République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Ferhat Abbas de Sétif - UFAS (Algérie)

Thèse

Présentée à la Faculté des Sciences

Département d'informatique

En vue de l'obtention du diplôme de

Doctorat

Option : Informatique

Par

Adel ALTI

Thème

Coexistence de la modélisation à base d'objets et de la modélisation à base de composants architecturaux pour la description de l'architecture logicielle

Soutenue le 22 Juin 2011 devant le jury composé de :

Président	Pr Mohamed Ahmed-NACER	Prof USTHB Alger
Examinateurs	Pr Mohamed Tayeb LASKRI	Prof Univ Annaba
	Pr Nacereddine ZAROUR	Prof Univ Constantine
	Dr Abdallah KHABABA	MCA UFAS Sétif
	Dr Allaoua REFOUFI	MCA UFAS Sétif
Rapporteur	Dr Abdallah BOUKERRAM	MCA UFAS Sétif

Remerciements

Je remercie chaleureusement Monsieur feu Tahar Khammaci. Maître de Conférences à Nantes, d'avoir accepté de diriger mes travaux de recherche. Tahar, j'ai eu la chance de travailler avec toi. Je n'oublierai jamais tes conseils et tes pertinentes remarques. Merci ! Et repose en paix. Que le Professeur Mourad Oussalah, trouve ici toute ma reconnaissance pour ses précieux conseils, et ses orientations, dans la construction de cette thèse.

Je remercie vivement Monsieur Abdallah Boukerram, MC à l'UFAS, pour tous ses conseils et ses orientations dans l'aboutissement de cette thèse. Qu'il soit vivement remercié.

Je remercie le Pr. M. Ahmed NACER de l'USTHB d'avoir d'accepté de présider ce jury, qu'il trouve toute ma gratitude.

Je remercie le Pr. M. Laskri de l'université de Annaba, pour l'honneur q'il me fait, en acceptant d'évaluer ce travail. Que le Pr. N. Zarour, trouve également toute ma gratitude en acceptant de faire partie de ce jury. Je tiens à présenter tous mes remerciements à Dr A. Reffoufi et au Dr A. Khababa, pour le temps consentis à l'évaluation de cette thèse.

À ma famille, mes frères *Samir* et *Fouad*, mes sœurs *Hayet*, *Nassima* et *Rima*, et notamment *ma mère* et *mon père* à qui je réserve bien évidement d'autres formes de remerciements !

Je remercie Mr A. Smeda, Doctorant à Nantes, je dis : Merci Adel, pour ton support moral et ton aide, qui ont compté beaucoup pour moi. Merci encore !

Merci aussi à Maiza. A, Reffad. H, L. Zitouni, M. Messai, A. Amirat, C. Khentout, L. Douidi, C. Mediani et K. Harbouche, D. Mahieddine sans oublier M. Nekkache pour leur soutien moral.

Enfin, que tous les collègues du département d'informatique de l'UFAS, avec qui j'ai passé de longues années de travail et d'échange en soient remerciés. Toute ma reconnaissance à la sympathique équipe MODAL.

Résumé

L'aspect clé de conception du système logiciel est son architecture. La description de l'architecture logicielle est fondée sur deux techniques de modélisation : la modélisation d'architecture logicielle à base de composants (composants académiques et composants industriels) et la modélisation orientée objet sous la houlette d'UML (Unified Modeling Language). Chacun de ces deux approches présente des points forts et des points faibles.

Notre contribution se décline en deux volets : le premier volet concerne la définition d'un profil UML 2.0 pour COSA (Component Object-based Software Architecture) ; est une approche hybride composant-objet de description d'architectures. On utilise toutes les capacités des profils UML (méta-modèle et modèle), pour définir une spécification complète pour l'architecture logicielle. Les profils UML existants sont dédiés à un type d'application (système distribués, temps réel, etc.), alors que notre profil est indépendant et spécialisé dans une activité particulière de modélisation des langages de description d'architectures. Le deuxième volet porte sur l'intégration du profil UML, COSA au sein de la démarche MDA (Model Driven Architecture). Nous proposons une stratégie de transformation directe à l'aide du profil UML pour l'élaboration des modèles d'implémentation objets ayant un niveau d'abstraction haut et un degré de réutilisation comme celui de COSA. Les outils COSABuilder et COSAInstantiator sont également développés et implémentés.

Mots clés : Architecture Logicielle, Langages de Description d'Architecture, COSA, Profil UML, MDA, Transformation.

Abstract

A key aspect of the design of any software system is its architecture. There are at least two different techniques of describing the architecture of a software system, either by using object-oriented modelling, in particular Unified Modelling Language (UML) or by using component-oriented modelling (academic components and industrial components). Each one of these two approaches has its advantages and also its drawbacks.

Our contribute is situated around two axes: the first axe concerns with building a UML 2.0 profile for COSA (Component Object-based Software Architecture) ; which is an approach for software architecture based on object oriented modelling and component based modelling. Using the capacities of UML profiles (meta-models and models), we define a complete specification for software architectures. Actually, UML profiles are particular kind of application (system distributed, real time, etc), while our profile is independent and specialized in a particular activity of modelling of the languages of description of architectures. The second axe is interested with integrating COSA UML profile within MDA (Model Driven Architecture). Also we define a strategy of direct transformation using UML profile in the generation of the object implementation models. These models have a higher level of abstraction and a degree of reuse of COSA. COSABuilder and COSAInstantiator Tools are also developed and implemented.

Keywords: Software Architecture, Architecture Description Language, COSA, UML profile, MDA, Mapping.

ملخص

أحد الجوانب الرئيسية لتصميم أنظمة البرمجيات هي المخطط. هناك ما لا يقل عن اثنين من التقنيات المختلفة التي تصف مخطط نظام البرامجيات، إما عن طريق استخدام التمثيل بالأشكال الموجهة، وبخاصة في UML، أو عن طريق استخدام التمثيل بالمكونات (المكونات الاكاديميه والمكونات الصنـاعيه). كل واحد من هذين النهجين له مزاياه وأيضا له عيوبه.

ومساهمتنا تقع حول محورين هما: الأول تم فيه بتقديم وصف COSA في UML 2.0 : التي هي ج لهندسة البرمجيات القائمة على استخدام التمثيل بالأشكال الموجهة والتمثيل بالمركبات. استخدام قدرات وصفات UML (ميتا - النماذج والنماذج) لنحدد مواصفات كاملة لهندسة البرمجيات. الوصفات الحالية هي مكرسة لنوع من التطبيقات(أنظمة الوقت الحقيقي الخ) بينما أعمالنا هي مستقلة ومتخصصة في نشاط معين من النماذج وصفا للهندسة البرامجية. تم في المحور الثاني على دمج وصف UML COSA ضمن MDA. ونقدم أيضا استراتيجية للتحول المباشر باستخدام هذا الوصف في الحصول على نماذج تنفيذية للأشكال الموجهة ذات مستوى أعلى من التجريد وتحقيق درجة استخدام تعادل COSA. وقد تم تطوير وتنفيذ البرامج COSABuilder و COSAInstantiator .

مفاتيح :الهندسة البرامجية، لغات الهندسة البرامجية، COSA، وصف UML، MDA، التحويل.

Table des matières

Liste des figures

Liste des tables

Introduction générale

Introduction

Cette thèse s'inscrit dans le domaine des architectures logicielles et plus particulièrement sur l'aspect structurel des langages de description des architectures (Architecture Description Languages, ADL) (Medvidovic, Taylor, 2000).

Actuellement, un grand intérêt est porté au domaine de l'architecture logicielle (Clements, 2003). Cet intérêt est motivé principalement par la réduction des coûts et les délais de développement des systèmes logiciels. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant que de le concevoir, le coder, le tester, le déboguer et le documenter. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre ces composants. Elle joue le rôle de passerelle entre l'expression des besoins du système logiciel et l'étape de codage du logiciel. Enfin, l'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage des composants logiciels.

Pour décrire l'architecture logicielle et modéliser ainsi les interactions entre les composants d'un système logiciel, deux principales approches ont vu le jour depuis quelques années : la modélisation par composants et la modélisation par objets (Garlan, 2000b ; Khammaci et al, 2005). La première approche, qui a émergé au sein de la communauté de recherche « Architectures Logicielles » décrit un système logiciel comme un ensemble de composants qui interagissent entre eux par le biais de connecteurs. Les chercheurs de ce domaine ont permis d'établir les bases intrinsèques pour développer de nouveaux langages de description d'architectures logicielles. L'architecture logicielle a permis notamment de prendre en compte les descriptions de haut niveau des systèmes complexes et de raisonner sur leurs propriétés à un haut niveau d'abstraction (protocole d'interaction, conformité avec d'autres architectures, etc.) La seconde approche est devenue un standard de description d'un système logiciel durant ces dernières années. Avec l'unification des méthodes de développement objet

sous le langage UML (Booch et al, 1998 ; Object Management Group, 1997), cette approche est largement utilisée et bien appréciée dans le monde industriel.

Les deux approches ont plusieurs points en commun. Elles se basent sur les mêmes concepts qui sont l'abstraction et les interactions entre les entités. En terme d'architecture logicielle en général, la similarité entre les deux approches est évidente. En termes d'intention, les deux approches ont pour but de réduire les coûts de développement des logiciels et d'augmenter la production des lignes de codes puisqu'elles permettent la réutilisation et la programmation à base d'objets et/ou composants. Ainsi, en tenant compte des similarités entre ces deux approches, nous rejoignons Garlan (Garlan, 2000b ; Garlan, Cheng, Kompanek, 2002) sur le principe de la réconciliation des besoins des deux approches. Par conséquent la tendance actuelle est l'utilisation conjointe de ces deux approches pour décrire les architectures logicielles. Nous voulons profiter de leur cœxistence puisque toutes les deux ont en commun, le privilège de développement d'un système afin que chaque paradigme tire profit de l'apport de l'un et de l'autre.

Motivations

Le manque de réutilisabilité est l'un des problèmes majeurs des systèmes informatiques. Il est le résultat de l'incompatibilité des principes architecturaux des composants qui interagissent entre eux. Il devient alors difficile de contrôler les interactions et les interconnexions entre composants. L'implémentation peut s'avérer de plus en plus difficile. L'absence de quelques concepts architecturaux (connecteur, configuration, …etc.), ainsi que le non-respect de l'architecture logicielle dans les plates-formes d'exécution sont les conséquences des nouveaux problèmes de maintenance et d'interopérabilité. Pour affronter ces problèmes, la solution recommandée est la définition d'un profil UML pour l'architecture logicielle, et ensuite l'intégration de ce profil dans la démarche MDA (Frankel, 2003).

Objectifs

L'objectif de nos travaux est de valider pratiquement l'architecture abstraite COSA (Component Object-based Software Architecture) basée sur la description architecturale et la modélisation par objets (Khammaci, Smeda, Oussalah, 2004 ; Smeda, Khammaci, Oussalah, 2004a). COSA est un modèle hybride basé sur la modélisation par objets et la modélisation par composants, pour décrire des systèmes logiciels. Cette approche est, empruntée au formalisme des ADLs étendu, grâce aux concepts et mécanismes objets. Elle explicite les connecteurs en tant qu'entités de première classe pour traiter des dépendances complexes entre les composants.

Contributions

Notre approche se décline de deux volets : le premier volet concerne la définition d'un profil UML 2.0 pour COSA. On utilise toutes les capacités des profils UML (méta-modèle et modèle), pour définir une spécification complète pour l'architecture logicielle. Les profils UML existants sont dédiés à un type d'application (système distribués, temps réel, etc.), alors que le profil défini dans notre cas, est indépendant de tout système. Il est spécialisé dans une activité particulière de modélisation des langages de description d'architectures. Ce profil permet d'exprimer les concepts architecturaux en UML 2.0 et donc de définir de manière formelle les concepts de l'architecture logicielle COSA en utilisant un ensemble de stéréotypes appliqués à des métaclasses issues du métamodèle UML 2.0. Un tel profil est défini pour favoriser la réutilisation des architectures logicielles cohérentes et complètes décrites en COSA. Le deuxième volet porte sur l'intégration du profil UML COSA au sein de la démarche MDA (Frankel, 2003). Nous proposons une stratégie de transformation directe par l'utilisation du profil UML à l'élaboration des modèles d'implémentations depuis des modèles d'architectures ayant un niveau d'abstraction haut et un degré de réutilisation à celui de COSA.

Organisation de la thèse

La suite de cette thèse est organisée comme suit :

Le chapitre 1 est consacré à l'étude des concepts de base de l'architecture logicielle, suivie d'une étude comparative des différentes approches. Les concepts de base des langages de description d'architecture et les langages de description d'architecture les plus représentatifs, forment le chapitre 2. Ce chapitre, présente également une comparaison des principaux ADLs et UML, selon des critères liés à la description architecturale. Le modèle multi paradigme (COSA), basé sur la modélisation par objets et la modélisation par composants, décrivant les systèmes logiciels est développé dans le troisième chapitre. Il est fait état également dans ce chapitre, de l'architecture à trois niveaux de COSA pour guider l'architecte dans son processus de modélisation d'architectures logicielles. Le chapitre 4, présente l'intégration du profil UML COSA, dans la démarche MDA.

Le dernier chapitre de cette thèse décrit les réalisations et les expérimentations que nous avons effectuées pour le profil COSA UML. La définition technique du profil UML 2.0 pour COSA, développé dans ce travail est bien explicitée.

Le bilan de ce travail, nos contributions et les résultats obtenus, composent la conclusion générale, suivie des perspectives futures.

Chapitre 1 : L'état du domaine

1.1 Introduction

Un grand intérêt est porté au domaine de l'architecture logicielle (Clements, 2003). Cet intérêt est motivé principalement par la réduction des coûts et les délais de développement des systèmes logiciels. L'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de pièces logicielles. Elle permet en effet au concepteur de raisonner sur des propriétés (fonctionnelles et non fonctionnelles) d'un système à un haut niveau d'abstraction. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre ces composants. Elle joue le rôle de passerelle entre l'expression des besoins du système logiciel et l'étape de codage du logiciel. Enfin, l'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage des composants logiciels.

Pour décrire l'architecture logicielle et modéliser ainsi les interactions entre les composants d'un système logiciel, deux principales approches ont vu le jour depuis quelques années : l'approche orientée objet, dite « *architecture logicielle à base d'objets* » (Jacobson et al. 1992 ; Brinkkemper et al. 1995 ; Booch, 1994 ; Martin 1996) et l'approche orientée composant, appelée « architecture logicielle à base de composants » (Garlan, Allen, Ockerbloom, 1994 ; Bass, Kazman, 2001 ; Clements, 2003). Il semble naturel de rapprocher ces deux approches, puisque toutes les deux ont en commun de privilégier le développement d'un système, et ce depuis sa phase de spécification jusqu'à son implémentation. Bien que chacune d'elles porte l'accent sur des aspects différents du développement logiciel en termes de niveaux d'abstraction, de granularités et de mécanismes opératoires. Il n'y a pas de frontière très nette entre les deux. En ce sens, il

n'est pas étonnant de retrouver des points communs dans ces deux approches tant au niveau des motivations, des techniques que des méthodes. Les deux approches focalisent la réduction des coûts de développement des applications et insistent sur la réutilisation et la performance des produits logicielles. Les développeurs et les concepteurs de système à base de composants sont souvent tentés d'utiliser les modèles à objet, comme support de spécification, de conception et d'implémentation (Garlan, 2000a ; Khammaci, Smeda, Oussalah, 2005 ; Medvidovic et al. 2002).

