Extraction métaheuristique d'une architecture à base de composants à partir d'un système orienté objet

Sylvain Chardigny*, Abdelhak Seriai*
Mourad Oussalah**, and Dalila Tamzalit**

* Ecole des Mines de Douai, 941 rue Charles Bourseul, 59508 Douai France
** LINA, université de Nantes, 2 rue de la Houssinière, 44322 Nantes France

Résumé. La modélisation et la représentation des architectures logicielles sont devenues une des phases principales du processus de développement de systèmes complexes. En effet, la représentation de l'architecture fournit de nombreux avantages pendant tout le cycle de vie du logiciel. Cependant pour beaucoup de systèmes existants, aucune représentation fiable de leurs architectures n'est disponible. Afin de pallier cette absence, source de nombreuses difficultés, nous proposons, dans cet article une approche, appelée ROMANTIC, visant à extraire une architecture à base de composants à partir d'un système orienté objet existant. L'idée première de cette approche est de proposer un processus quasi automatique d'identification d'architecture en formulant le problème comme un problème d'optimisation et en le résolvant au moyen de métaheuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système.

1 Introduction

La modélisation et la représentation de l'architecture logicielle des systèmes complexes sont devenues une phase importante dans leur processus de développement (Bertolino et al. (2005)). L'architecture d'un système décrit sa structure à un haut niveau d'abstraction, en terme de composants et de connecteurs. Cette abstraction offre de nombreux avantages tout au long du cycle de vie du logiciel (Shaw et Garlan (1996)). En effet, disposer d'une représentation de l'architecture facilite les échanges entre les concepteurs et les programmeurs. Ensuite, pendant les phases de maintenance et d'évolution, cette représentation permet de localiser les défauts du logiciel et réduit les risques lors de l'ajout d'une nouvelle fonctionnalité. De plus, la distinction entre les composants et les connecteurs rend explicite la séparation entre les aspects métiers et communications et facilite la compréhension et l'évolution du système. Enfin, l'architecture à base de composants favorise la réutilisation de certaines parties du système représentées par les composants.

Cependant, force est de constater, que beaucoup de systèmes existants ne disposent pas d'une représentation fidèle de leur architecture. En effet, le système peut avoir été conçu sans représentation de son architecture, comme dans le cas de certains systèmes patrimoniaux. Pour d'autres systèmes, la représentation disponible peut être décalée par rapport à l'implémentation

du système. Ce décalage apparaît avec les écarts entre l'architecture prévue et implémentée puis s'accroît avec le manque de synchronisation entre la documentation et l'implémentation.

Partant de ces constats, nous proposons une approche appelée ROMANTIC[1] visant à extraire une architecture à base de composants à partir d'un système orienté objet. Notre processus doit sélectionner parmi l'ensemble des architectures pouvant être abstraites du système, la meilleure possible par rapport à un ensemble de propriétés de qualité et en respectant un ensemble de guides telle que la documentation du système. Ainsi, nous proposons de formuler ces propriétés comme des contraintes mesurables et nous utilisons une approche d'optimisation métaheuristique afin d'extraire l'architecture qui satisfait le maximum de ces contraintes. Notre choix d'une approche d'optimisation métaheuristique est motivé par les travaux récents dans le domaine du *Search-based software engineering* (SBSE) qui ont montré l'efficacité de ces techniques pour résoudre des problèmes de ce type (Harman (2007)). Nous modélisons donc le problème d'extraction comme un problème d'optimisation en définissant les différents éléments nécessaires à la résolution de ce type de problème. Ainsi, l'un des avantages majeurs de notre approche est son degré élevé d'automatisation qui diminue le besoin en expertise humaine qui est souvent chère et pas toujours disponible.

Dans la suite de cet article, nous présentons cette approche d'extraction d'architecture comme suit. La section 2 introduit notre modélisation du problème d'extraction comme un problème d'optimisation et présente l'étude des guides pour l'exploration de l'espace de recherche. Dans la section 3, nous étudions les caractéristiques des composants architecturaux afin de définir notre fonction objectif à optimiser. Notre algorithme d'optimisation et l'ensemble des opérateurs de manipulation sont présentés dans la section 4. Un cas d'étude est présenté dans la section 5. Enfin, les travaux apparentés et une conclusion sont présentés, respectivement, dans les sections 6 et 7.

2 L'extraction d'architecture comme un problème d'optimisation

Afin de modéliser le problème d'extraction comme un problème d'optimisation, nous devons définir l'espace de recherche, *i.e.* une représentation des solutions du problème, et les guides permettant de diriger l'algorithme de recherche.

2.1 Définition de l'espace de recherche

L'extraction d'architecture d'un système orienté objet consiste à utiliser son code source pour obtenir une abstraction de ce système en terme d'éléments architecturaux : les composants qui décrivent la partie métier, les connecteurs qui décrivent les interactions et la configuration qui représente la topologie des connexions entre les composants. Une première étape du problème d'extraction d'architecture consiste donc à définir l'ensemble des architectures logicielles pouvant abstraire le système. Ces architectures constituent l'espace de recherche de notre algorithme d'optimisation. La définition de cette espace de recherche est basée sur la définition d'un modèle de correspondance entre les concepts architecturaux (*i.e.* composants,

[1] ROMANTIC : Re-engineering of Object-oriented sytesMs by Architecture extractioN and migraTIon to Component based ones.

