertical).

6. Conclusion

Nous avons présenté une approche et des outils destinés à assister la gestion de l'évolution et de la maintenance des applications multi-tiers. Ce travail a eu comme résultats la définition et l'implémentation d'un modèle de représentation des artefacts logiciels basé sur une utilisation combinée des notions de graphes et d'hyperdocuments XML. Ces artefacts peuvent être des constituants d'un code source ou d'une description d'architecture. Les outils que nous avons mis en œuvre concernent le recouvrement d'architectures logicielles et la propagation de l'impact du changement. Ce travail a pour but de montrer que l'adoption combinée des concepts issus de la théorie des graphes, des systèmes de réécriture de graphes et de GXL permet à la fois de faciliter l'interopérabilité des outils dédiés au génie logiciel et rend plus aisée la mise en œuvre de mécanismes nécessitant des possibilités de raisonnement et cela d'une façon intégrée et uniforme et sans divergences sémantiques (*impedence mismatch*) entre les concepts utilisés. En effet, on reste toujours dans le cadre de la théorie des graphes. Nous avons également implémenté des passerelles entre GXL et le modèle XMI qui est utilisé comme moyen de méta modélisation dans le cadre du modèle MOF. Ainsi, nous réalisons une sorte d'interopérabilité entre les techniques de méta modélisation par le MOF et GXL en utilisant comme media l'espace technologique XML. Nous sommes actuellement en phase d'extension de nos travaux pour la prise en compte d'attributs non fonctionnels tels que les mesures qualitatives et cela dans le but d'étendre les capacités de raisonnement sur les effets de l'évolution pour prendre en compte les effets du changement sur la qualité du logiciel.

7. Bibliographie

Badros G.. JavaML: a markup language for Java source code. Computer Networks (Amsterdam, Netherlands: 1999), 33(1-6):159-177, 2000.

Basson H. An integrated model for impact analysis of software change. In International Conference on Software Quality Manage- ment, Amsterdam, Netherlands, Springer Verlag, pages 260-296, April 1998.

Ehrig H., Engels G., Kreowski H.-J., and Rozenberg G.Hand-book of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Lan-guages, andTools. World Scientific,1999.

Ermel G., Rudolf M., and Taentzer G.. The agg approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation, volume 2. World Scientific,1999.

eXtensible Markup Language (xml) 1.0 (second edition), w3c recommandation. Technical Report http://www.w3c.org/TR/2000/REC xml 20001006, World Wide Web Consortium, 2000.

Garlan D., Monroe R.T., and Wile D. Acme: Architectural description of component based systems. In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems, pages 47-68. Cambridge University Press, 2000.

Holt R., Winter A., Schürr A and Sim S. Gxl: Towards a standard exchange format. In 7th Working Conference on Reverse Engineering, pages 23-25, Brisbane, Queensland, Australia, November 2000.

Kontogiannis Kostas. Towards portable source code representations using xml. In Proceedings of WCRE'00, pages 172-182, Brisbane Australia, November 2000.

Manolescu I., Florescu D., Kossmann D., Xhumari F., and Olteanu D. Agora: Living with xml and relational. In VLDB, 2000.

Medvidovic N. and Taylor R.N. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1):70-93, January 2000.

Rajlich V. Propagation of change in object-oriented programs. In ESEC/FSE'97 Workshop on Object-Oriented Reengineering, Zurich, September 1997.

Sousa J.P. and Garlan D. Formal modeling of the enterprise javabeans component integration framework. In World Congress on Formal Methods, pages 1281-1300, 1999.

Sun Microsystems. J2ee platform specification. http://java.sun.com/j2ee/, 2002.

W3C. Xsl transformations (xslt). http://www.w3.org/TR/xslt, 2001