Dans ce chapitre, nous présentons les concepts de base de ces deux approches ainsi que leurs avantages et leurs inconvénients. Nous présentons également les points forts des deux approches et ses points communs. Enfin, une brève discussion envers la cœxistence de la description à base de composants architecturaux et la description orientée objet.

1.2 Approche de modélisation par objet

Les architectures logiciels à base d'objets décrivant les systèmes logiciels comme une collection de classes (les entités à abstraire et l'encapsulation des fonctionnalités) qui peuvent avoir des objets ou instances et communiquent entre eux, via des messages. Avec l'unification des méthodes de développement objet sous le langage UML (Unified Modeling Language), cette approche est largement utilisée et bien appréciée dans le monde industriel.

1.2.1 Les concepts de base de la modélisation par objet

Le paradigme objet est principalement basé sur l'extraction et l'encapsulation des caractéristiques et des fonctionnalités des classes. Une classe peut être instanciée pour obtenir des objets. Un objet (instance) a des propriétés (attributs) et un comportement (méthodes). Les objets des différentes classes communiquent par l'intermédiaire d'envois de messages. La figure 1.1, montre les concepts de base de l'approche objet, que sont les classes, les objets et leurs relations. La figure montre également quelques mécanismes tels que l'héritage et la composition.

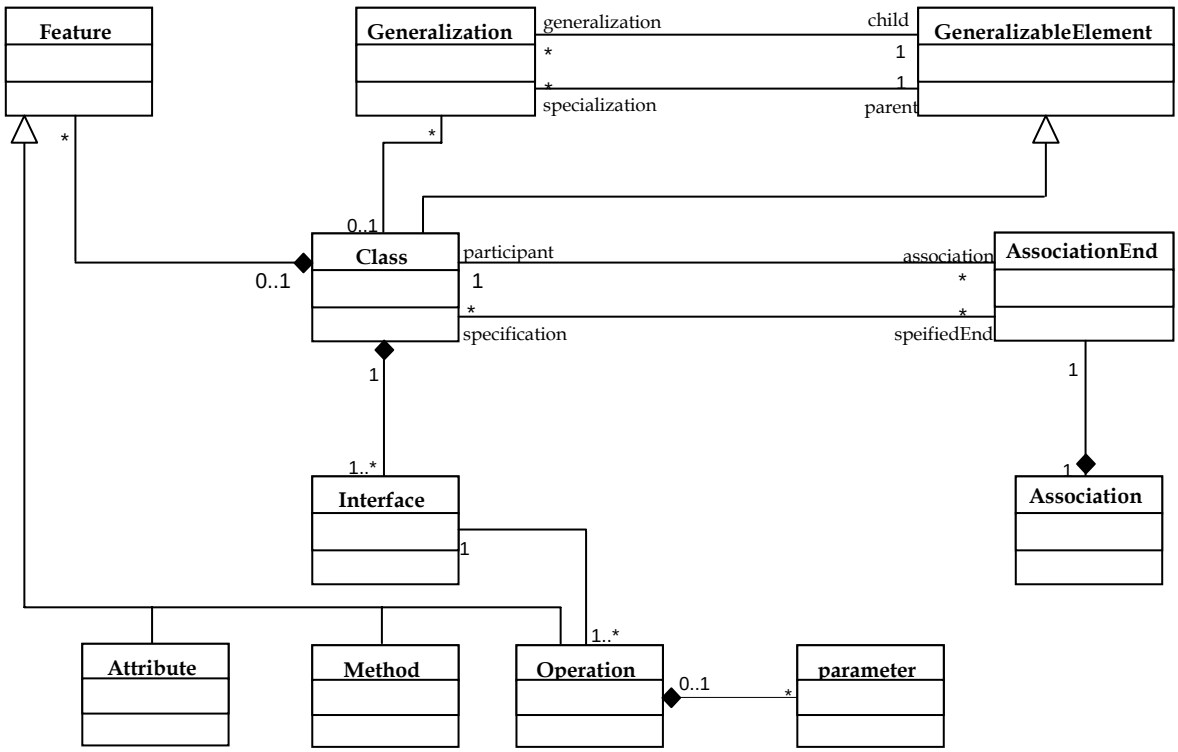


Figure 1.1. *Les concepts de base de l'approche objet (extrait d'UML 1.1 (OMG, 1997)).*

1.2.2 Les langages de la modélisation par objet

La modélisation par objet (OOM : Object Oriented Modeling) est une technique systématique utilisée pour capturer, analyser et décrire le comportement d'un système. Le comportement est décrit en terme d'objets qui constituent le système, sa structure et comment les uns interagissent avec les autres. Ainsi le comportement décrit aussi bien un objet individuel, que le comportement des objets en interactions.

Dans les années 90, on distinguait trois méthodes Object-Modeling Technique (OMT) de Rumbaugh (Rumbaugh et al., 1991), Object-Oriented Analysis and Design (OOD) de Booch (Booch, 1994), Object Oriented Software Engineering (OOSE) de Jacobson (Jacobson et al., 1992). Aujourd'hui, elles ne donnent pas complètement satisfaction.

En 1995, la plupart des notations orientées objets sont unifiées et intégrées dans le Language de modélisation unifié (UML) (Booch, Rumbaugh, Jacobson, 1998). UML est devenu un langage standard de spécification, de visualisation, de construction et de documentation des systèmes logiciels. UML, représente une collection des meilleures

pratiques d'ingénierie ayant un grand succès dans la modélisation des systèmes larges et complexes. UML utilise généralement des notations graphiques pour représenter les aspects structurels, dynamiques et fonctionnels d'un système logiciel. Mais, la notation graphique est intrinsèquement limitée lors de la spécification des contraintes complexes. Donc, UML décrit un ensemble de contraintes en langage naturel, et à l'aide du langage OCL (Object Constraint Language (Warmer, Kleppe, 1998)). Ces contraintes peuvent être définies au niveau du méta-modèle comme au niveau du modèle. OCL permet une conception riche et correcte.

Finalement, avec l'apparition de la dernière version d'UML (UML 2.0 (Object Management Group, 2004a)), UML improuve la définition des composants et ses interfaces par l'introduction de la notion des ports et la séparation des services fournis des services requis d'un composant. Mais, UML 2.0 définit toujours implicitement les interactions entre les composants (les interactions en UML 2.0 sont les propriétés des composants) et n'introduit pas la notion du connecteur comme entité de première classe.

1.3 Approche de modélisation par composants

La modélisation à base de composants architecturaux permet aux développeurs de faire abstraction des détails d'implémentation, de se concentrer sur des éléments de plus haut niveau comme les différentes vues et structures des systèmes complexes et de raisonner sur leurs propriétés (protocole d'interaction, conformité avec d'autres architectures, etc.) (Medvidovic, Taylor, 2000).

Cette approche décrit un système comme un ensemble de composants (unité de calcul ou de stockage) qui interagissent entre eux par le biais de connecteurs (unités d'interactions). Leurs objectifs consistent à réduire les coûts de développement, à améliorer des modèles, à faire partager des concepts communs aux utilisateurs et enfin à construire des systèmes hétérogènes à base de composants réutilisables sur étagères (COTS, *commercial off the shelf*).

L'architecture logicielle à base de composants (composants architecturaux) (Kazman et al. 2001 ; Clements, 2003) décrit les systèmes logiciels comme un ensemble de composants (encapsulation des fonctionnalités) qui interagissant entre eux par l'intermédiaire des connecteurs (encapsulation de communication), comme indiqué dans la figure 1.2.

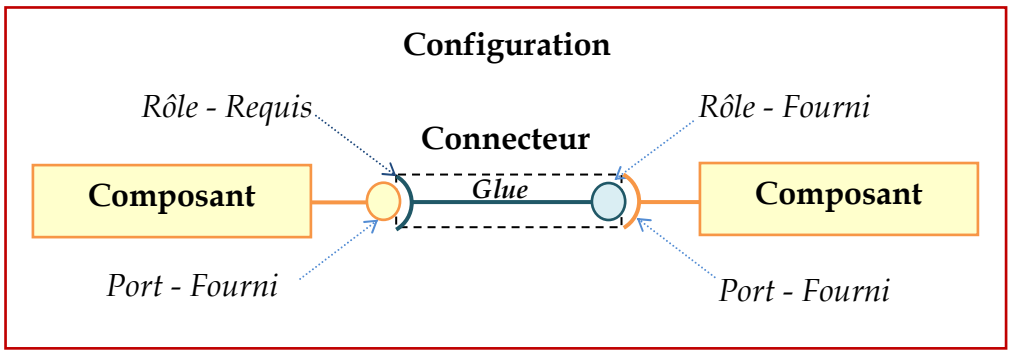


Figure 1.2. *Description architecturale des systèmes à base de composants.*

Pour asseoir le développement des architectures à base de composants, plusieurs langages de description (ADL : Architecture Description Language) ont été proposés. Les ADLs offrent un niveau d'abstraction élevé pour la spécification et le développement des systèmes logiciels, qui se veut indépendant des langages de programmation et de leurs plateformes d'exécution. Dans le développement à base de composants Component-Based Software Engineering (CBSE) (Hasselbring, 2002), il y a beaucoup d'avantages. On inclut la réutilisation des modèles, le partage des concepts communs aux utilisateurs de système et la réduction des coûts de développement.

1.3.1 Les concepts de base de modélisation par composants

1.3.1.1 Définitions notables

Le concept composant, est né dans le vocabulaire informatique depuis la naissance du génie logiciel. Il a cependant désigné d'abord des fragments de code, pour englober ensuite toute unité de réutilisation (Barbier et al. 2004).

A l'heure actuelle, il n'existe toujours pas de définition précise et normalisée de l'architecture logicielle, et il en existe de nombreuses interprétations. A titre d'exemple, le SEI (Software Engineering Institute) a documenté et catalogué des centaines de définitions.

Une définition très populaire de l'architecture logicielle a été avancée par Garlan et Shaw (Garlan et Shaw, 1993). Garlan et Shaw ont proposé qu'une architecture logicielle pour un système spécifique soit représentée comme "un ensemble de composants de calcul - ou tout simplement de composants - accompagné d'une description des interactions entre ces composants - les connecteurs".

A l'origine, l'approche par composants a été fortement inspirée des composants de circuits électroniques. L'expérience gagnée dans cette discipline a largement contribué aux idées de composants et de réutilisation. Concernant les composants électroniques, il suffit de connaître leur principe de fonctionnement et la façon dont ils communiquent avec leur environnement (tensions d'entrée, de sortie,…) pour construire un système complet sans pour autant dévoiler les détails de leur implémentation. Comme le souligne Cox (Cox 86), l'idée de circuits logiciels intégrés conduit à la notion d'éléments logiciels qui peuvent être connectés ou déconnectés d'un circuit plus complexe, remplacés et/ou configurés. Les constructeurs d'applications adoptent de plus en plus cette démarche. Il s'agit, de composer des composants logiciels de nature très diverses pour construire une application. On ne parlera alors plus de programmation d'applications mais plutôt de composition d'applications.

Aujourd'hui le développement d'applications à base de composants constitue une voie prometteuse. Pour réduire la complexité de ce développement et le coût de maintenance, et accroître le niveau de réutilisablilité, deux principes fondamentaux doivent être respectés :

Acheter plutôt que Construire et Réaliser plutôt qu'Acheter (Mcllroy, 1968).

Le concept de réutilisation de composant logiciel a été introduit par McIlroy en 1968 (Mcllroy, 1968) dans un conférence de l'OTAN consacré à la crise du logiciel suite à un échec patent de la part de développeurs pour livrer des logiciels de qualité à temps et à prix compétitif. L'idée est de mettre en place une industrie du logiciel pour accroître la productivité des développeurs et réduire le temps de mise sur le marché (*time-to-maket*). Pour ce faire, les développeurs construisant des applications à partir de composants logiciels sur étagère (COTS, *Commercial off the shelf*) plutôt que de créer à partir de rien (*from scratch*). Ainsi, la construction d'applications pourrait se faire d'une manière plus

rapide en assemblant des composants préfabriqués. La popularité croissante des technologies logicielles à base de composants, à l'instar des technologies à objets, est entrain de s'installer et de se propager auprès de millions de programmeurs. Les nouvelles technologies logicielles adoptent de plus en plus le concept de « composant » comme clé de voûte de la réutilisation pour construire rapidement des applications. Parmi ces technologies, nous pouvons citer : ActiveX, OLE, DCOM (Rubin, Brain, 1998), .NET de Microsoft (Grim, 1997), EJB de SUN (Roman et al. 2004), CORBA (Common Object Request Broker Architecture) de l'OMG (Baker, 1997). Elles permettent d'utiliser des composants et des objets à forte granularité, distribués et complexes. Cependant, plusieurs questions se posent dès qu'on aborde la problématique des composants. Par exemple : quelle est la définition d'un composant ? Sa granularité ? Sa portée ? Comment peut-on distinguer et rechercher les composants de manière rigoureuse ? Comment peut-on les manipuler ? Les assembler ? Les réutiliser ? Les installer dans des contextes matériels et logiciels variant dans le temps, Les administrer, Les faire évoluer ? …etc.

Le but de cette section est d'apporter les notions de base nécessaires à la compréhension du paradigme de composants pour enfin caractériser leurs propriétés.

1.3.1.2 Les concepts de la description architecturale

Les concepts et les notations de l'approche de description d'architecture logicielle (composants architecturaux) sont similaires aux notations de modélisation à base d'objets. Aussi il y a une diversité considérable dans les capacités des différentes ADLs, qui se partagent les mêmes concepts de base et qui déterminent les mêmes fondations des concepts pour la description d'architecture logicielle. Les concepts essentiels de cette approche sont :

- Les *composants* sont définis comme des unités de calcul ou du stockage pour lesquelles est associée une unité d'implantation. Un composant dans une architecture peut être aussi petit qu'une procédure simple ou aussi grande qu'une application entière. Il peut exiger ses propres donnés et/ou des espaces d'exécution comme il peut les partager avec d'autres composants. La plupart des systèmes à base de composants

peuvent avoir plusieurs interfaces : chaque interface définit un point d'interaction entre un composant et son environnement.

- Les *connecteurs* représentent les interactions entre les composants et les règles qui régissent ces interactions correspondent aux lignes dans les descriptions de type "boites-et-lignes". Ce sont des entités architecturales qui lient des composants ensemble et agissent en tant que médiateurs entre elles. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure, et l'émission d'événements. Les connecteurs peuvent également représenter des interactions complexes comme un protocole Remote Procedure Call (RPC) de client-serveur ou un lien de SQL entre une base de données et une application contrairement aux composants.

- Les *interfaces* d'un composant sont des points de communication qui lui permettent d'interagir avec son environnement ou avec d'autres composants.

- Les *propriétés* représentent les informations sémantiques des composants et de leurs interactions.

- Les *contraintes* représentent les moyens permettant à un modèle d'architecture de rester valide durant toute sa durée de vie et de prendre en compte l'évolution et le remplacement des composants logiciels dans cette dernière. Ces contraintes peuvent inclure des restrictions sur les valeurs permises de propriétés, sur l'utilisation d'un service offert par un composant et garantir la validité des résultats retournés par ce service.

- Les *configurations* architecturales représentent les graphes de composants et de connecteurs et la façon dont ils sont reliés entre eux. Cette notion est nécessaire pour déterminer si les composants sont bien reliés, si leurs interfaces s'accordent, si les connecteurs correspondants permettent une communication correcte. La combinaison de leurs sémantiques aboutit au comportement désiré, qui vient en appui des modèles de composants et de connecteurs. Les descriptions des configurations permettent l'évaluation des aspects distribués et concurrents d'une architecture, comme par exemple, la possibilité de déterminer des verrous, de connaître le potentiel de

performance, de fiabilité, de sécurité…etc. Le rôle clé des configurations est de faciliter la communication entre les différents intervenants dans le développement d'un système. Leur but est d'abstraire les détails des différents composants et connecteurs. Ainsi, elles décrivent le système à un haut niveau d'abstraction qui peut être potentiellement compris par des personnes avec différents niveaux d'expertise et de connaissances techniques.