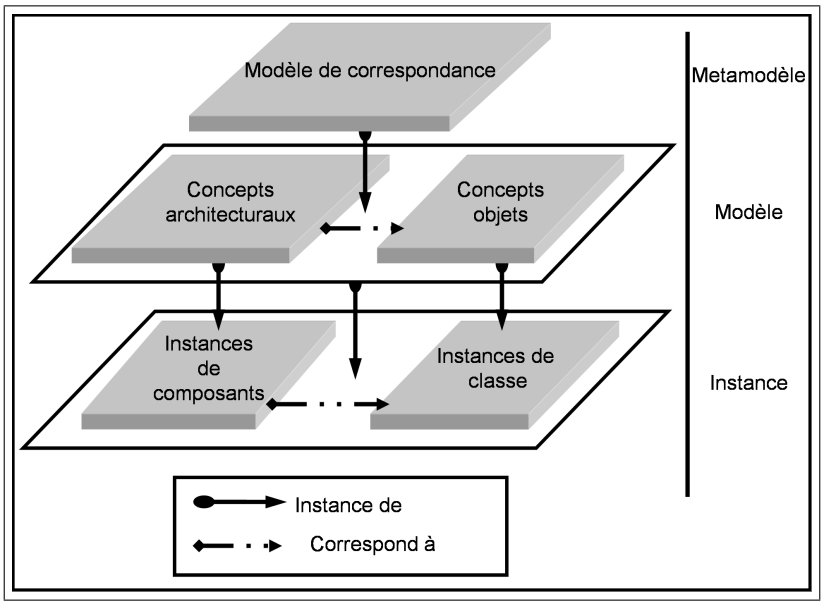

FIG. 1 – *modèle de correspondance et architecture à base de composants*

connecteurs, interfaces, etc.) et les concepts objets (*i.e.* classes, interfaces, paquetages, etc.). L'espace de recherche est alors l'ensemble des instances de ce modèle(*cf.* Fig.1).

Selon ce modèle de correspondance, nous définissons une architecture comme une partition des classes du système. Chaque élément de cette partition représente un composant. Nous appelons ces éléments des contours. Ils contiennent des classes qui peuvent appartenir à différents paquetages (*cf.* Fig.2). Nous assimilons l'ensemble des interfaces du composant à «l'interface du contour». Chaque contour est constitué de deux ensembles de classes : «l'interface du contour» qui contient l'ensemble des classes qui ont un lien avec des classes situées à l'extérieur du contour, par exemple un appel de méthode vers l'extérieur ; et le «centre» qui contient le reste des classes du contour.

Comme le montre la Fig.3, nous assimilons l'ensemble des interfaces du composant à l'interface du contour et le composant au contour. Nous nous concentrons ici sur l'extraction de composants. Pour cela nous considérons que les connecteurs sont tous les liens existants entre deux composants.

Au final, l'espace de recherche de notre problème d'optimisation est composé de toutes les partitions des classes du système. Cela signifie, que pour un système contenant n classes, on aura un espace de recherche contenant $\frac{(2*n)!}{(n+1)!n!}$ architectures possibles.

2.2 Guides du processus d'extraction d'architecture

En plus de la définition de l'espace de recherche, nous avons besoin de définir des guides pour diriger le processus d'exploration de l'espace. Ainsi, nous avons identifié plusieurs élé-

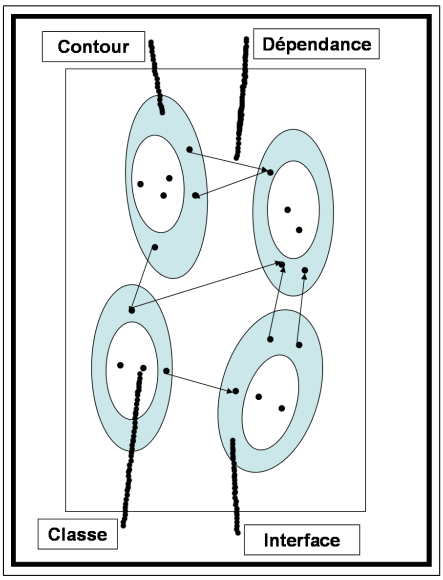

FIG. 2 – *Éléments de contours*

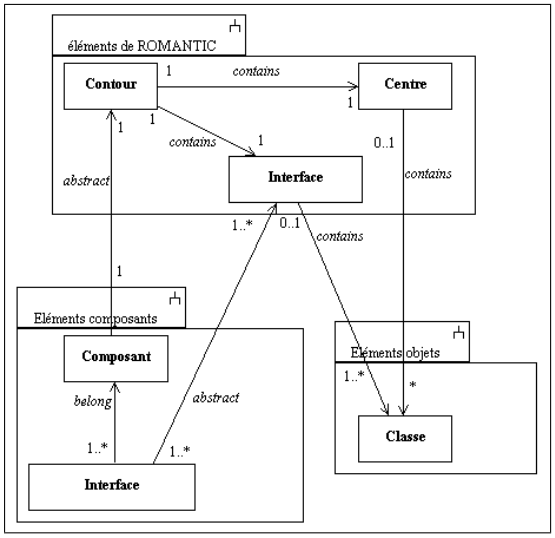

FIG. 3 – *Notre modèle de correspondance objet-composant*

ments (*cf.* Fig.4) qui peuvent être séparés en deux catégories selon la façon dont ils influencent ce processus d'exploration.

D'abord, les propriétés sémantiques et de qualité nous permettent de définir la fonction objectif. Ainsi, nous proposons de réifier les caractéristiques communément admises d'une architecture et ses propriétés de qualité pour définir une fonction objectif mesurant d'une part la validité sémantique et d'autre part la qualité de cette architecture.

Ensuite, d'autres éléments peuvent être utilisés pour guider notre processus sans agir sur la définition de la fonction objectif. Ainsi, les documents de modélisation disponibles, tels que les diagrammes UML, et les recommandations de l'architecte peuvent être utilisés pour diriger notre processus. Les documents de modélisation peuvent être utilisés pour ajuster les résultats obtenus suivant les fonctionnalités prévues du système. De la même façon les recommandations peuvent, par exemple, être exploitées pour rejeter une solution explorée qui possède trop de composants. Enfin, les contraintes de déploiement (ressource, configuration, *etc.*) peuvent affecter la modélisation de l'architecture du système. Ainsi, nous utilisons les propriétés de l'architecture matérielle pour guider notre processus d'extraction afin d'obtenir une meilleure adéquation entre les architectures matérielles et logicielles. Par exemple, l'adaptation d'une architecture extraite à une utilisation embarquée impose des limitations sur la taille des composants ou le nombre de connecteurs.

Nous pensons que l'utilisation de l'ensemble de ces guides permettra à notre processus de nous fournir une représentation pertinente d'une architecture à base de composants. Cependant, parmi ces guides, celui qui utilise la sémantique de l'architecture et des éléments architecturaux est prépondérant. Pour cette raison nous présentons dans ce papier notre définition de la fonction objectif en utilisant la sémantique associée au concept d'architecture. Nous présentons ensuite une métaheuristique utilisant cette fonction pour parcourir l'espace de recherche

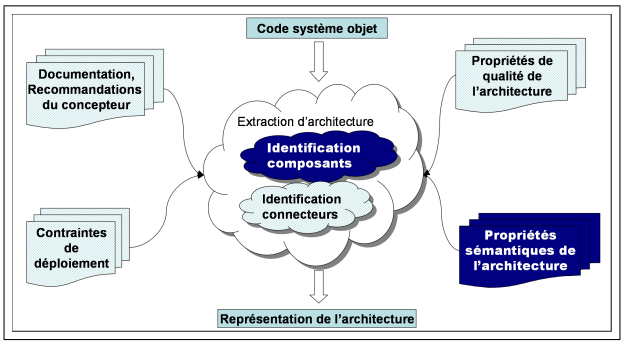

FIG. 4 – *Guides de l'extraction*

afin d'obtenir la meilleure architecture possible.

3 Validité sémantique d'une architecture

La définition de la fonction objectif nécessite une définition précise de ce qu'est une architecture sémantiquement valide. Nous considérons qu'une architecture est sémantiquement valide si ses éléments (composants, connecteurs et configuration) sont sémantiquement valides. Dans cet article nous nous limitons à l'étude de la sémantique des composants. Nous procédons à la définition de la fonction en deux étapes. Premièrement, nous étudions les caractéristiques sémantiques des composants et nous définissons un ensemble de fonctions pour les mesurer. Cette étude est basée sur les définitions les plus couramment admises des composants logiciels plutôt qu'architecturaux. En effet, les propriétés des composants architecturaux sont incluses dans celles des composants logiciels. Ces contraintes supplémentaires nous permettent de mieux envisager la migration d'un système orienté objet vers un système à base de composants. Finalement, nous définissons une combinaison de ces fonctions d'évaluation des caractéristiques sémantiques pour définir notre fonction objectif mesurant la validité sémantique.

3.1 Caractérisation de la sémantique des composants logiciels

Il existe de nombreuses définitions qui caractérisent un composant. Néanmoins, il existe d'importantes similitudes parmi les définitions les plus courantes. Ainsi, Szyperski définit un composant comme une unité de composition possédant des interfaces spécifiées par contrat et des dépendances explicites avec le contexte. Il peut être déployé indépendamment et peut être composé par un tiers (Szyperski (1998)). Heinemann et Councill définissent un composant comme un élément logiciel qui est conforme à un modèle de composant et peut être déployé indépendamment et composé sans modification selon un standard de composition (Heinemann et Councill (2001)). Enfin Luer définit un composant comme un élément logiciel qui (a) encapsule une implémentation réutilisable d'une fonctionnalité, (b) peut être composé sans modification et (c) adhère à un modèle de composant. Il distingue cette définition de celle d'un

composant déployable qui est un composant (a) pre-paquetagé, (b) distribué indépendamment, (c) facile à installer et désinstaller et (d) auto-descriptif (Luer et van der Hoek (2002)).

En combinant et en raffinant les éléments communs de ces définitions ainsi que ceux d'autres définitions généralement admises (Jacobson et al. (1997)), nous proposons la définition suivante du composant :

> *Un composant est un élément logiciel qui (a) est composable sans modification, (b) peut être distribué de manière autonome, (c) encapsule une fonctionnalité, et (d) qui adhère à un modèle de composant.*

Nous utilisons pour définir le modèle d'un composant, la définition donnée dans (Luer et van der Hoek (2002)) : un modèle de composant est la combinaison de (a) un standard de composant qui gouverne la construction de chaque composant et (b) un standard de composition qui régit comment organiser un ensemble de composants en une application et comment ces composants communiquent et interagissent entre eux. Ainsi, on retrouve dans cette définition le critère de Heinemann sur l'adhésion à un standard de composition ainsi que les propriétés de Luer sur l'auto-description, le pré-paquetage ou encore la facilité d'installation/désinstallation qui sont régis par le standard de composant.

En conclusion, d'après notre définition d'un composant, nous avons identifié trois caractéristiques sémantiques des composants : la **composabilité**, l'**autonomie** et la **spécificité**. La spécificité impose qu'un composant ait un nombre limité de fonctionnalités.

3.2 Évaluation de la validité sémantique d'un composant logiciel

Nous avons identifié dans la section précédente trois caractéristiques sémantiques, que nous proposons d'évaluer. Pour cela nous utilisons le modèle proposé par la norme ISO-9126 (ISO/IEC-9126-1 (2001)) (*cf.* Fig.5.A) pour la mesure de caractéristiques d'un produit. Selon ce modèle, nous pouvons mesurer la caractéristique *validité sémantique* en la raffinant en trois sous-caractéristiques que sont les trois caractéristiques sémantiques identifiées dans la section précédente. Nous définissons un ensemble de propriétés mesurables des composantspour chaque sous-caractéristique. Cependant on ne peut pas mesurer ces propriétés directement sur les contours. Nous utilisons donc notre modèle de correspondance (*cf.* Fig.5.B) et établissons les liens entre ces propriétés et celle des contours. Enfin nous proposons des métriques pour mesurer les propriétés des contours.