- La *composition/assemblage* permet de construire des applications complexes à partir de composants simples. La composition ou assemblage permet de lier un composant demandant des services à d'autres composants offrant ces dits services.

- Les *styles architecturaux* sont des aspects intéressants, même s'ils ne sont présents que dans certains systèmes à base de composants (Garlan, 2000b). Le choix d'un style architectural permet de :

1. déterminer l'ensemble des vocabulaires désignant le type des entités de base (composants et connecteurs).
2. spécifier l'ensemble des règles de configuration qui déterminent les compositions et les assemblages, permis entre ces éléments.
3. donner la sémantique de configuration. Exemple : un système multi -agents communiquant via un tableau de bord, signifie que ces agents partagent des connaissances.
4. déterminer les analyses qui peuvent être réalisées sur un système construit selon un tel style. Par exemple : vérification de la causalité dans un système distribué basé sur la communication par événement.

1.3.1.3 Les trois dimensions d'un composant

Un composant peut être de deux natures :

- *produit* : il s'agit d'une entité « *building block* » autonome passive (entité logicielle ou entité conceptuelle) qu'il est possible d'adapter. Les composants logiciels et les bibliothèques de fonctions mathématiques en font partie.
- *processus* : un composant processus correspond à une suite d'actions qu'il faut réutiliser pour obtenir un produit final. Ces actions sont souvent encapsulées dans un

processeur (unité de traitement). Un composant processus possède en général des fragments de démarche.

Au regard des travaux existants un composant produit ou processus doit refléter les trois dimensions qui sont le niveau d'abstraction, le mode d'expression et le domaine (cf. figure 1.3) (Oussalah, Khammaci, Smeda, 2005).

La première dimension ou *niveau d'abstraction* permet d'exprimer les degrés de raffinement d'un composant et ce depuis sa spécification. Elle est considérée comme étant le plus haut niveau d'abstraction. Son code source représente le plus bas niveau. Chaque niveau d'abstraction peut lui être associé *un niveau de transparence* qui définit le niveau de visibilité des détails internes d'un composant lors de sa/son (ré) utilisation. Le niveau de transparence d'un composant peut être de type :

- *boîte noire* : l'interface du composant fait abstraction de son implémentation. L'interaction avec l'extérieur se fait uniquement à travers l'interface utilisateur. Ce qui permet de donner la possibilité de remplacer un composant par un autre, réalisée par l'offre de la même interface.
- *boîte blanche* : le composant rend transparent tous les détails de l'implémentation. L'interaction avec l'extérieur peut se réaliser non seulement à travers l'interface du composant mais aussi à travers son implémentation. Ce type de composant, a l'avantage de fournir toute l'information relative à son implémentation.
- *boîte grise* : il s'agit d'un niveau de transparence intermédiaire entre les composants de type boîte noire et boîte blanche. Si les détails d'implémentation peuvent être révélés pour comprendre la réalisation du composant, ils ne peuvent pas être sujet à des modifications émanant de l'extérieur : l'interaction se fait uniquement à travers l'interface.

La deuxième dimension ou *mode d'expression* permet de décrire les différents modèles de représentation d'un composant (représentation textuelle, graphique, flot de données, implémentation). La description d'un composant doit être simple, compréhensible avec une sémantique pas nécessairement définie de manière formelle mais au moins claire. En effet, les participants à l'élaboration de composants (concepteurs, développeurs, utilisateurs, managers…) peuvent requérir différents modes d'expression.

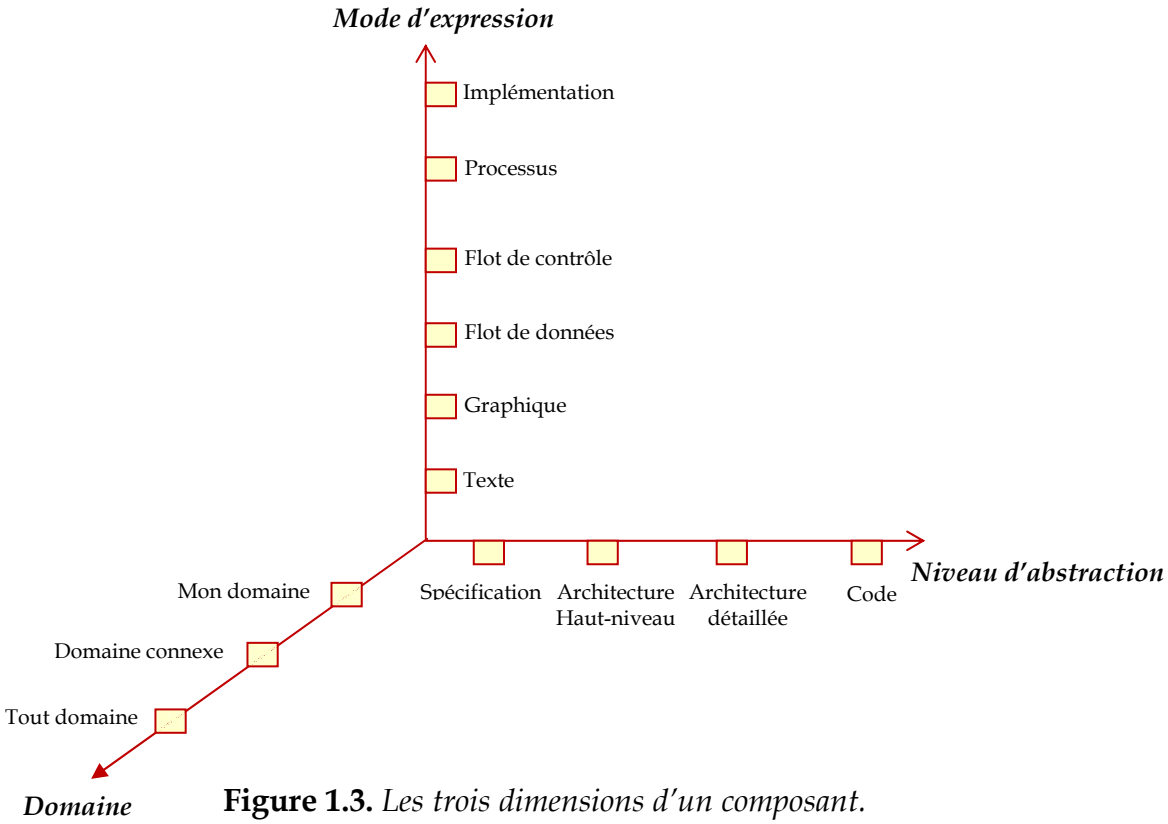


Figure 1.3. *Les trois dimensions d'un composant.*

Les utilisateurs peuvent se contenter d'une description graphique de haut niveau (type boîte et flèche). Les développeurs voudront détailler par exemple les modèles de connecteurs et de composants, alors que les managers peuvent requérir une vue du processus de développement. Le mode d'expression d'un composant dépend également de sa granularité. La notion de granularité de composant recouvre, selon les langages de représentation, aussi bien des unités atomiques telles que les structures de données, les fonctions mathématiques ou de véritables structures composées d'éléments et de liens.

Du point de vue de la granularité, nous pouvons citer :

- *les composants dits simples* ou de *fine granularité* sont directement liés à une construction syntaxique du langage de développement exprimée souvent à l'aide d'un langage d'implémentation comme les classes en programmation objet, les packages en programmation Ada ou des fonctions dans la programmation usuelle.
- *les composants à granularité moyenne* sont les composants logiciels utilisés dans la production du logiciel ou les patrons de conception sont souvent exprimés de façon textuelle ou graphique.

- *les composants dits complexes ou à forte granularité*, s'apparentent à une véritable structure complexe comme les infrastructures logicielles (*frameworks*), les systèmes d'exploitation, les serveurs ou les *blackboard*.

La troisième dimension (*domaine)* reflète les différents domaines structurés en couches sur lesquels repose un système donné. Cette structuration permet à un composant appartenant à un domaine particulier, de ne pouvoir interagir, qu'avec seulement les composants de la même couche ou de la couche adjacente. Les composants du « Tout Domaine » constituent la couche du plus bas niveau. Ils sont indépendants du domaine d'application, comme par exemple le cas de systèmes, de compilateurs, de base de données…etc. Au niveau plus haut « Domaine connexe », on trouve des composants appartenant à des domaines connexes de l'application en cours de construction, comme par exemple les interfaces utilisateurs, les simulateurs, ...etc. Enfin en haut de la hiérarchie, les composants de « Mon Domaine » sont spécifiques au domaine d'application choisi, par exemple les composants de circuits intégrés. Il est clair, que le nombre de couches de domaines n'est pas figé et dépend de la maturité des domaines d'application, en général. Enfin, cette troisième dimension caractérise la portée des composants, soit leur degré de réutilisabilité. Nous pouvons recenser dans la littérature :

- *Les composants génériques ou indépendants d'un domaine* : ce sont des composants généraux qui ne dépendent pas d'un domaine particulier, les objets graphique, les patrons de conception de Gamma et al. (Gamma et al. 1995) et les types de données abstraits sont des exemples de ce type de composants,
- *Les composants métiers ou dépendants du domaine* : le concept de composant métier résulte de celui d'objet métier. Les composants métiers sont des composants réutilisables à travers les applications d'un même domaine,
- *Les composants orientés applications* : ce sont des composants spécifiques à une application donnée. Ils ont peu réutilisables. Ce type de composants est généralement engendré par des réutilisations ad hoc non planifiées.

1.3.2 Les langages de modélisation par composants

Les langages de description d'architecture sont des langages formels qui peuvent être utilisés pour représenter l'architecture d'un système logiciel. L'architecture joue un rôle important dans les systèmes de développement et d'acquisition. Dans les dernières décades, la communauté de recherche en architecture logicielle propose plusieurs formalismes de description d'architecture que l'on appelle les langages de description d'architectures (en anglais ADL : Architecture Description Language) pour spécifier la structure et le comportement des architectures logicielles. Malheureusement, il y a plusieurs ADLs et aucun signe n'indique que ces ADLs convergent vers une notation standard pour spécifier leurs architectures. Les plus connus de ces langages sont : Aesop (Garlan, Allen, Ockerbloom, 1994), Rapide (Luckham et al. 1995), Unicon (Shaw, DeLine, Zelesnik, 1996), C2 (Taylor, 1996), Darwin (Magee, Karmer, 1996), Wright (Allen, Garlan, 1997 ; Allen, Douence, Garlan, 1998), ACME (Garlan, Monroe, Wile, 2000), COSA (Khammaci, Smeda, Oussalah, 2004). Les ADLs proposent des notations formelles pour décrire les concepts de la description architecturale et permettent une analyse rigoureuse des architectures.

1.4 Similarité entre la modélisation à base d'objets et la modélisation à base de composants

La modélisation à base d'objets et la modélisation à base de composants ont plusieurs concepts en commun. Ces deux approches se basent sur les mêmes concepts, qui sont l'abstraction et l'interaction des composants. Dans les descriptions d'architectures logicielles à base de composants, les composants et les connecteurs sont les concepts de base de description des systèmes logiciels. Les composants sont des abstractions qui contiennent des fonctionnalités de stockage de données dans des blocs réutilisables alors que les connecteurs définissent des abstractions qui englobent les mécanismes de communication, de coordination et de conversion entre les composants (Smeda, Khammaci, Oussalah, 2004a ; Alti, Khammaci, 2005). D'autre part les descriptions d'architectures logicielles à base d'objets, les classes sont des abstractions des données, et les fonctions serves comme des interfaces d'accès à ces données.

En général, en terme d'architecture, la similarité entre deux domaines est évidente. En terme d'intention, les deux approches favorisent :

- la réduction des coûts de développement des applications,
- la programmation à base de composants,
- la réutilisation (Perry, Wolf, 1992 ; Shaw, Garlan, 1996).

Les deux approches ont permis aux développeurs de faire abstraction des détails d'implémentation, afin de se concentrer sur des éléments de plus haut niveau comme les différentes vues et structures de systèmes informatiques (Medvidovic, Taylor, 2000).

1.4.1 Avantages et inconvénients de la modélisation à base d'objets

Les modèles à base d'objets présentent plusieurs avantages :

- ils sont familiers à une large communauté d'ingénieurs et de développeurs du logiciel,
- ils se basent sur des méthodologies bien définies pour développer des systèmes à partir d'un ensemble de besoins,
- ils sont supportés par des outils commerciaux,
- ils fournissent plusieurs vues (structuraux et comportementaux) de description des systèmes,
- ils présentent le langage UML comme un standard des notations unifiées de plusieurs notations des modélisations orientées objets,
- ils fournissent souvent une correspondance directe de la spécification à l'implémentation.

Cependant, force est de constater que l'approche objet souffre d'un certain nombre de lacunes par rapport aux systèmes à base de composants. Nous en donnons les plus significatives :

- l'approche objet a montré des limites importantes en terme de granularité et dans le passage à l'échelle. Le faible niveau de réutilisation des objets est dû en partie au fort couplage des objets. En effet, ces derniers peuvent communiquer sans passer par leur interface,
- la structure des applications objets est peu lisible (un ensemble de fichiers),

- la plupart des mécanismes objets sont gérés manuellement (création des instances, gestion des dépendances entre classes, appels explicites de méthodes,…)

Par ailleurs, les approches objets :

- spécifient seulement les services fournis par les composants d'implémentation mais ne définissent en aucun cas les besoins requis par ces composants,
- ne fournissent pas (ou peu) de support directe pour caractériser et analyser les propriétés non-fonctionnelles,
- proposent peu de solutions pour faciliter l'adaptation et l'assemblage d'objets,
- prennent en compte difficilement les évolutions d'objets (ajout, suppression, modification, changement de mode de communication,…)
- Il y a peu ou pas d'outils pour déployer (installer) les exécutables sur les différentes sites,

Enfin, les modèles objets :

- ne sont pas adaptés à la description de schémas de coordination et de communication complexes. En effet, ils ne se basent pas sur des techniques de construction d'applications qui intègrent d'une certaine façon l'homogénéité des entités logicielles provenant de diverses sources,
- disposent de faibles supports pour les descriptions hiérarchiques, rendant difficile la description de systèmes à différents niveaux d'abstraction,
- permettent difficilement la définition de l'architecture globale de système avant la construction complète (implémentation) de ses composants. En effet les modèles à objets exigent l'implémentation de leurs composants avant que l'architecture ne soit complètement définie,
- ne permettent pas facilement la recherche des relations d'interactions dans une architecture (surtout de pointeurs et autres) et requièrent forcément l'inspection de tout le système,
- ne fournissent pas un support direct de définition des propriétés non fonctionnelles d'analyse et de vérification des systèmes,

- ne supportent pas la définition des styles architecturaux qui facilite la conception des domaines spécifiques et l'analyse des propriétés des systèmes.

1.4.2 Avantages et inconvénients des modèles à base de composants

Les modèles à base de composants présentent plusieurs avantages :

- les interfaces sont en général des entités de première classe décrites explicitement par des ports et des rôles,
- les interactions sont séparées par des calculs (séparation de préoccupations) et explicitement définies dans la plupart des ADLs (Medvidovic, Taylor, 2000),
- les connecteurs sont des entités de première classe. Ils sont définis explicitement par la séparation de leurs interfaces et de leurs implémentations via les ports et les rôles. En effet, les connecteurs permettent de décrire des communications, et prennent plusieurs formes complexes comme un nouveau mécanisme d'intégration des composants (Garlan 2000b ; Shaw, Garlan, 1996),
- les propriétés non fonctionnelles sont prises en compte comme la performance, la sécurité, la portabilité, la conformité aux standards et le partage, utiles à l'analyse des systèmes,
- les représentations hiérarchiques sont sémantiquement plus riches que de simples relations d'héritage. Elles permettent, par exemple, les représentations multiples d'une architecture (Garlan 2000a),
- Ils supportent l'association plus de modèles détaillés dans une partie individuelle d'une architecture,
- Ils définissent des styles architecturaux à la base d'un vocabulaire de conception et de spécialisation de description des architectures spécifiques. Aussi, un ensemble des contraintes d'utilisation de ces vocabulaires,
- Ils permettent facilement la définition de l'architecture globale de système avant la construction complète (implémentation) de ses composants.

Les modèles à base de composants présentent des avantages comme ils présentent des inconvénients:

- ils sont connus seulement par les communautés académiques,

- ils ne supportent pas plusieurs vues. En effet, ces vues sont très importantes puisque différents aspects de système (i.e. comportement envers structural) requièrent différents besoins de description,
- ils se basent sur différentes notations et plusieurs approches quelques fois ambigues,
- ils ne fournissent pas souvent une correspondance directe, entre la spécification et l'implémentation,
- fournit seulement les modèles à niveau élevé, sans expliciter comment ces modèles peuvent être reliés au code source. Comme le souligne Garlan (Garlan 2000a), de telles liaisons sont importantes pour préserver l'intégrité de la conception.

1.4.3 Similarités et différences en terme de concepts de base

Le tableau 1.1 montre les similarités et les différences entre la modélisation à base d'objets et celle à base de composants en terme de concepts de base (Khammaci, Smeda, Oussalah, 2005).

1.5 Cœxistence des deux modélisations pour la description de l'architecture logicielle

A partir de la comparaison de la section précédente, entre l'approche à base d'objets et celle à base de composants, on peut noter que chaque approche a ses avantages et ses inconvénients et chacun d'eux a ses propres caractéristiques.

Concepts de base	Orientée - composant	Orientée - objet
Entités d'encapsulation des fonctionnalités	Composant	Classe
Entités d'encapsulation de communication	Connecteur	Méthode
Communication inter-entitiés	Ports et rôles	Envoi de messages
Structure du système	Systèmes et représentation	Programme et package
Interface	Interfaces des composants et connecteurs	Interfaces des classes
Propriétés	Fonctionnelles et non fonctionnelles	Attributs (non fonctionnels)

Table 1.1. *Les concepts de base des deux approches.*

La question qu'on peut se poser est : *Quelle est la meilleure manière de cœxistence entre ces deux approches ?*

Plusieurs travaux ont été réalisés sur la cœxistence des deux approches (Egyed, Krutchen, 1999 ; Hofmeister, Nord, Soni, 1999 ; Selic, Rumbaugh, 2000 ; Garlan, Cheng, Kompanek, 2002 ; Kruchtan, Selic, Kozaczynski, 2001 ; Medvidovic et al. 2002 ; Smeda, Khammaci, Oussalah, 2004a). Nous présentons dans cette section quelques perspectives pour répondre à cette question. La première se concentre sur les aspects comportementaux des architectures (interaction et contraintes), la deuxième, sur les aspects structuraux des architectures (composants, connecteurs, …etc.), la troisième sur les vues de modélisation des architectures logicielles et la dernière perspective focalise sur les mécanismes opérationnels des architectures logicielles.

1.5.1 La modélisation à base de composants utilisant la modélisation à base d'objets

Dans cette perspective, on se base sur la possibilité d'utiliser UML (Unified Modeling Language) (Booch, Rumbaugh, Jacobson, 1998) comme un point de départ d'intégration de la description des architectures logicielles ou composantes architecturaux dans le monde industriel (Medvidovic, Taylor, 2000). Le principal but de cette étude est d'exploiter la capacité d'expressivité d'UML à la modélisation des architectures logicielles avec les techniques des modélisations à base de composants. Dans ce contexte, les auteurs ont défini des critères d'évaluation et d'adaptation d'UML pour représenter effectivement l'architecture logicielle (Medvidovic, Taylor, 2000). Ces critères sont :

1. UML doit être bien adapté pour modéliser les aspects structuraux d'un système.
2. UML doit capturer les variétés stylistiques décrites explicitement ou implicitement par les ADLs.
3. UML doit modéliser tous les aspects comportementaux d'un système qui sont fournis par la plupart des ADLs.
4. UML doit être capable de modéliser les aspects des paradigmes d'interaction des composants.

5. UML doit être capable de capturer les contraintes de violation de la structure, le comportement, l'interaction et le style.

Pour modéliser des architectures logicielles, trois stratégies sont possibles:

1. Utilisation UML "tel qu'il est ".
2. Utilisation des mécanismes d'extensibilité.
3. Augmentation du méta - modèle UML pour supporter directement les besoins des composants architecturaux.

Chaque stratégie a ses avantages et ses inconvénients. L'évaluation des trois stratégies nous permet d'éliminer la troisième car la nouvelle version d'UML devient de plus en plus complexe. Ceci à un impact sur la facilité d'utilisation de ce langage : la nouvelle version d'UML perd son caractère standard et devient par conséquent, incompatible avec les ateliers UML existants.

1.5.1.1 UML : langage de description d'architectures

L'intérêt principal de l'utilisation d'UML, comme modélisation des architectures logicielles décrites par les ADLs, consiste en une compréhension universelle de cette dite modélisation. De plus, une telle modélisation peut être manipulée par des ateliers UML, supportant les étapes de développement suivantes: expression des besoins, conception et implémentation. Cette stratégie consiste à utiliser les notations offertes par UML pour représenter les concepts architecturaux des ADLs tels que composants, connecteurs, rôles, ports et configurations. Pour démontrer cette stratégie, Medvidovic et al. ont utilisé la norme UML1.x pour modéliser les ADLs (C2, Wright et Rapide) (Medvidovic et al. 1996; Medvidovic et al. 2002). Cependant, pour deux raisons, cette stratégie ne satisfait pas complètement tous les besoins structuraux de description d'architecture:

- UML ne fournit pas des notations spécifiques pour les artéfacts de la modélisation architecturelle.
- Les règles d'un style d'architecture donnée, reflètent celle d'un ADLs.

1.5.1.2 Extensibilité d'UML pour modéliser la description d'architectures

UML est un langage de modélisation *générique* pouvant être adapté au domaine d'architecture logicielle grâce aux mécanismes d'extensibilité offerts par ce langage, tels que, les stéréotypes, les valeurs marquées et les contraintes. Les extensions UML ciblant un domaine particulier forment des profils UML. Les mécanismes d'extensibilité offerts par UML permettent d'étendre UML sans modifier son métamodèle. Medvidovic et al. (Medvidovic et al. 2002), proposent d'utiliser des extensions d'UML (stéréotypes, valeurs marquées, contraintes) pour incorporer les concepts de trois langages de description d'architectures (C2, Wright et Rapide). Dans (Goulao, Abreu, 2003), les auteurs établissent un profil UML2.0 pour l'ADL ACME. Ils regroupent des concepts minimaux et communs à tous les ADLs. Les travaux de (Roh, Kim, Jeon, 2004), signalent quelques faiblesses liées notamment à la représentation proposée du connecteur (au sens d'ADL) en UML2.0 et proposent un ADL générique sous forme d'un profil UML 2.0. Dans ce travail, on note notamment l'utilisation des collaborations UML 2.0 pour représenter des connecteurs ADL. De plus, un autre aspect intéressant se dégage : type et instance de connecteur sont modélisés par deux stéréotypes. En effet, le type de connecteur est défini comme un stéréotype à base de métaclasse *Collaboration* d'UML 2.0. L'instance du connecteur est définie comme un stéréotype à base de métaclasse *Connector* d'UML.

1.5.2 Sélection de notations UML pour représenter les descriptions d'architectures

Selon Garlan et al. (Garlan, Cheng, Kompanek, 2001), une autre perspective de cœxistence de la description d'architecture logicielle et de la modélisation orientée objet, est la sélection de notations UML pour représenter des éléments architecturaux. L'évaluation de chaque sélection est proposée ensuite. On distingue, quatre stratégies de représentation des principaux concepts architecturaux en UML1.x. Ces dernières se focalisent autour de la représentation des types de composants et des instances de composants en UML1.x. Les auteurs proposent les alternatives suivantes :

- *classes et objets* : les types de composants sont représentés par des classes UML1.x et les instances de composants par des objets.
- *classes et classes* : les types de composants et les instances de composants sont représentées par des classes.
- *Types et instances de composants*: les types de composants sont représentés par des types de composants UML1.x et les instances de composants par des instances de composants UML1.x.
- *Types et instances de subsystems* : les types de composants sont représentés par des subsystems UML et les instances de composants par des instances de subsystems.

UML-RT (Selic, 1999) est une variante particulière de ces stratégies, basée sur l'implantation des concepts ADL vers UML-RT (Cheng, Garlan, 2001). Cette étude illustre bien que chaque sélection de ces quatre est méritée. Mais, il se trouve qu'aucun d'eux n'est capable de supporter les besoins d'architectures en termes de correspondance sémantique, de compréhension et de complétude.

1.5.3 Représentation des vues architecturales en UML

C'est dans cette perspective, que les auteurs se sont orientés vers la représentation des modèles de vues des architectures logicielles. Krutchen (Kruchten, 1995) présente le modèle 4+1 vues des architectures logicielles. Les quatre principaux vues sont de type : logique, processus, développement et physique. Ces vues permettent de mieux capturer l'architecture du système logiciel. Dans (Soni, Nord, Hofmeister, 1995), les auteurs identifient les quatre catégories structurales de l'architecture logicielle, conceptuelle, module d'interconnexion, exécution et code. Les différentes vues visent différents aspects d'ingénierie. La séparation de ces aspects aide l'architecte à prendre une décision sur la conception architecturale. Chaque vue a ses éléments de description. Une vue conceptuelle décrit des éléments en termes de domaine. Une vue du modèle d'interaction raffine la vue conceptuelle, elle fournit des fonctions et des couches décompositionnelles du système. Les vues d'exécution et le code correspondent aux vues structurelles et dynamiques de Krutchen respectivement. Dans (Hofmeister, Nord, Soni, 1999). Ces auteurs concluent qu'UML est insuffisant à la description des aspects

dynamiques d'une architecture et des séquences d'activités. Par contre, UML est mieux adapté pour décrire une structure statique d'une architecture, une variante des configurations conceptuelles et des séquences d'activités spécifiques.

1.5.4 Combinaison de la modélisation à base de composants et à base d'objets

Dans cette perspective, nous pouvons citer les travaux sur la prise en compte de mécanismes opérationnels par les composants architecturaux. Ce travail a abouti à la proposition de l'approche COSA (Component Object Software Architecture) qui combine des concepts issus des composants architecturaux et des concepts inhérents à l'approche objet, et qui bénéficie des qualités de chacune des deux approches (Smeda, Khammaci, Oussalah, 2004a ; Smeda, Khammaci, Oussalah, 2004b) permet de mieux décrie l'architecture d'un système logiciel. Il décrit un système comme une collection des composants qui interagissent entre eux par l'intermédiaire des connecteurs. Dans COSA, les composants et les connecteurs sont définis comme des entités de première classe et empruntent des mécanismes opérationnels issus des systèmes à objets. Cette approche distingue explicitement la notion de calcul de la notion de communication. Elle sépare l'interface de connecteur (*rôle*) de son comportement (*glu*). La glu décrit les fonctionnalités attendues, représentées par les parties en interactions. La contribution principale de cette perspective est, d'une part d'emprunter le formalisme des ADLs et de les étendre grâce aux concepts et mécanismes opérationnels objets, et d'autre part d'expliciter les connecteurs en tant qu'entités de première classe pour traiter les dépendances complexes entre les composants.

1.6 Conclusion

Pour représenter, les systèmes logiciels complexes de manière architecturale, des notations expressives s'imposent. Les notations orientées objets (UML) et les notations spécifiques des langages de descriptions d'architectures (ADLs) sont les deux approches de description des architectures logicielles des systèmes complexes, qu'on retient. Ces deux approches de description des architectures logicielles :

- L'approche orientée objet basée sur de construction logicielle depuis des entités encapsulées définissant des interfaces fournit un ensemble des services.
- L'approche orientée composant, fournit les composants et les connecteurs qui comme concepts clés de description des systèmes logiciels.

On peut conclure que la cœxistence de qualité des deux approches est sans aucun doute profitable pour la description de l'architecture logicielle. Sa modélisation nécessite absolument, l'utilisation de langages spécifiques dédiés, qui font l'objet du chapitre suivant.

Chapitre 2 : Les langages de modélisation d'architecture : concepts et outils de support

2.1 Introduction

L'architecture logicielle occupe une position centrale dans le processus de développement des systèmes complexes. Vu le rôle important que l'architecture joue dans le développement de systèmes complexes, il est devenu indispensable de disposer des méthodologies formelles ou semi-formelles et de bénéficier d'outils de support pour valider et analyser les modèles.

La description de l'architecture logicielle est fondée sur deux techniques de modélisation : la modélisation d'architecture logicielle à base de composants (composants architecturaux) décrite par les ADLs (Architecture Description Languages) (Clements et al. 2002 ; Medvidovic, Taylor, 2000) et la modélisation orientée objet utilisant le langage UML (Unified Modeling Language) (Jacobson et al. 1992 ; Booch, Rumbaugh, Jacobson, 1998 ; Object Management Group, 2001). Les deux techniques de modélisation sont dites, successivement Object-Based Software Architecture (OBSA) et Component-Based Software Architecture (CBSA) (Khammaci, Smeda, Oussalah, 2005). Ces langages de modélisation d'architectures permettent la formalisation des architectures logicielles, la réduction du coût et l'augmentation de la performance du système logiciel ainsi que la compréhension des gros logiciels en les représentants à un niveau d'abstraction élevé.

Dans ce chapitre, nous présentons les langages de description d'architectures les plus connus comme : Wright, Acme, Darwin, Archjava, Fractal et ses outils de support. On introduit ensuite, les langages de modélisation par objets les plus complets qui actuellement, sont devenus un standard de facto dans le monde industriel : UML, en

particulier avec la version UML 2.0 (Object Management Group, 2004a ; Pilone, Pitman, 2005). On termine avec le standard XMI, qui permet de représenter les modèles d'architectures sous forme XML, et qui favorisent leurs échanges et leurs pérennités.

2.2 Définitions

Nous nous intéressons dans cette section aux concepts d'architecture logicielle. L'objectif n'est pas de proposer une nouvelle définition ni de comparer les définitions proposées, mais plutôt de s'interroger sur les concepts eux-mêmes.

2.2.1 Architecture logicielle

Une architecture logicielle est définie comme un niveau de conception qui contient la description des composants à partir desquels un système est construit, les spécifications comportementales de ces composants, les modèles et les mécanismes de leurs interactions (connecteurs) et enfin un modèle définissant la topologie (configuration) d'un système. La figure 2.1, présente un métamodèle montrant les concepts de base des architectures logicielles ainsi que leurs relations.

2.2.2 Composant

Un composant est une entité de calcul ou de stockage de données spécifiant, par contrat, ses interfaces (fournies et requises). Un composant logiciel peut être déployé indépendamment et peut être sujet de composition par un tiers pour la conception d'applications logicielles (Garlan, Taylor, 1997).

"*A Software component is a unit of computation with contractually specified interfaces. A Software component can be deployed independently and is subject to composition by third parties*".