3.2.1 Sous-caractéristiques de la sémantique *vs.* propriétés des composants

Nous avons raffiné les sous-caractéristiques sémantiques en un ensemble de propriétés mesurables des composants. D'abord, un composant est autonome si il ne possède pas d'interface requise. Son autonomie décroît lorsque le nombre d'interfaces requises augmente. Par conséquent, la propriété **nombre d'interfaces requises** permet de mesurer la sous-caractéristique autonomie. Ensuite, un composant est composé à travers ses interfaces fournies et requises. Cependant, un composant sera plus facile à utiliser avec d'autres si les services, dans chaque interface, sont cohésifs. Ainsi la propriété **moyenne de la cohésion des services par interface** doit être une mesure correcte de la sous-caractéristique composabilité. Enfin, l'évaluation du nombre de fonctionnalités est basée sur les affirmations suivantes. Premièrement, un composant qui propose plusieurs interfaces, doit fournir plusieurs fonctionnalités. Ainsi plus grand est

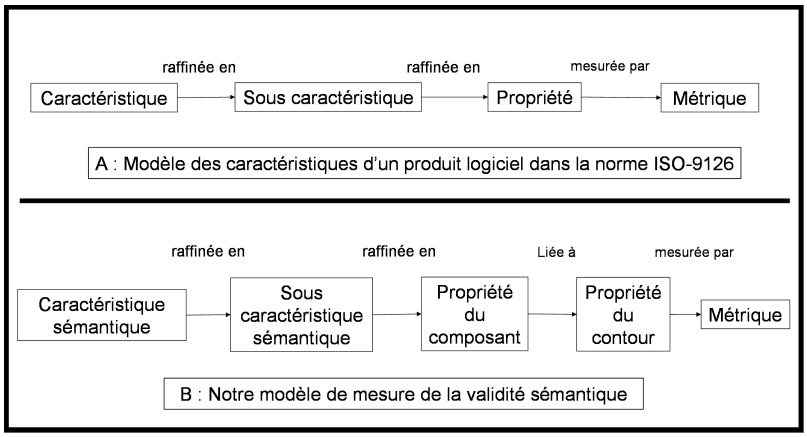

FIG. 5 – *Modèle de mesure de caractéristiques dans la norme ISO-9126 et dans notre approche*

le nombre d'interfaces, plus grand est le nombre de fonctionnalités. Deuxièmement, si les interfaces (*resp.* les services dans chaque interfaces) sont cohésives (*i.e.* partagent des ressources), elles offrent probablement des fonctionnalités proches. Troisièmement, si le code du composant est très couplé (*resp.* cohésif), les différentes parties du code du composant s'entraident (*resp.* utilisent des ressources communes). Par conséquent, elles travaillent probablement de concert pour fournir un petit nombre de fonctionnalités. De ces affirmations nous raffinons la sous-caractéristique spécificité par les propriétés suivantes : **nombre d'interfaces fournies**, **moyenne de la cohésion des services par interface**, **cohésion des interfaces** et **cohésion et couplage internes du composant**.

3.2.2 Propriétés du composant *vs.* propriétés du contour

Les propriétés des composants ne peuvent pas être mesurées sur les contours. Selon notre modèle d'évaluation (*cf.* Fig.5.B), nous relions donc les propriétés des composants aux propriétés mesurables des contours.

Premièrement, d'après notre modèle de correspondance (*cf.* Fig.3), l'ensemble des interfaces du composant est relié à l'interface du contour. La **cohésion moyenne des classes de l'interface** donne une bonne mesure de la moyenne de la cohésion des services par interface.

Deuxièmement, la cohésion des interfaces, la cohésion interne du composant et le couplage interne du composant peuvent être respectivement mesurés par les propriétés du contour **cohésion de l'interface**, **cohésion du contour** et **couplage interne du contour**.

Troisièmement, pour relier le nombre d'interfaces fournies à une propriété des contours, nous associons une interface fournie du composant à chaque classe de l'interface du contour qui possède une méthode publique. Grâce à cette hypothèse, nous pouvons mesurer le nombre d'interfaces fournies en utilisant la propriété **nombre de classes de l'interface possédant une méthode publique**.

Enfin, le nombre d'interfaces requises peut être évalué par le couplage entre le composant et l'extérieur. Ce couplage est relié au couplage externe du contour. Ainsi, nous mesurons cette propriété par la propriété **couplage externe du contour**.

3.3 Définition des métriques pour l'évaluation de la sémantique

Nous avons établi dans les sections précédentes, les liens entre les sous-caractéristiques sémantiques et les propriétés des contours (*cf.* Fig.6). Nous définissons maintenant les métriques nous permettant de les mesurer.

Les propriétés **couplage interne du contour** et **couplage externe du contour** nécessitent une mesure de couplage. Nous définissons $Couplage(E)$ la métrique mesurant le couplage interne d'un contour E et $CouplageExt(E)$ celle mesurant le couplage externe de E. Ces métriques mesurent trois types de dépendances entre les objets : les appels de méthodes, l'utilisation d'attributs et de paramètre d'une autre classe. Ces métriques sont des pourcentages et sont reliées par l'équation : $couplageExt(E) = 100 - couplage(E)$. Due à la limitation de l'espace, nous ne présentons pas les détails de ces métriques.

Les propriétés des contours **cohésion du contour** et **cohésion moyenne des classes de l'interface** nécessitent une mesure de cohésion. La première demande une mesure de la cohésion d'un ensemble de classes alors que la dernière demande la mesure de la cohésion d'une classe. Dans les deux cas, la mesure peut être obtenue en calculant la cohésion de l'ensemble des méthodes et attributs respectivement de l'ensemble des classes ou de la classe. La métrique «Loose Class Cohesion» (LCC), proposée par Bieman et Kang (Bieman et Kang (1995)), mesure le pourcentage de paires de méthodes qui sont directement ou indirectement connectées. Deux méthodes sont connectées si elles utilisent directement ou indirectement un attribut commun. Deux méthodes sont indirectement connectées si il existe une chaîne de méthodes connectées qui les relie. Nous utilisons cette métrique pour calculer la cohésion dans notre processus.

3.4 Évaluation de la validité sémantique des composants

En exploitant les liens, établis précédemment, nous définissons une fonction d'évaluation pour chacune des sous-caractéristiques. Nous utilisons ensuite ces fonctions pour définir la fonction d'évaluation de la validité sémantique des composants qui constitue notre fonction objectif.

Nous définissons les fonction Spe, A, C respectivement pour la spécificité, l'autonomie et la composabilité, où $nbPub(I)$ est le nombre de classes de l'interface possédant une méthode publique et $|I|$ est le cardinal de l'interface du composant :

$$Spe(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Couplage(E) - nbPub(I))$$

$$A(E) = couplageExt(E) = 100 - Couplage(E)$$

$$C(E) = \frac{1}{|I|} \cdot \sum_{i \in I} LCC(i)$$

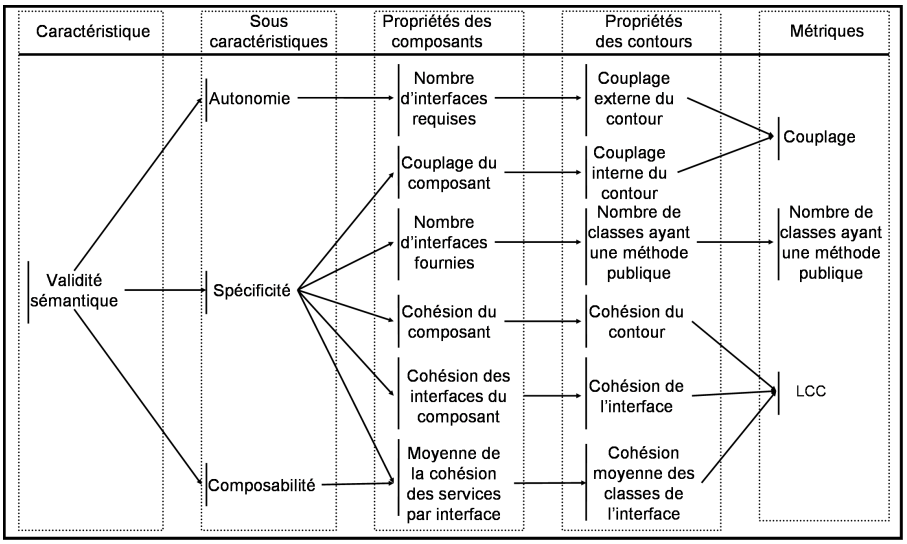

FIG. 6 – *Notre modèle de mesure de la validité sémantique des composants*

L'évaluation de la caractéristique **validité sémantique** est basée sur l'évaluation de chaque sous-caractéristique. Ainsi, nous définissons cette fonction comme une combinaison linéaire de chaque fonction d'évaluation (Spe, A, et C) :

$$S(E) = \frac{1}{\sum_i \lambda_i}(\lambda_1 \cdot C(E) + \lambda_2 \cdot A(E) + \lambda_3 \cdot Spe(E)) \quad (1)$$

Cette forme linéaire nous permet de considérer uniformément les sous-caractéristiques. Le poids associé à chaque fonction permet à l'architecte de modifier, au besoin, l'importance relative des sous-caractéristiques.

4 Algorithme d'identification des composants

4.1 Principes du métaheuristique

Nous avons choisi d'utiliser une métaheuristique pour résoudre le problème d'extraction. Cet algorithme est appelé recuit simulé («*Low-Temperature Simulated Annealing*») (Laarhoven et Aarts (1987)). Il consiste essentiellement en une série de tentatives de modification sur une solution d'un problème d'optimisation. Les changements qui accroissent la qualité de la solution sont acceptés et la solution modifiée devient le point de départ d'une nouvelle série de tentatives de modification. De plus, certains changements qui réduisent la qualité de la solution sont acceptés afin de permettre à l'exploration de s'échapper des minima locaux. De tels changements sont acceptés avec une probabilité qui décroît progressivement durant le processus d'exploration selon l'*équation* 2, où p est la probabilité d'accepter la solution donnée, δq est la variation de qualité par rapport à la solution courante, et T est la valeur de la température.

Chaque début de nouvelles séries de tentatives de modification provoque une décroissance de la température. L'algorithme s'arrête lorsque $T = 0$.

$$p = e^{-\frac{\Delta q}{T}} \tag{2}$$

Comme la plupart des métaheuristiques, cet algorithme nécessite une fonction objectif, une représentation du problème et un moyen d'altérer les solutions. En plus, il nécessite un schéma de refroidissement (*cooling schedule*) qui détermine à quelle vitesse le paramètre de température décroît et ainsi la vitesse d'exécution et la qualité de la solution obtenue. Enfin, le recuit nécessite une solution initiale qui est le point de départ de l'optimisation. Afin d'utiliser cet algorithme, nous utilisons notre fonction objectif (*cf. équation* 1) pour évaluer la qualité de la solution. Nous présentons dans la suite les choix faits pour les autres paramètres du recuit.

4.2 Schéma de refroidissement et opérateurs de manipulation

Le schéma de refroidissement inclut plusieurs paramètres tels que T_{start} la température de départ, M le nombre de tentatives de modification à chaque étape et f la fonction de refroidissement. Nous utilisons un schéma géométrique, c'est à dire que la température est réduite d'un facteur constant après chaque étape du processus. Ainsi, f est ce facteur et tend vers 1 en fonction du temps disponible. Concernant la température de départ, nous calculons T_{start} selon la qualité de la solution de départ pour obtenir une probabilité de 0.8 qui est la probabilité couramment admise dans les utilisations de cet algorithme.

Concernant le nombre de tentatives de modification et la nature de ces modifications, nous définissons trois opérateurs de manipulation :

- $move(c, s_1, s_2)$: déplace une classe c du contour s_1 vers le contour s_2.
- $extract(c, s)$: crée un nouveau contour contenant la classe c, et supprime c de s.
- $fusion(s_1, s_2)$: regroupe les contours s_1 et s_2 en un nouveau contour contenant toutes les classes de s_1 et de s_2.

Les deux premiers opérateurs sont élémentaires et nous permettent d'obtenir par combinaison toutes les opérations possibles. Néanmoins, nous avons ajouté le troisième opérateur pour augmenter la probabilité de réduire le nombre de contours. En effet, l'utilisation des opérateurs est aléatoire et le choix des seuls deux opérateurs élémentaires conduit notre processus à une partition contenant beaucoup de petits contours.