De cette définition, il résulte que :

- Un composant est une unité de composition spécifiant, par contrat, ses interfaces (fournies et requises),

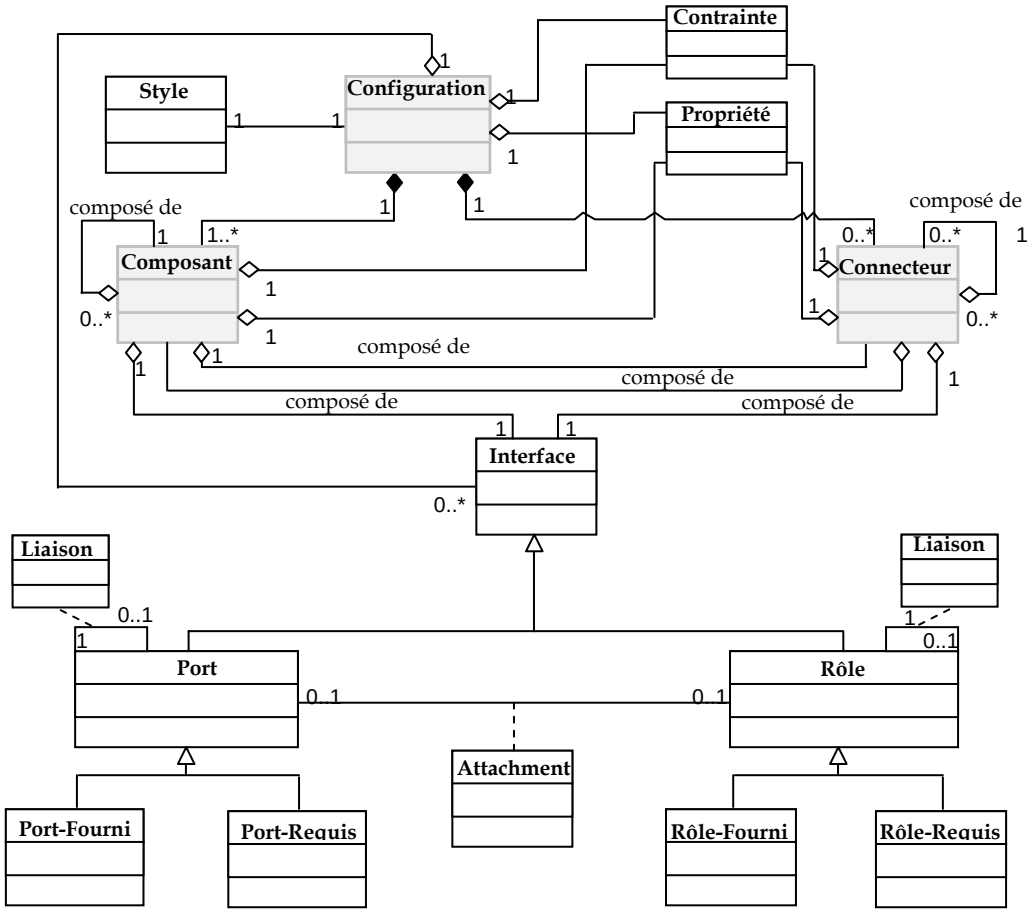


Figure 2.1. *Les concepts de base des langages de description d'architectures.*

- Un composant logiciel peut être déployé indépendamment (installation sur différentes plates-formes, collaboration et coopération avec d'autres composants),

- Un composant peut être typé, et peut avoir une sémantique formelle ou informelle. Il peut exporter des contraintes d'utilisation et présenter des propriétés fonctionnelles ou non fonctionnelles.

Un composant logiciel possède, principalement, les trois éléments suivants :

- ***L'interface* :** est la partie visible d'un composant. L'interface d'un composant consiste en un ensemble de points d'interaction entre le composant et son environnement qui permettent l'invocation des services. Dans les approches objets, les services sont spécifiés comme des méthodes, puis comme interface de communication, construite pour encapsuler ces services. Cette interface constituée d'un ensemble de ports munis d'une sensibilité, attachés à un service qui définit le comportement à réaliser lors de

son invocation. Deux types de ports peuvent être distingués : les ports *services* (*fournis*), qui exportent les services des composants et les ports *besoins* (*requis*), qui importent les services vers les composants.

- ***Les propriétés***: généralement, ce sont des attributs. Elles permettent d'adapter et d'évoluer des composants par des modifications de certaines propriétés (interface, comportement). Il existe deux types de propriétés : les propriétés fonctionnelles et les propriétés non fonctionnelles. Les propriétés fonctionnelles concernant la sémantique des fonctions des composants alors que les propriétés non fonctionnelles représentent d'autres besoins tels que la sécurité, la portabilité et la performance. Elle peut être configurable en fonction du contexte d'exécution. Les spécifications des propriétés non fonctionnelles des composants sont nécessaires pour permettre la simulation, de leur comportement, de leur analyse, de leur traçabilité depuis leur conception jusqu'à leur implémentation, et de l'aide dans la gestion de projet (par exemple, en définissent des conditions de performance rigoureuses).

- ***Les contraintes*** : propriétés spécifiques qui permettent de spécifier des restrictions sur les éléments architecturaux sur lesquels elles s'appliquent. Elles doivent être spécifiés afin de représenter les utilisations prévues d'un composant et d'établir les dépendances parmi ses éléments internes ; contraintes qui peuvent être, la sécurité, la persistance ou les transactions, etc...

Les composants sont représentés par des classes (comme le paradigme objet) qui encapsulent, les services, les contraintes sur ces services et les paramètres éventuels du composant. Ces classes définissent le type du composant, c'est-à-dire une abstraction qui permet l'encapsulation de fonctionnalités dans des blocs réutilisables. Elles permettent également l'instanciation de plusieurs composants similaires. Un type de composant peut être instancié plusieurs fois dans une même architecture ou peut être réutilisé dans d'autres architectures (Medvidovic, Taylor, 2000). Les types de composants peuvent être paramétrés, facilitant ainsi la réutilisation (Coupaye, Bruneton, Stefani, 2002 ; Garlan, Monroe, Wile, 2000). La modélisation explicite des types facilitent également la compréhensibilité et l'analyse d'une architecture, sachant que les propriétés d'un type sont partagées par toutes ses instances.

2.2.3 Connecteur

Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions (transfert de contrôle et de données) entre les composants. Ils contiennent des informations concernant les règles d'interaction entre les composants. Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants. En effet, la raison de l'existence des connecteurs est de faciliter le développement d'applications à base de composants logiciels. Les composants s'occupent du calcul et stockage tandis que les connecteurs s'occupent de gérer les interactions (communication/coordination) entre les composants.

Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure, et l'émission d'événements d'une manière directe entre des interfaces de même type. Les connecteurs peuvent également représenter des interactions complexes, comme un protocole client-serveur ou un lien SQL entre une base de données et une application (Garlan, Monroe, Wile, 2000).

De nouveaux connecteurs peuvent être spécifiés de la même façon que les composants. Par exemple, on pourrait spécifier des connecteurs pour implémenter un protocole de communication spécifique. Medvidovic (Medvidovic, Taylor, 2000) a classé les services d'interaction offerts par les connecteurs en quatre types. Chaque type de connecteur offre un ou plusieurs services d'interaction. Ces services sont les suivants :

- **Le service de communication :** un connecteur assure ce service s'il s'occupe des transmissions de données entre composants.
- **Le service de coordination :** supporte le transfert de contrôle entre composants. Les appels de fonctions sont un exemple de cette catégorie de connecteurs.
- **Le service de conversion :** convertit les interactions inter-composant si nécessaire. Il permet aux composants hétérogènes d'interagir. L'inadéquation d'interaction est un obstacle majeur dans la composition des grands systèmes. Les services de conversion permettent aux composants qui n'ont pas été spécialement conçus pour fonctionner les uns avec les autres, d'établir et de mener des interactions.
- **Le service de facilitation :** négocie et améliore l'interaction entre composants.

Les connecteurs sont des logiciels de communication capables d'adapter les besoins associés aux spécifications d'interfaces requises et fournies. De manière générale, les connecteurs dans les langages de description d'architectures peuvent être classés en trois groupes :

1. les connecteurs implicites, comme ceux par exemple de Darwin (Magee et al. 1995), MethaH (Vestal, 1996), Rapide (Luckham et al. 1995),

2. les ensembles énumérés de connecteurs prédéfinis, comme ceux par exemple d'UniCon (Shaw, DeLine, Zelesnik, 1996), C2 (Medvidovic, Rosenblum, Taylor, 1999), SOFA (Plasil, Besta, Visnovsky, 1999),

3. les connecteurs dont les sémantiques sont définies par les utilisateurs, comme ceux par exemple de Wright (Allen, Douence, Garlan, 1998), Acme (Garlan, Monroe, Wile, 1997), SADL (Medvidovic, Taylor, Whitehead, 1996).

Premier groupe : dans Darwin, les connexions entre les composants sont spécifiées en termes de liaison directe (en anglais, «binding») entre les ports « Requis » et les ports « Fournis ». La sémantique de ces connecteurs est définie dans l'environnement sous-jacent et la communication entre les composants prend en compte cet environnement. D'autres langages décrivent les connecteurs implicitement comme (Luckham et al. 1995) et MethaH (Vestal, 1996). Les « bindings » de Darwin, Rapide et les connexions de MeatH sont modélisées en ligne et ne peuvent en aucun cas être renommés ou réutilisés.

Deuxième groupe **:** un connecteur dans UniCon est spécifié par un protocole (Shaw et al, 1995 ; Shaw, DeLine, Zelesnik, 1996). Un protocole est défini par un type de connecteurs ; UniCon possède sept types de connecteurs, (FileIO, Pipe, Procedure Call, RPC, Scheduler, RT, Data Access et PL Bundler), possèdent un ensemble de propriétés et de rôles typés qui servent comme points d'interaction avec les composants. Les connecteurs d'UniCon ne peuvent pas être instanciés ni peuvent évoluer et ne peuvent être composés qu'uniquement de connecteurs.

Troisième groupe : dans le langage Wright, les connecteurs, sont définis comme un ensemble de *rôles* et une *glu* (Allen, Garlan, 1997). Chaque *rôle* définit le comportement

d'un participant dans l'interaction. La *glu* définit, comment les rôles interagissent entre eux pour aboutir à une fonctionnalité bien définie. Les connecteurs de Wright sont définis par les utilisateurs et peuvent évoluer via des instanciations de leurs différents paramètres. Ils peuvent être composés d'autres connecteurs. Les concepts caractérisant ces connecteurs sont, les *interfaces*, les *types*, les *contraintes* et les *propriétés*.

L'interface d'un connecteur est un ensemble de points d'interaction entre lui-même et les composants qui lui sont attachés. Les interfaces d'un connecteur portent le nom de *rôles*. On distingue deux types: le *rôle fourni* et le *rôle requis*. L'interface du connecteur reflète le nombre et le type des rôles autorisés.

Les propriétés fonctionnelles concernent les fonctions des connecteurs. Les propriétés non fonctionnelles d'un connecteur ne sont pas forcément obtenues à partir des spécifications de sa sémantique. Elles spécifient les besoins du connecteur pour une implémentation correcte. Par exemple, elles peuvent concerner la performance, la sécurité, la simulation de leurs comportements, leur analyse et la sélection de connecteurs appropriés et leurs correspondances (Medvidovic, Taylor, 2000).

Des contraintes de connecteurs doivent être spécifiées afin d'assurer les protocoles prévus, d'établir les dépendances intra-connecteurs et de fixer les conditions d'utilisation des connecteurs. Un exemple d'une contrainte simple, est la restriction du nombre de composants qui interagissent à travers un connecteur donné.

2.2.4 Configuration

Une configuration est un graphe de composants et de connecteurs. Cette information est nécessaire pour déterminer si les composants sont bien reliés, que leurs interfaces s'accordent, que les connecteurs correspondants permettent une communication correcte et que la combinaison de leurs sémantiques aboutit au comportement désiré. Elle définit la façon dont ils sont reliés entre eux.

Le rôle clé des configurations est de faciliter la communication entre les différents intervenants dans le développement d'un système. Leur but est d'abstraire les détails des différents composants et connecteurs. Ainsi, elles décrivent le système à un haut niveau d'abstraction qui peut être potentiellement compris par des personnes de

différents niveaux d'expertise et de connaissances techniques. Une configuration permet la modélisation de systèmes importants. Il est donc indispensable d'être capable de modéliser des systèmes hétérogènes et de permettre le lien entre plusieurs langages.

Les interfaces permettent les interactions entre les configurations et permettent aux configurations de communiquer avec leurs composants. Elles sont nécessaires pour faire communiquer les éléments internes de la configuration (composants) et les composants provenant d'autres configurations. Celles-ci, sont principalement utilisées dans les couplages de sous-systèmes.

Les configurations doivent être définies comme des classes instanciables pour permettre la construction de différentes architectures d'un même système. Ainsi, on peut déployer une architecture donnée de plusieurs manières, sans réécrire le programme de configuration/déploiement.

Les contraintes qui décrivent les dépendances entre des composants et des connecteurs dans une configuration sont aussi importantes que celles spécifiées au niveau des composants et des connecteurs.

Certaines propriétés ne sont reliées ni aux connecteurs ni aux composants et doivent être exprimées au niveau des configurations. Les propriétés non fonctionnelles au niveau de configurations sont nécessaires pour choisir les composants et les connecteurs appropriés, pour réaliser des analyses, pour appliquer et faire respecter des contraintes de topologies, pour lier les composants architecturaux aux processeurs et pour aider dans la gestion des projets.

2.2.5 Styles architecturaux

Un style architectural caractérise une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques (exemple : Pipe-Filtre, Client-Serveur/RPC). Le but principal des styles architecturaux est de simplifier la conception des logiciels et leur réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système (Ratcliffe, 2005). Un style architectural fournit (Garlan, 1995) :

- un vocabulaire désignant le type des entités de base (composants et connecteurs) tels que pipe, filtre, client, serveur, évènement, processus, etc.
- des règles de configuration pour spécifier les compositions d'éléments qui sont permises ;
- une interprétation sémantique de configuration. Par exemple, un système multi-agents communiquant via le tableau noir signifie que c'est un partage de connaissances entre ces agents.
- les analyses qui peuvent être appliquées sur les systèmes construits selon un tel style. Par exemple, la vérification de la causalité dans un système distribué basée sur la communication par événements.

L'utilisation des styles architecturaux, offre plusieurs avantages :

- permet la réutilisation au niveau de la configuration, et au niveau du code (Monroe et Garlan 1996),
- permet la normalisation une famille d'architectures, ce qui améliore la compréhension de l'organisation d'un système,
- permet l'utilisation d'analyses spécifiques au style concerné (Ciancarini, Mascolo, 1996).

2.2.5.1 Langage de description d'architecture : ADL[1]

Il n'y a pas de définition officielle de ce qu'est un ADL. La définition admise est qu'un ADL spécifie les composants d'un système, leurs interfaces, les connecteurs (lieux d'interaction entre les composants), et la configuration architecturale (Garlan, Shaw, 1993 ; Luckham et al. 1995 ; Shaw et al. 1995 ; Accord 2002 ; Soucé, Duchien ; 2002 ; Khammaci, Oussalah, Smeda, 2003 ; Smeda, Oussalah, Khammaci, 2005a).

Les langages de description d'architecture (ADLs) sont des langages formels qui peuvent être utilisés pour représenter l'architecture d'un système logiciel. Comme l'architecture est devenue un thème important dans le développement de systèmes

[1] Architecture Description Language

logiciels, les méthodes pour spécifier de façon non ambiguë une architecture, deviennent indispensables.

Les modifications dans une architecture peuvent être planifiées ou non planifiées. Elles peuvent également se produire avant ou pendant la phase d'exécution. Les ADLs doivent supporter de tels changements grâce à des mécanismes opérationnels (l'instanciation, l'héritage et le sous-typage, la composition, la généricité, le raffinement et la traçabilité). En plus, les architectures sont prévues pour fournir aux développeurs des abstractions dont ils ont besoin pour faire face à la complexité et à la taille des logiciels. C'est pourquoi les ADLs doivent fournir des outils de spécification et de développement pour pouvoir prendre en compte des systèmes à grand échelle susceptibles d'évoluer. Aussi, pour améliorer l'évolutivité et le passage à l'échelle et pour augmenter la réutilisabilité et la compréhensibilité des architectures. Les mécanismes spéciaux doivent être pris en compte par les ADLs.

Un ADL doit fournir les critères suivants (Smeda, Khammaci, Oussalah, 2004b) :

- ***Syntaxe concrète :*** un ADL doit fournir une syntaxe concrète qu'un cadre conceptuel pour caractériser des architectures (Garlan, Monroe, Wile, 1997). Le cadre conceptuel reflète les caractéristiques du domaine pour lequel l'ADL est prévu. Il englobe aussi leurs sémantiques (par exemple, les CSP (Hoare, 1995)), les réseaux de Petri (Murata, 1989) et les machines à états (Villa et al. 1997)).
- ***Spécification lisible :*** un ADL doit permettre la description explicite des composants, des connecteurs, de la configuration de l'architecture et bénéficier d'outils de support pour valider et analyser les modèles. Ceci permet de déterminer si une notation particulière est un ADL ou pas. Par ailleurs, afin d'inférer la moindre information sur une architecture, il faut au minimum disposer des interfaces de composants. Sans cette information, une description architecturale n'est qu'une collection d'identificateurs interconnectés entre eux.
- ***Raffinement et traçabilité :*** un ADL doit offrir la possibilité de raffiner une configuration à chaque étape du processus de développement. La traçabilité permet de garder la trace des changements successifs entre les différents niveaux d'abstraction.

Notons que le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les ADLs actuels.

- ***Passage à l'échelle :*** les architectures sont prévues pour décrire des systèmes logiciels à grande échelle qui peuvent évoluer dans le temps. Un ADL doit permettre de réaliser des applications complexes et dynamiques, dont la taille peut devenir importante.
- ***Evolution :*** un ADL doit permettre l'évolution de la configuration pour qu'elle puisse prendre en compte de nouvelles fonctionnalités. Cela se traduit essentiellement par la possibilité d'ajouter, de retirer ou de remplacer des composants ou des connecteurs.
- ***Réutilisation :*** l'héritage et le sous-typage sont deux manières différentes de réutiliser des modèles (de composant, de connecteurs ou de configurations) alors que l'héritage permet la réutilisation du modèle lui-même. Le sous-typage permet la réutilisation des constituants d'un modèle. L'héritage dans les ADLs peut également être multiple, où un sous-modèle hérite de plusieurs modèles. Le sous-typage peut être défini par la règle suivante : un type x est le sous type y si les valeurs du type x peuvent être utilisées dans n'importe quel contexte où le type y est prévu sans erreurs.
- ***Séparation des préoccupations :*** un ADL doit offrir des moyens de séparation des préoccupations des aspects d'architecture et des aspects d'implémentation. Cela permet de les réutiliser de façon indépendante.
- ***Séparation des niveaux d'abstractions :*** un ADL doit séparer clairement les aspects architecture et application. Les composants, les connecteurs et les configurations sont des types qui doivent être instanciés plusieurs fois dans une architecture. Chaque instance peut correspondre à une implémentation différente et doivent inclure la structure définie par son type.
- ***Composition hiérarchique :*** un ADL doit supporter le fait qu'une architecture entière peut être représentée comme un seul composant dans une autre architecture plus large. Ainsi, il est crucial qu'un ADL supporte la propriété de composition hiérarchique dans laquelle un composant primitif est une unité non décomposable et un composant composite est composé de composants (composites ou primitifs). La prise en compte de la composition ou de la composition hiérarchique, est cruciale. Cependant, ce mécanisme exige que les composants aient des interfaces bien définies puisque leurs

implémentations sont cachées. Comme nous le verrons dans les sections suivantes, plusieurs ADLs utilisent ce mécanisme pour définir des configurations, où les systèmes sont définis comme des composants composites faits de composants et de connecteurs.

- ***Généricité :*** la généricité se rapporte à la capacité de paramétrer des types. L'instanciation ne fournit pas de services pour paramétrer des types. Souvent des structures communes dans les descriptions de systèmes complexes sont amenées à être spécifiées à plusieurs reprises. Bien que l'héritage et la composition permettent de réutiliser le code source en fournissant au compilateur la manière de substituer le nom des types dans le corps d'une classe. Ceci, aide à la conception et à l'utilisation de bibliothèques de composants et de connecteurs. Ces dernières constituent d'importants outils pour le développement rapide et efficace du logiciel à base de composants.
- ***Contraintes :*** les contraintes qui décrivent les dépendances entre les composants et les connecteurs dans une configuration sont aussi importantes que celles spécifiées dans les composants et les connecteurs eux-mêmes et viennent les compléter. Le concepteur spécifie ces contraintes, ce qui revient à définir des contraintes globales, c'est-à-dire des contraintes qui s'appliquent à tous les éléments d'une application.
- ***Propriétés non fonctionnelles* :** les propriétés non fonctionnelles qui ne concernent ni les connecteurs ni les composants doivent être spécifiées au niveau de la configuration. Par conséquent, un ADL doit pouvoir définir les contraintes liées à l'environnement d'exécution au niveau de la configuration.

L'ensemble de ces critères permet de faire une comparaison entre les différents ADLs, et de choisir l'ADL adéquat relativement à la nature de l'application et à la satisfaction des besoins recherchés.

2.3 Les langages de description des architectures à base de composants

2.3.1 Description des principaux ADLs

Les ADLs trouvent leurs racines dans les langages d'interconnexion de modules (Module Interconnexion Languages) des années 70 (DeRemer, Kron, 1976). Ils sont aujourd'hui dans une base de maturité. Parmi les représentants actuels, nous citons :

Darwin (Magee et al. 1995 ; Magee, Karmer, 1996), MethaH (Vestal, 1996), Aesop (Garlan, Allen, Ockerbloom, 1994), C2 (Medvidovic, Taylor, Whitehead, 1996 ; Medvidovic, Rosenblum, Taylor, 1999), Rapide (Luckham et al. 1995), UniCon (Shaw et al., 1995 ; Shaw, DeLine, Zelesnik, 1996), Wright (Allen, Garlan, 1994 ; Allen, Garlan, 1997, Allen , Douence, Garlan, 1998), Acme (Garlan, Monroe, Wile, 1997 ; Garlan, Monroe, Wile, 2000), Olan (Balter et al. 1998), Fractal (Bruneton, 2004 ; Coupaye, Bruneton, Stefani, 2002), SOFA (Plasil, Balek, Janecek, 1998 ; Plasil, Besta, Visnovsky, 1999), ArchWare (π-ADL) (Oquendo et al. 2002 ; Oquendo, 2004) et ArchJava (Aldrich, Chambers, Notkin, 2001 ; Aldrich, Chambers, Notkin, 2002 ; ArchJava, 2004).

Différents langages de description d'architecture logicielle existent. Même s'ils possèdent parfois des caractéristiques communes, ils diffèrent sur d'autres relatives aux composants, à la composition de composants, à leurs cycles de vie, etc. Certains modèles sont hiérarchiques car ils intègrent la notion de composant composite, c'est-à-dire un composant qui contient des sous-composants. Les modèles non hiérarchiques sont dits plats car ils n'impliquent que des composants primitifs. Les ADLs permettent la description et la visualisation des éléments architecturaux, l'analyse d'architectures, la documentation et la génération automatique de codes et offrent des outils spécifiques. Dans ce contexte les ADLs étudiés sont :

- Darwin (Magee et al. 1995 ; Magee, Karmer, 1996) : il supporte l'analyse des systèmes de transmission de message distribués.

- C2 (Medvidovic, Taylor, Whitehead, 1996 ; Medvidovic, Rosenblum, Taylor, 1999) : est un style architectural basé sur les notions de composants et de messages. Le style C2 peut être résumé comme un réseau de composants concourants reliés par un ensemble des connecteurs. La partie haute (top) d'un composant peut être reliée à la partie basse (bottom) d'un connecteur et la partie basse d'un composant peut être reliée à la partie haute d'un connecteur. Il n'y a pas de limites sur le nombre de composants liés à un connecteur.

- UniCon (Shaw et al., 1995 ; Shaw, DeLine, Zelesnik, 1996) : les langages avec des connecteurs prédéfinis sont: Pipe, File IO, ProcedureCall, RmoteProcedureCall, DataAccess, RTScheduler, et PLBundler.

- Rapide (Luckham et al. 1995) : permet d'exprimer la dynamique d'une application de manière précise et détaillée. Son but est de vérifier la validité des architectures par des techniques de simulation de l'exécution.

- Wright (Allen, Garlan, 1994 ; Allen, Garlan, 1997, Allen, Douence, Garlan, 1998) : il fournit un modèle pour les architectures logicielles, il est largement utilisé pour l'analyse des protocoles d'interaction entre les composants architecturaux.

- ACME (Garlan, Monroe, Wile, 1997 ; Garlan, Monroe, Wile, 2000) : est un langage spécifique d'échange qui permet de décrire des structures architecturales à travers les ADLs. ACME, est le fruit d'un effort sérieux prenant en considération tous les éléments et les notations nécessaires pour modéliser des architectures. Il décrit des composants, des connecteurs, des ports, des rôles, des représentations. ACME supporte aussi la définition des styles et permet d'assurer le respect des contraintes de conception à l'aide de ses outils.

- SOFA (Plasil, Balek, Janecek, 1998 ; Plasil, Besta, Visnovsky, 1999) est un projet visant à fournir une plate-forme pour les composants logiciels. Dans le modèle de composants de SOFA, les applications sont définies comme une hiérarchie de composants.

- ArchJava (Aldrich, Chambers, Notkin, 2001; Aldrich, Chambers, Notkin, 2002 ; Aldrich et al. 2003) pour les langages combinant la modélisation et la programmation.

- ArchWare-ADL (Oquendo et al. 2002 ; Oquendo, 2004): ArchWare-ADL fournit les éléments de base pour décrire les architectures logicielles dynamiques. ArchWare, spécialise le π-calcul pour permettre la description de comportements dynamiques.

- Fractal (Bruneton, 2004 ; Coupaye, Bruneton, Stefani, 2002) est un modèle de composants développé par France Télécom R&D et l'INRIA. Contrairement à d'autres modèles comme les EJB ou CCM dont les composants sont plutôt de grain moyen et destinés aux applications de gestion tournées vers l'Internet, la granularité des composants Fractal est quelconque. Leurs caractéristiques font qu'ils conviennent aussi bien à des composants de bas niveau (par exemple un pool d'objets) que de haut niveau (par exemple une IHM complète). Le but de Fractal est de développer et de gérer des systèmes complexes comme les systèmes distribués. Fractal est composé de deux

modèles : un modèle abstrait et un modèle d'implantation. Fractal fournit un langage de description d'architecture (Fractal ADL) dont la caractéristique principale est d'être extensible. La motivation pour une telle extensibilité est double. D'une part, le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (LoggerController) soit ajouté à un composant, il est nécessaire que l'ADL puisse être facilement étendu pour prendre en compte ce nouveau contrôleur. C'est-à-dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (exemple : debug, warning, error). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL : déploiement, vérification, analyse, etc. Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage.

2.3.2 Outils de support des ADLs

L'utilité d'un ADL est directement liée aux outils et aux environnements permettant la description, l'analyse et l'exploitation d'architectures. Les outils de support sont principalement des :

– éditeurs graphiques et textuels des architectures,
– outils de vérification structurelle,
– outils d'analyse comportementale, utilisés pour l'exécution d'architectures (simulation et supervision),
– outils de production automatique des interfaces graphiques,
– outils de raffinement et de génération de code exécutable,
– outils permettant l'évolution d'architectures,
– outils de transformation de descriptions architecturales entre différents ADLs.

La plupart des ADLs décrits dans la section précédente possèdent des outils ou des environnements de développement permettant de définir et d'exploiter des descriptions architecturales (ADL Toolkit, 2004).

- UNICON, possède un compilateur et un générateur de code pour les architectures écrites en UNICON.

- C2, possède un environnement de développement contenant un:

  - éditeur graphique (ArchStudio) qui permet la spécification, la visualisation, et l'évolution dynamique des architectures logicielles selon le style C2.
  - vérificateur des contraintes des évolutions dynamiques des systèmes logiciels décrits selon le style C2.

- RAPIDE possède :

  - DoMe : éditeur graphique et visuel d'architectures selon RAPIDE,
  - générateur de code gérant les processus temps réel, la communication, et la synchronisation des ressources,
  - Analyseur de fiabilité et l'analyse de sûreté des systèmes embarqués,

- ACME qui possède plusieurs outils et bibliothèques, parmi ceux-ci :

  - AcmeLib : une bibliothèque disponible en Java et en C++ de classes réutilisables,
  - AcmeStudio : un environnement de développement incluant un éditeur graphique des styles et un vérificateur de règles,
  - un outil de génération de documentation au format HTML,
  - ACME PowerPoint Analysis Package : un outil de manipulation, d'analyse et de génération des architectures avec PowerPoint.
  - ACME PowerPoint Dynamic Analysis Package : simulation et supervision pour représenter et manipuler les architectures ACME,
  - ACME PowerPoint Dynamic Analysis Package : permettant la supervision et la simulation d'architectures décrites en « Dynamic ACME ».

- ArchWare qui possède un environnement de développement contenant un :

  - éditeur graphique basé sur l'approche profil UML qui permet la définition et la manipulation des architectures,
  - compilateur et un moteur d'exécution d'architectures,
  - simulateur graphique,
  - outil de vérification des propriétés structurelles et dynamiques,
  - outils de raffinement et de génération de code exécutable,
  - outil de vérification des architectures basées styles architecturaux.

- Fractal ADL fournit une palette d'outils :

  - Fractal API : ensemble d'interfaces JAVA pour l'introspection et la reconfiguration dynamique de composants et d'assemblage de composants,
  - Farclet : outil d'annotations des composants et des interfaces java et de génération de code associé aux annotations.
  - FractalGUI: outil de conception d'architectures Fractal et de génération de code Fractal 2.0,
  - Fractal pour Eclipse : environnement de développement d'applications Fractal,

2.4 Les langages de modélisation des architectures à base d'objet

De nombreuses méthodes objets ont été définies, mais aucune n'a pu s'imposer en raison du manque de standardisation. A partir de 1997, l'OMG a adopté UML (Unified Modeling Language) comme unique langage de modélisation objet (Object Management Group, 1997).

2.4.1 Description des principaux méthodes objets

Durant les années 70 et 80, de nombreux langages de spécification d'architectures logicielles orientés objet ont été définis. Pendant la première moitié des années 90, on en recensait une cinquantaine. Afin d'enrayer la multiplication des langages de spécification, G. Booch et J. Rumbaugh, ont décidé en 1994 d'unifier leurs méthodes respectives OOD de Booch (Object-Oriented Analysis and Design with Application) (Booch, 1994) et OMT de Rumbaugh (Object Management Technique) (Rumbaugh et al. 1991) au sein de la société Rationnal Software Corporation.