Enfin, l'algorithme trouvera l'optimal si M tend vers l'infini et f vers 1. Cependant, la valeur de M et de f déterminent le temps d'exécution. Nous choisissons, donc de fixer $M = 1$ et de modifier f en fonction du temps disponible.

4.3 Point de départ de l'exploration

Le dernier paramètre du recuit à choisir est son point de départ, *i.e.* la solution initiale à partir de laquelle le processus explore l'espace de recherche. On peut choisir aléatoirement ce point de départ mais nous préférons utiliser une solution qui est probablement proche de la solution optimale.

Nous calculons les composantes fortement connexes d'un graphe représentant le système orienté objet et les utilisons comme point de départ. Les sommets de ce graphe représentent les classes du système et l'existence d'un arc entre deux sommets a et b signifie que la classe

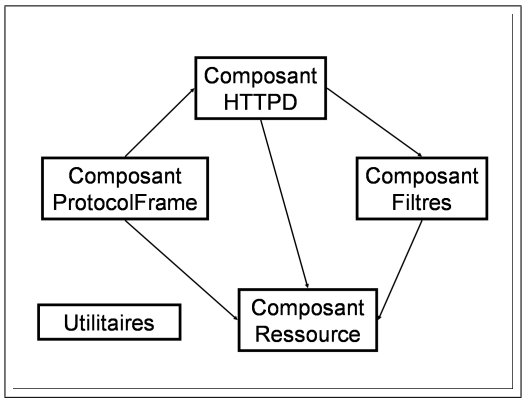

FIG. 7 – *Architecture de Jigsaw*

a utilise la classe b, *i.e.* utilise un attribut, un paramètre ou une variable de retour dont le type est b et crée ou utilise un objet dont le type peut être b. Une composante fortement connexe est un sous-ensemble de sommets dans lequel toute paire de sommets est reliée par un chemin. Ces composantes fortement connexes forment une partition des sommets du graphe et par conséquent une partition des classes du système.

Tous les éléments de cette partition sont un ensemble de classes qui partagent un fort couplage et sont plus couplés entre eux qu'avec les autres classes du système. Ainsi par comparaison avec les travaux de Coulange (1998) proposant de commencer l'identification des composants réutilisables par la localisation des sous-ensembles connexes parmi les éléments du système, la solution obtenue a de bonnes chances d'être proche de la solution optimale.

5 Étude de cas : Jigsaw

Nous avons testé notre processus d'extraction sur Jigsaw qui est un serveur Web java. Ce logiciel a été utilisé pour présenter des processus d'extraction d'architecture dans des travaux existants (Medvidovic et Jakobac (2006)). Son architecture est présenté sur la figure 7

Jigsaw contient autour de 300 classes. Le calcul des composantes fortement connexes nous donne une première partition de 100 contours. Cette partition a une valeur de 35.7% pour notre fonction objectif. Ainsi, la température initiale, qui est calculée pour obtenir une probabilité d'acceptation de 80%, est de 19°. Nous lançons le processus plusieurs fois avec des valeurs différentes pour f entre 0.99 et 0.9999. Les résultats sont assez similaires et ont pratiquement le même résultat pout notre fonction objectif : 87%.

La figure 8 présente l'architecture extraite par ROMANTIC (à gauche) ainsi que sa superposition avec l'architecture prévue (à droite) et les correspondances entre les composants par des flèches allant de gauche à droite. Notre architecture contient 16 composants de plus que celle prévue. Cependant, la comparaison des deux architectures montre que la plupart de nos composants sont des sous-composants des composants prévus. Quelques uns sont inclus en même temps dans plusieurs composants prévus.

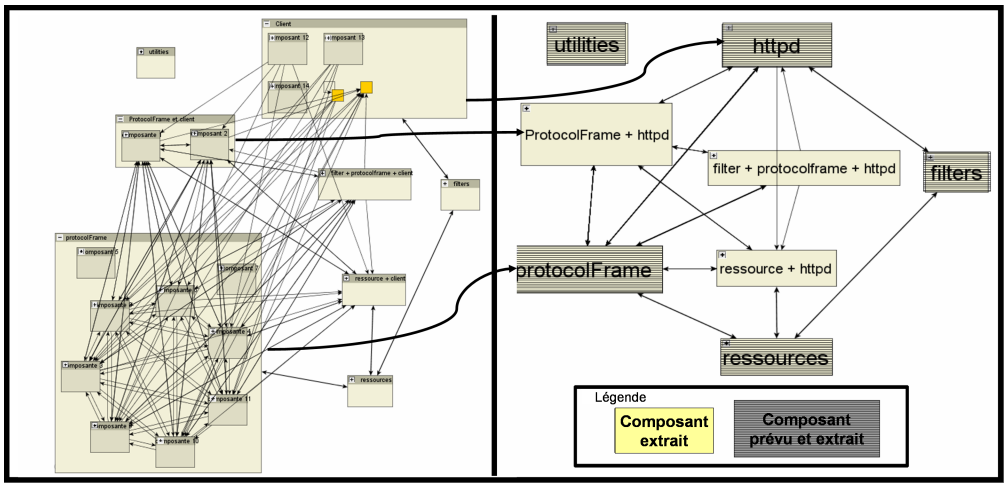

FIG. 8 – *Architecture de Jigsaw en utilisant ROMANTIC*

L'étude de ces résultats montre que les composants extraits par ROMANTIC et inclus dans les composants prévus définissent une fonctionnalité qui est une sous-fonctionnalité des composants prévus. Par exemple, le composant *protocolFrame* inclut les composants de ROMANTIC qui gèrent les cadres sur une page Web.

Pour évaluer l'efficacité de ROMANTIC, nous avons procédé à plusieurs tests en utilisant un algorithme aléatoire. Cet algorithme utilise le même schéma de refroidissement que notre processus mais à chaque étape il se déplace aléatoirement vers une nouvelle solution et conserve la meilleure solution rencontrée. Sur les 100 tests réalisés sur Jigsaw, le résultat moyen est de 64%. Ce résultat n'est pas très loin du score obtenu par ROMANTIC, mais la principale différence est la granularité des résultats. En effet les résultats obtenus aléatoirement ont en moyenne 100 composants de taille moyenne de 3 classes. De plus l'identification des fonctionnalités de ces composants est plus délicate que ceux obtenus avec ROMANTIC.

En conclusion, notre processus nous offre une architecture de Jigsaw proche de celle attendue. De plus le résultat est bon selon notre fonction objectif (87%) et certains composants encapsulent une fonctionnalité identifiée. Enfin les resultats obtenus peuvent être améliorés en regroupant certains des composants extraits et en en fractionnant d'autres. Cet écart avec l'architecture prévue peut être réduit en augmentant le facteur de décroissance f au prix d'un temps d'execution allongé.

6 Travaux apparentés

De nombreux travaux sont présentés dans la littérature pour réaliser l'extraction d'une architecture d'un système orienté objet (Pollet et al. (2007)). Nous distinguons ces travaux selon trois critères : le processus, les entrées et la technique. Du point de vue du processus, l'extraction peut suivre une approche montante (*bottom-up*), descente (*top-down*) ou hybride. Les processus montant débutent de connaissances de bas niveau pour récupérer l'architecture. A

partir des modèles du code source, ils augmentent progressivement le niveau d'abstraction jusqu'à atteindre le haut niveau de compréhension souhaité. Dali Kazman et al. (2001) est un exemple d'approche montante. Au contraire les processus descendant travaillent à partir des connaissances de haut niveau telles que les spécifications ou les styles architecturaux et visent à découvrir l'architecture en formulant des hypothèses conceptuelles et en les mettant en correspondance avec le code. Le modèle **Reflexion** de Murphy *et al.* Murphy et al. (1995) est un exemple typique d'un tel processus. Enfin, les processus hybrides combinent les approches montantes et descendantes. D'un côté, ils abstraient les connaissance de bas niveau en utilisant différentes techniques. De l'autre, les connaissances de haut niveau sont raffinées et confrontées à la vue extraite précédemment. **Focus** Medvidovic et Jakobac (2006) et **ManSART** Harris et al. (1995) sont des exemples de processus hybrides. ROMANTIC est une approche dans son état actuel un processus montant. Cependant, l'utilisation des autres guides, et en particulier la documentation et les recommandations de l'architecte doivent en faire un processus hybride.

Les entrées des processus d'extraction sont diverses. La plupart travaillent à partir de représentations du code, mais utilisent aussi d'autres informations, pour la plupart non-architecturales. Nous pouvons citer, par exemple, l'expertise humaine qui est utilisée de manière interactive pour diriger le processus (Medvidovic et Jakobac (2006)) et l'organisation physique, tel que les fichiers et les répertoires (Harris et al. (1995)). D'autres utilisent des informations architecturales. Medvidovic et Jakobac (2006) utilisent ainsi les styles pour obtenir une architecture idéalisée. Au final, la plupart des travaux sont basés sur l'expertise humaine : certains utilisent l'expertise de l'architecte qui utilise l'approche alors que d'autres utilisent l'expertise de ceux qui ont proposé cette approche. Dans ROMANTIC, nous utilisons la sémantique architecturale afin de réduire le besoin d'expertise qui peut être coûteux ou indisponible. Cette entrée sera complétée par les autres guides identifiés et en particulier la documentation qui est utilisée dans les travaux existants à travers l'expertise humaine.

Les techniques utilisées pour extraire l'architecture sont variées et peuvent être classées selon leur degré d'automatisation. Premièrement, certaines approches sont quasi manuelles. Par exemple, Focus (Medvidovic et Jakobac (2006)) est un guide pour un processus hybride qui regroupe les classes et met en correspondance les entités extraites avec une architecture idéalisée à partir du style architectural en fonction de l'expertise humaine. Deuxièmement, la plupart des approches proposent des techniques semi-automatiques. Elles automatisent les aspects répétitifs de l'extraction mais l'expert dirige le processus de raffinement ou d'abstraction menant à l'identification des éléments architecturaux. Ainsi ManSART (Harris et al. (1995)) tente de mettre en correspondance les éléments du code source avec le style architectural et les patterns définis par l'expert. Enfin certaines techniques sont quasi-automatiques. On peut par exemple citer les algorithmes de regroupement qui sont utilisés pour produire des regroupements cohésifs et faiblement interconnectés (Anquetil et al. (1999)). ROMANTIC est également quasi-automatique et elle est la seule approche à utiliser une métaheuristique d'exploration. La principale différence avec les autres approches quasi-automatiques est que nous raffinons les définitions couramment admises des composants en un ensemble de caractéristiques sémantiques et modèles de mesure alors que les autres travaux utilisent l'expertise de leurs auteurs afin de définir les règles qui contrôlent le processus.

Notre approche par métaheuristique, nous rapproche aussi d'autres travaux dans le domaine du SBSE. Ce domaine peut être défini comme l'application de métaheuristiques pour résoudre les problèmes d'optimisation rencontrés dans l'ingénierie logiciel (Harman (2007)).

Ces techniques ont été appliquées à diverses activités de l'ingénierie logicielle, tout au long du cycle de vie, depuis la spécification des besoins (Bagnall et al. (2001)), jusqu'à la maintenance (O'Keeffe et Cinnéide (2006)). Cependant, ces techniques n'ont pas encore été appliquées à un niveau architectural comme dans notre approche. Cependant, les travaux de Mancoridis et al. (1999) sont très proches de ce niveau et de notre approche. Ils modélisent le problème de modularisation du logiciel comme un problème de regroupement où les métaheuristiques peuvent être appliquées. Leur outil Bunch propose ensuite plusieurs métaheuristiques et retourne un graphe de dépendances entre les modules qui n'est donc pas précisément une vue architecturale au contraire de notre approche.