La première version de ces travaux est sortie en octobre 1995 sous le nom *Unified Method 0.8*. G. Booch et J. Rumbaugh n'ont eu de cesse d'unifier sous une même méthode toutes les méthodologies existantes. Jacobson a rejoint cette initiative vers la fin de l'année 1995 avec la méthode OOSE (Object Oriented Software Engineering). Les autres méthodes de première génération ont ensuite été unifiées avec UML (Unified Modeling Language). Après 1995, un RFP (Request For Proposal) a été émis à l'OMG en 1996 pour la standardisation d'UML. De nombreuses entreprises, telles que HP, Oracle,

MicroSoft et IBM, ont soumis leurs réponses à cette RFP, et UML 1.0 est sorti vers la fin de l'année 1996.

Des débuts de l'initiative UML jusqu'à la version 1.4, élaborée par l'OMG, l'objectif fondamental était de résoudre les problèmes de l'hétérogénéité des spécifications architecturales indépendamment des plates-formes. Actuellement, une nouvelle version d'UML est l'UML 2.0 (Object Mangement Group, 2004a), devenue un standard de facto dans le monde industriel. Ce standard est un langage de spécification, de visualisation, de construction et de documentation des systèmes logiciels. La grande nouveauté d'UML 2.0 est le support du paradigme composant, qui permet de définir des patrons sur les composants. Il supporte les plates-formes à composants EJB (Entreprise Java Bean) et CCM (Component CORBA Model), la définition des profils pour les architectures logicielles et la définition d'une DTD selon le standard XMI. UML devient donc potentiellement un langage de modélisation permettant l'élaboration des architectures logicielles indépendamment et spécifiques des plates-formes.

2.4.2 UML 2.0

UML 2.0 a été standardisé en juin 2003 et fournit quelques améliorations au-dessus d'UML 1.x. Sa sémantique est plus précise que celle d'UML 1.4, même si elle demeure informelle. La notation graphique est intégralement alignée sur le métamodèle. Le standard UML 2.0 est composé de deux standards, UML 2.0 Superstructure, qui définit la version utilisateur et UML 2.0 Infrastructure, qui spécifie l'architecture de méta-méta-modélisation d'UML ainsi que son alignement avec MOF 2.0 (Méta Object Facility).

Dans cette section, nous présentons la version 2.0 d'UML, un standard orienté architecture tout en explicitant ses mécanismes d'extension, suivie d'une section de présentation du standard OCL 2.0 (Object Constraint Langauge). Une autre section est consacrée aux modèles d'architectures XML.

2.4.2.1 UML 2.0 orienté architecture

UML 2.0 est un standard orienté architecture, ce que n'était pas le cas d'UML 1.4. L'UML 2.0 permet d'élaborer des architectures logicielles à base d'objets et de composants. UML 2.0 est plus ouvert, ce qui permet l'intégration des nouveaux

concepts et styles architecturaux. Les objectifs les plus importants d'UML 2.0, qu'il se doit d'atteindre, dans le contexte de l'architecture logicielle sont les suivants :

- établir clairement une séparation entre la sémantique des architectures logicielles et la notation semi-formelle en établissant un mapping bidirectionnel entre eux.
- définir une DTD pour UML 2.0, selon le standard XMI. XMI est considéré comme le seul standard capable d'assurer les échanges de modèles d'architectures.
- supporter le paradigme composant et permettre de définir des patrons pour les connecteurs (Alti, Djoudi, 2009).
- permettre la définition des profils pour les architectures logicielles.
- supporter les plates-formes à composants CCM (CORBA Component Model) et EJB (Entreprise Java Bean).

2.4.2.2 Les améliorations dans UML 2.0

Les principales améliorations d'UML 2.0 sont :

- la création des nouveaux concepts pour décrire la structure architecturale interne des classes, des composants et des collaborations au moyen des parts, connecteurs et ports.
- une meilleure encapsulation des composants par des ports complexes avec le protocole de machine à états qui peuvent « commander » l'interaction avec l'environnement.
- le pouvoir expressif concernant les aspects de spécification, de réalisation et de câblage des composants.
- l'enrichissement des interactions avec une meilleure architecture et des concepts de commandes tels que la composition, les références, les exceptions, les boucles et une vue d'ensemble améliorée avec des diagrammes de vue d'ensemble d'interaction (*Interaction Overview Diagrams*).

2.4.2.3 Architecture et niveaux

UML 2.0 est défini selon l'architecture à 4 couches de méta-modélisation de l'OMG (méta-métamodèle, métamodèle, modèle, les objets de l'utilisateur) (Object Mangement

Group, 2004a). Chaque couche définit des modèles pour spécifier les modèles de la couche inférieure. La figure 2.2, illustre cette architecture MOF 2.0 au niveau des méta-métamodèles (niveau M_3) qui contient les concepts de base de définition des métamodèles. UML 2.0 est définit au niveau des métamodèles (niveau M_2), et les modèles UML (niveau M_1) contiennent tous les concepts nécessaires à la construction et à l'évolution des systèmes logiciels (niveau M_0). Les relations existantes entre les niveaux M_1-M_2 et M_2-M_3 sont équivalentes. M_2 définit la structure de M_1 et M_3 définit la structure de M_3. Le haut niveau de MOF définit sa propre structure. UML 2.0 considère que le niveau des modèles (M_1), le niveau des métamodèles (M_2) et le niveau des méta-métamodèles (M_3) sont tous des modèles.

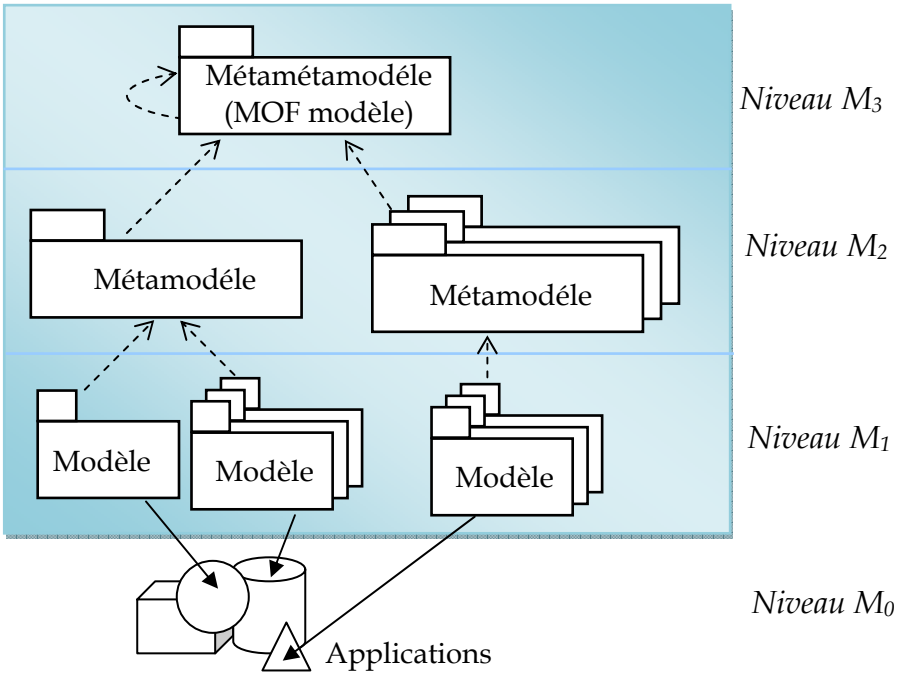


Figure 2.2. *Architecture à quatre niveaux en UML 2.0.*

2.4.2.4 Le paradigme composant

La grande nouveauté d'UML 2.0, est le support du paradigme composant. En plus des concepts proposés dans UML 1.4, la nouvelle version UML 2.0, propose de nouveaux concepts et raffine plusieurs autres concepts déjà existants. UML 2.0 dispose d'une capacité descriptive importante pour représenter les éléments architecturaux. Cette version d'UML propose cinq nouveaux éléments qui sont : l'interface, les ports, le classifieur structuré, le composant et le connecteur (figure 2.3).

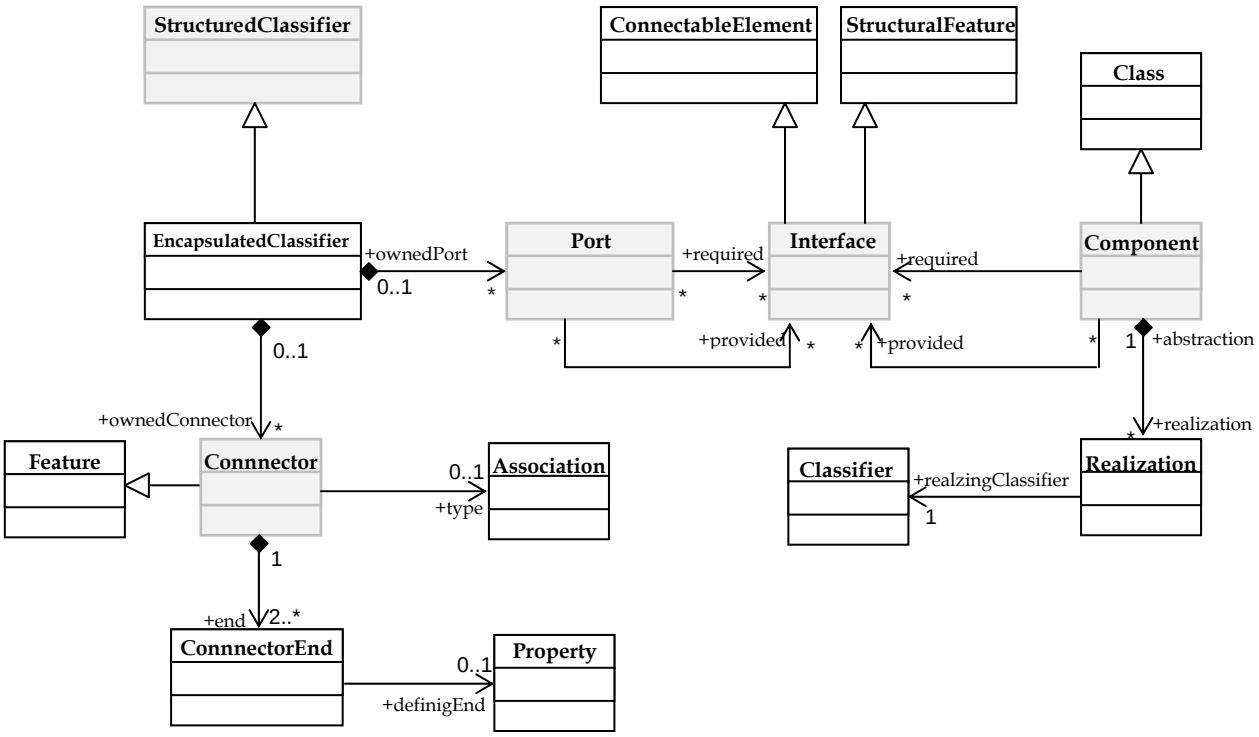


Figure 2.3. *Les éléments architecturaux dans le métamodèle UML 2.0.*

- ***Interface***

UML 2.0 étend le concept d'interface d'UML 1.4 pour inclure explicitement des interfaces de type « *Fournis* » et des interfaces de type « *Requis* ». Une interface peut inclure des attributs et augmenter sa description comportementale (appelée machines d'états). Il existe trois manières différentes représentant les mêmes interfaces dans UML 2.0. La première utilise un stéréotype classifieur avec les compartiments. La deuxième, représente chaque interface qui s'identifie à l'utilisation, aux besoins requis ou fournis dans les listes des interfaces. Chacune des interfaces est précédée par un stéréotype approprié « *ProvidedInterface* » ou « *RequiredInterface* ». La troisième manière présente les interfaces dans une forme concise et utile pour connecter les classes.

- ***Composant***

Un composant est une entité modulaire d'une application, qui peut être facilement remplacé par un autre composant sans que cela ait impact sur son environnement. La notion d'interface de composant devient très importante. Un composant a des interfaces offertes et des interfaces requises. D'ailleurs, c'est ce qui caractérise son environnement. Actuellement, un composant est une sous-classe de la classe du métamodèle UML 2.0. Un composant est aussi expressif que les classes, et il a en plus de la classe la capacité de

posséder plus de types d'éléments (contraintes, cas d'utilisation, objets) et la spécification de déploiement.

- ***Classifieurs structurés (Parts)***

UML 2.0 fournit une nouvelle façon de représenter la structure interne de classifieur. Un classifieur structuré est défini globalement ou partiellement, en terme de nombre d'instances que possède un classifieur structuré. Ces instances sont appelées des parties. Les parties sont créées avec la création de classifieur et sont détruites en même temps que le classifieur.

- ***Connecteur***

Le connecteur qui est un nouveau concept dans UML 2.0 représente le lien entre deux ou plusieurs éléments connectables (ports/interfaces). Il est pourvu d'opérations et d'attributs. UML 2.0 spécifie deux types de connecteurs : *assemblage* et *délégation*. Un connecteur d'assemblage est un lien entre une interface « Fourni » qui fournit des services à une interface de type « Requis ». Un connecteur de délégation lie le comportement externe d'un composant (port) à une réalisation interne de ce composant.

- ***Port***

C'est un nouveau concept dans UML 2.0. Il est défini comme un point d'interaction distinct de son classifieur. Le port peut avoir des types et un classifieur peut spécifier la multiplicité d'un port. Quand le classifieur est instancié, le nombre de ports est créé. Des classifieurs de même type peuvent instancier différents ports de même type. Chaque port peut s'associer à plusieurs interfaces « fournis » et/ou « requis ». Comme les interfaces, les ports peuvent être associés avec les descripteurs de type machine à états, cela permet de définir les contraintes d'usage. Les ports peuvent être simples (sans interfaces associées) ou associés à des interfaces (fournis ou requis).

2.4.2.5 Les profils UML (*UML Profile*)

UML permet de modéliser des architectures logicielles à base d'objets dont les concepts sont suffisamment génériques pour être utilisés dans le contexte des architectures logicielles à base de composants. En effet, utiliser UML pour décrire les

architectures logicielles, permet à ses utilisateurs de bénéficier des analyses puissantes offertes par les ADLs et aux utilisateurs d'ADLs pour utiliser les outils UML.

Pour permettre l'adaptation d'UML au domaine de l'architecture logicielle ou à d'autres domaines, l'OMG a standardisé le concept du *profil UML*. Un profil est un ensemble de techniques et de mécanismes d'extensions (stéréotypes, valeurs marquées, contraintes) permettant d'adapter UML à un domaine particulier (exemple de l'architecture logicielle).