7 Conclusion

Nous proposons, dans cet article, une approche métaheuristique d'extraction d'une architecture à base de composants depuis un système orienté objet. Nous avons présenté plusieurs guides qui peuvent diriger une exploration de l'espace des architectures possibles. L'approche détaillée ici, est basée sur le guide utilisant la validité sémantique de l'architecture. Par conséquent, nous avons défini une fonction objectif basée sur la sémantique des composants. Cette fonction est ensuite utilisée comme fonction objectif dans un algorithme de recuit simulé et dirige ainsi la recherche de l'optimum parmi les architectures pouvant être extraites.

La principale différence avec les travaux existants sur l'extraction d'architecture est que nous raffinons les définitions couramment admises des composants en un ensemble de caractéristiques sémantiques et de modèles de mesure alors que les autres travaux utilisent l'expertise de leurs auteurs pour définir les règles dirigeant le processus. Ainsi, l'avantages majeurs de notre approche est son degré élevé d'automatisation qui diminue le besoin en expertise humaine qui est souvent chère et pas toujours disponible.

Nous avons choisi de centrer l'étude présentée sur les composants. Cependant les connecteurs et la configuration sont également des éléments architecturaux. L'étude de leurs sémantiques constitue la première perspective de ce travail. Il nous faut étudier ces éléments et déterminer si leur extraction doit avoir lieu avant, pendant ou après celle des composants.

Cependant les caractéristiques sémantiques que nous utilisons ne sont pas les seuls guides identifiés. L'utilisation des autres guides est une autre perspective de ce travail. On a ainsi débuté l'étude des qualités de l'architecture et la définition d'une fonction objectif basée dessus. Pour cela nous avons sélectionner plusieurs caractéristiques de qualité, telle que la fiabilité ou la maintenabilité, et nous développons, de la même manière que pour la validité sémantique, un modèle de mesure pour ces caractéristiques. Nous étudions également la manière dont la documentation et les recommandation de l'architecte peuvent agir sur notre processus. L'idée principale dans l'utilisation de ce guide est d'utiliser l'expertise humaine disponible à travers les documents de conception existant et, si un architecte est disponible, ses recommandations. Nous souhaitons donc utiliser ces informations pour définir un point de départ de l'exploration qui soit plus pertinent que les composantes fortement connexes. Ces informations peuvent aussi nous permettent de réduire l'espace de recherche en interdisant certaines solutions.

Références

Anquetil, N., C. Fourrier, et T. C. Lethbridge (1999). Experiments with clustering as a software remodularization method. In *Proc. of the Sixth WCRE*, pp. 235. IEEE.

Bagnall, A. J., V. J. Rayward-Smith, et I. M. Whittley (2001). The next release problem. *Information & Software Technology 43*(14), 883–890.

Bertolino, A., A. Bucchiarone, S. Gnesi, et H. Muccini (2005). An architecture-centric approach for producing quality systems. In *QoSA/SOQUA*, pp. 21–37.

Bieman, J. M. et B.-K. Kang (1995). Cohesion and reuse in an object-oriented system. In *Proc. of the Symp. on Software reusability,SSR '95*, pp. 259–262.

Coulange, B. (1998). *Software Reuse*. Springer-Verlag.

Harman, M. (2007). The current state and future of search based software engineering. In *Future of Software Engineering*, pp. 342–357. IEEE.

Harris, D. R., H. B. Reubenstein, et A. S. Yeh (1995). Reverse engineering to the architectural level. In *Proc. of ICSE*, pp. 186–195. ACM, Inc.

Heinemann, G. et W. Councill (2001). *Component-based software engineering*. Addison-Wesley.

ISO/IEC-9126-1 (2001). In *Software engineering - Product quality - Part 1 : Quality Model*. ISO-IEC.

Jacobson, I., M. Griss, et P. Jonsson (1997). *Software Reuse*. Addison Wesley/ACM Press.

Kazman, R., L. O'Brien, et C. Verhoef (2001). Architecture reconstruction guidelines. Technical report.

Laarhoven, P. J. M. et E. H. L. Aarts (Eds.) (1987). *Simulated annealing : theory and applications*. Norwell, MA, USA : Kluwer Academic Publishers.

Luer, C. et A. van der Hoek (2002). Composition environments for deployable software components. Technical report.

Mancoridis, S., B. S. Mitchell, Y.-F. Chen, et E. R. Gansner (1999). Bunch : A clustering tool for the recovery and maintenance of software system structures. In *ICSM*, pp. 50–.

Medvidovic, N. et V. Jakobac (2006). Using software evolution to focus architectural recovery. *Automated Software Engineering 13*, 225–256.

Murphy, G. C., D. Notkin, et K. Sullivan (1995). Software reflexion models : bridging the gap between source and high-level models. In *SIGSOFT '95*, pp. 18–28.

O'Keeffe, M. et M. Ó. Cinnéide (2006). Search-based software maintenance. In *CSMR*, pp. 249–260.

Pollet, D., S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, et H. Verjus (2007). Towards a process-oriented software architecture reconstruction taxonomy. In *Proc. of the 11th CSMR*, pp. 137–148.

Shaw, M. et D. Garlan (1996). *Software architecture : perspectives on an emerging discipline*. USA : Prentice-Hall.

Szyperski, C. (1998). *Component Software*. ISBN : 0-201-17888-5. Addison-Wesley.

Extraction métaheuristique d'architecture

Summary

Software architecture modeling and representation are a main phase of the development process of complex systems. In fact, software architecture representation provides many advantages during all phases of software life cycle. Nevertheless, for many systems, like legacy or eroded ones, there is no available representation of their architectures. In order to benefit from this representation, we propose, in this paper, an approach called ROMANTIC which focuses on extracting a component-based architecture of an existing object-oriented system. This extraction is a balancing problem of competing constraints which aims at obtaining the best architecture that can be abstracted from a system. Consequently, the main idea of this approach is to propose a quasi-automatic process of architecture identification by formulating it as a search-based problem. The latter acts on the space composed of all possible architectures abstracting the object-oriented system.