- ***Utilisation de profils existants (Object Mangement Group, 2008)***

Actuellement, plusieurs profils sont standardisés par l'OMG. Quelques profils sont utilisés dans la modélisation commerciale et les autres pour une technologie spécifique. Parmi ces profiles, on peut citer: *UML Testing Profile*, *UML Profile for CORBA Specification*, *UML Profile for CORBA Component Model (CCM)*, *UML Profile for Enterprise Application Integration (EAI)*, *UML Profile for Enterprise Distributed Object Computing (EDOC)*, *UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms*, *UML Profile for Schedulability, Performance and Time*, *UML Profile for System on a Chip (SoC)*, *UML Profile for Systems Engineering (SysML).*

- ***Définition de nouveaux profils***

UML 2.0 facilite la création des nouveaux profils, la figure 2.4 illustre la partie du métamodèle UML 2.0 qui contient les métaclasses relatives aux profils, nécessaires à la définition de ce dernier. Un modèle instance, de ces métaclasses correspond à la définition d'un nouveau profil. Il n'y pas, à l'heure actuelle, de définition standard du profil UML. Ainsi, nous pouvons dire que les profils sont des moyens d'adapter d'UML à un type d'application (systèmes distribués, temps réel, etc.). **L'objectif de notre travail porte sur la définition du profil UML, dédié au domaine d'architecture logicielle.**

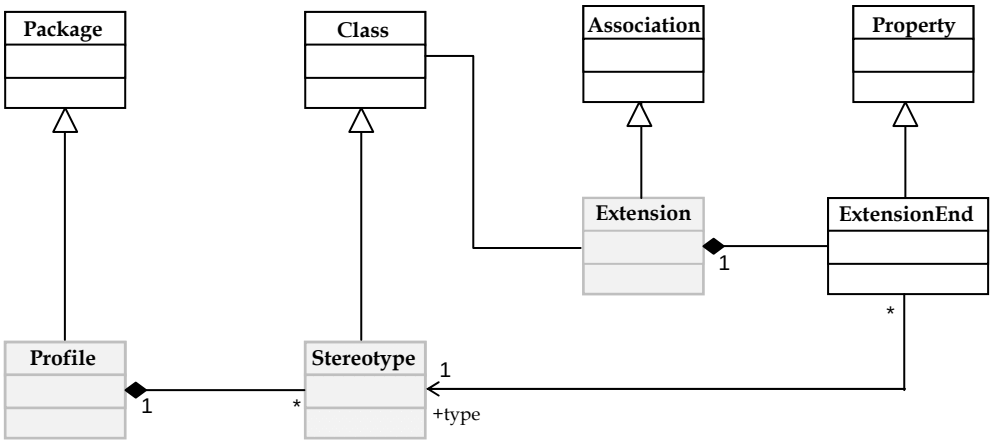


Figure 2.4. *Les profils dans UML 2.0.*

Il est communément admis qu'un nouveau profil UML est une spécification conforme à un ou à plusieurs points :

- identification d'un sous-ensemble du métamodèle UML. Ce sous-ensemble peut être le métamodèle UML entier,
- définition de règles de bonne construction (*well-formedness rules*), en plus de celles contenues dans le sous-ensemble du métamodèle UML. Le terme « well-formedness rules » est utilisé dans la spécification normée du métamodèle. De telles règles permettent de décrire un ensemble de contraintes en langage naturel, et à l'aide du langage OCL (Object Constraint Language) (Warmer, Kleppe, 1998), afin de définir correctement un élément du métamodèle,
- définition d'éléments standards (*standard elements*), en plus de ceux contenues dans le sous-ensemble du métamodèle UML. Le terme « standard element » est utilisé dans la spécification du métamodèle UML, pour décrire une instance standard d'un stéréotype UML ou une contrainte,
- définition de nouvelles sémantiques, exprimées en langage naturel, en plus de celles contenues dans le sous-ensemble du métamodèle UML,
- définition d'éléments communs du modèle, c'est à dire des instances de constructeurs UML, exprimées dans les termes du profil.

2.4.2.6 Le langage de contrainte objet : OCL[2]

Le langage OCL (Object Constraint Language) permet d'exprimer des contraintes sur tous les éléments de modèles d'architectures et des pré-conditions sur des services. Avant la version 2.0, OCL n'était spécifié qu'en langage naturel. Le concept d'expression est au cœur du langage OCL. Une expression est rattachée à un contexte, portant sur des éléments de modèles d'architectures, et peut être évaluée et vérifiée sur des restrictions relatives à ses éléments. Les expressions OCL 2.0 sont des modèles, explicitement liées aux modèles d'architectures UML.

2.4.2.7 Les modèles d'architectures en XMI[3]

Le standard XMI (XML Metadata Interchange) offre une représentation concrète des modèles d'architectures sous forme de documents XML. L'OMG utilise les mécanismes de définition de structure de balises XML DTD (Document Type Definition) et XML Schema. XMI permet de définir des structurations de balises nécessaires à la représentation des modèles au format XML. XMI s'appuie sur les documents XML et leurs structurations (modèles des instances d'une architecture) et ses documents DTD et leurs structurations (métamodèle d'une architecture). XMI est un standard assurant l'interopérabilité pour l'échange des modèles entre outils.

2.4.3 Les outils de modélisation UML

Les outils UML qui, à notre connaissance sont les plus connus dans la communauté industriel sont : Eclipse IDE, IBM Rationnel Rose, IBM Rational Software Modeler, IBM Rational Software Architect, Entreprise Architect, Poseidon pour UML, Magic Draw UML, [UMLToolKit, 2005].

- **Eclipse IDE :** est un environnement de développement universel. La spécificité d'Eclipse vient du fait de son architecture totalement développée autour de la notion de

---

[2] Object Constraint Langauge

[3] XML Metadata Interchange

plugin toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in.

- **IBM Rational Rose :** est un outil de modélisation UML. Il facilite la gestion des projets de développements. Les concepteurs de Rational Rose ont commencé à établir un « reverse engineering » à partir d'une application Java ou Delphi.
- **IBM Rational Software Modeler (RSM)** : est un outil de modélisation et de conception visuelle des modèles UML 2.0. Il offre des facilités de production sur ces modèles, la génération de code et de documentation ainsi que la définition de transformations de modèles et de générations de textes et de patterns. RSM contient Eclipse. Il propose l'approche par programmation pour réaliser les opérations de transformation et de production sur les modèles.
- **IBM Rational Software Architect (RSA)** : est un outil complet qui permet de construire visuellement des applications complexes à base des modèles. Il facilite la collaboration de plusieurs intervenants (développeurs, ingénieurs, architectes, chefs de projet) sur un même projet. RSA contient RSM.
- **Entreprise Architect :** est un outil de modélisation flexible, complet et puissant conçu pour les plateformes Windows et Unix. Il couvre toutes les étapes du cycle de development des systèmes logiciels de l'ingénierie des besoins jusqu'à la génération de code. IL gère la traçabilité entre les modèles. Il fournit une gamme impressionnante d'engineering pour les langages C#, C++, VB.NET, XML Schéma, WSDL.
- **Poseidon pour UML :** permet d'élaborer des modèles UML. Il propose l'approche par template (squelette de code pattern) à la génération de texte et de code. Les intervenants sont les ingénieurs de qualité, architectes et chefs de projet.
- **Magic Draw :** est un outil graphique de modélisation UML disposant de fonctions de travail collaboratif. Cet outil de développement facilite l'analyse et la conception de systèmes orientés objets et des bases de données. Il fournit le meilleur mécanisme d'engineering de code de l'industrie (avec un reverse complet pour les environnements J2EE, C#, C++, CORBA IDL, .NET, WSDL) ainsi que la génération de DDL.

2.5 Comparaison et synthèse

Cette section présente une comparaison des principaux ADLs suivant des critères liés à la description architecturale. Puisque à l'heure actuelle UML est considéré comme un ADL (Ivers et al. 2004), nous appliquons également cette comparaison sur UML.

2.5.1 Description d'architectures

Actuellement, il y a des ADLs qui n'ont pas de constructions explicites pour définir des configurations. Au contraire, ces ADLs définissent les architectures comme des composants composites. Concernant la définition des connecteurs, nous pouvons classifier les ADLs selon trois groupes en fonction de leur niveau de support des connecteurs, qui définissant les connecteurs implicitement, les langages utilisant un ensemble prédéfini des connecteurs, et les langages qui définissent les connecteurs explicitement. Le tableau 2.1, présente une comparaison des principaux ADLs concernant les aspects structurels des architectures logicielles.

	Composants	Connecteurs	Interface	Structure
C2	Composants	Connecteurs (groupe 2)	Interfaces de composants	Configuration
ACME	Composants	Connecteurs (groupe 3)	Interfaces de composants et de connecteurs	Composants (composites)
UniCon	Composants	Connecteurs (groupe 2)	Interfaces de composants et de connecteurs	Composants (composites)
Wright	Composants	Connecteurs (groupe 3)	Interfaces de composants et de connecteurs	Composants (composites)
ArchJava	Composants	Connecteurs (groupe1)	Interfaces de composants et de connecteurs	Composants (composites)
Archware	Composants	Connecteurs (groupe 3)	Interfaces de composants et de connecteurs	Composants (composites)
Fractal	Composants	Connecteurs (groupe1)	Interfaces de composants	Composants (composites)
Darwin	Composants	Connecteurs (groupe 1)	Interfaces de composants	Composants (composites)
UML	Composants	Connecteurs (groupe1)	Interfaces de composants	Composants (composites)

Table 2.1. *Comparaison selon les principaux concepts architecturaux.*

De l'étude comparative des principaux ADLs et du langage UML suivant des critères liés à la description architecturale, il ressort qu'UML présente des insuffisances dans la représentation explicite des concepts d'architecture logicielle (connecteurs, configuration, rôle, glu…etc.). A l'exception de ACME, Archware et Wright, la plupart des ADLs ne considèrent pas les connecteurs comme éléments de première classe et donc l'architecture n'est pas lisible (les abstractions de calculs et abstractions de communication sont mélangés). La majorité des ADLs ne considèrent pas les configurations comme éléments de première classe, donc les aspects architecture et déploiement sont mélangés.

2.5.2 Critères des ADLs

Le tableau 2.2 présente une comparaison selon les critères des ADLs cités précédemment. Nous constatons que l'ADL Wright, malgré l'avantage de vérification dynamique des applications et le mécanisme de raisonnement formel sur les architectures, ne propose pas une bibliothèque des composants. Tandis que, pour Fractal, c'est l'architecture qui n'est pas très visible. La plupart des ADLs négligent l'aspect de communication et ne proposent pas de solutions explicites pour le faire. Pour finir, UML ne considère pas les connecteurs architecturaux comme des entités de première classe, mais de simples liens entre les composants et ne peut pas spécifier directement les services de communication et de coordination.

	C2	ACME	UniCon	Wright	ArchJava	Archware	Darwin	Fractal	UML
Spécification structurelle	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
Séparation des préoccupations	Oui	Non	Oui	Oui	Non	Oui	Oui	Oui	Oui
Bibliothèques des composants	Oui	Non	Non	Non	Non	Non	Oui	Oui	Non
Communication	Connecteur prédéfini	connecteur	Players et rôles	connecteur	Liaisons d'objets	connecteur	Liens entre les ports	communication à travers les liaisons	Non explicite
Point d'accès à un composant	Interface	Interface	Interface	Port	Objet	Port	Interface	Interface	Port

Table 2.2. *Comparaison selon les critères des ADLs*

2.5.3 Le support d'évolution

La conception de l'architecture d'un système est à la définition des différents composants et connecteurs utilisés dans le système ainsi que la topologie de leur interconnexion. Comme les architectures évoluent pour refléter l'évolution des systèmes logiciels (Rosenblum, Medvedovic, 1997), les éléments dans une architecture peuvent également évoluer.

Les supports d'évolutions qui permettent la prise en compte de l'évolution d'un système sont des aspects cruciaux pour les ADLs. L'héritage, le sous-typage, la composition, la généricité et le raffinement sont les mécanismes le plus représentatifs. Malgré leur importance, la plupart des ADLs étudiés ne supportent qu'un nombre restreint de ces mécanismes, comme l'illustre le tableau 2.3.

	Héritage	Sous-typage	Généricité	Composition	Raffinement
C2	x	√	x	x	√
ACME	√	√	√	√	x
UniCon	x	x	x	√	√
Wright	x	√	√	√	x
ArchJava	√	x	√	√	x
Fractal	√	x	x	√	x
Darwin	x	x	√	√	√
UML	√	x	√	√	√

√ pris en compte x non pris en compte

Table 2.3. *Comparaison selon les supports d'évolution.*

2.5.4 Qualités et caractéristiques d'un outil de support

Nous pensons que les qualités et les caractéristiques qui doivent être raisonnablement satisfaites en définissant un nouvel outil de composants sont:

1. La simplicité de modélisation : exprime la facilité de supporter la conception graphique des applications selon un modèle de composants donné.
2. La génération des applications : exprime le degré d'automatisme de génération des instances d'architecture.

3. La compatibilité, exprime la comptabilité avec des outils d'architectures industriels (Eclipse IDE, Magic Draw, Rational Modeler).

4. La génération de code : exprime le degré de génération de code (partielle, complet).

5. La traçabilité: exprime la distance entre l'architecture et l'implémentation, qui permet entre autres d'établir une relation entre une architecture et sa méta-architecture, entre un composant et son type et entre un connecteur et son type.

Le tableau 2.4 montre une comparaison des outils d'ADLs selon les critères précédents.

	Interface graphique	Génération des instances	Génération de code	Traçabilité	Comptabilité
C2	√	manuelle	√	syntaxique	X
Fractal	√	conteneurs	√	syntaxique	√
ACMEStudio	√	automatique	√	syntaxique	√
ArchJava	√	manuelle	√	syntaxique	x
ArchStudio	√	manuelle	√	syntaxique	√
Eclipse IDE	√	manuelle	√	x	√

√ pris en compte x non pris en compte

Table 2.4. *Comparaison selon les critères des qualités d'un outil de support.*

De l'étude comparative des outils d'ADLs et UML suivant des critères liés à la séparation des préoccupations (architecture et application) et la traçabilité sémantique, on note que quelques ADLs ont été émergé dans le monde industriel (Bass, Clements, Kazman, 1998 ; Van Ommering & al. 2000). Nous avons identifié les limites communes, en particulier dans le contexte de la traçabilité sémantique entre une architecture et ses méta-architectures, et entre une architecture et ses implémentations. La plupart des architectes conçoivent ces systèmes en utilisant des concepts de haut niveaux (composants, connecteurs, interfaces, configurations), les développeurs implémente leurs logiciels avec des concepts de bas niveaux (classes, références). Donc, ils manquent d'une stratégie pour générer automatiquement des instances d'architectures et manquent également d'une méthodologie pour conserver les traçabilités sémantiques entre une architecture et sa méta-architecture, entre un composant et son type et entre un connecteur et son type. On note que :

- Peu d'ADLs offrent des bons outils de support qui permet d'intégrer certain mécanismes opérationnels (comme l'héritage, l'instanciation, composition, etc.) et

certain notions et sémantique d'ADLs (comme les connecteurs, les configurations, les architectures, etc.) dans les techniques de modélisation architecturales proposées par l'OMG notamment dans UML 2.0 et MOF. Quand l'outil et l'environnement existent, ils sont rarement compatibles avec des outils d'architectures industriels (Eclipse, Visual Studio, Magic Draw, Rational Modeler, Rational Rose).

- Un ADL doit avoir une interface graphique. C'est l'aspect clé des partenaires industriels.
- Un ADL doit conserver la relation sémantique entre la conception et l'implémentation. Les aspects architecture et déploiement sont mélangés.
- Les praticiens recherchent des passerelles qui permettent d'établir des modèles de correspondance entre les espaces des modèles d'architectures (ADLs) et les espaces des modèles technologiques (MOF). Le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les outils actuels.

2.6 Conclusion

L'architecture logicielle occupe une position centrale dans le processus de développement des systèmes complexes. Les systèmes logiciels complexes nécessitent des notations expressives et bien définies pour représenter leurs architectures logicielles. Dans ce chapitre, nous avons présenté les concepts de base des langages de description d'architectures et leurs principaux outils de supports. La présentation du standard UML 2.0 souligne toute l'importance de ce langage dans la description des architectures logicielles. Ce standard décortiqué par le biais du profil, donne toute la dimension de flexibilité d'UML 2.0. Grâce aux profils, il est possible de construire un profil UML 2.0 propre au domaine de l'architecture logicielle. De cette étude comparative des principaux ADLs et du langage UML suivant des critères liés à la description architecturale, il ressort qu'UML présente des insuffisances dans la représentation explicite des concepts d'architecture logicielle (connecteur, configuration, glu, rôle, etc.). A cet effet, une approche hybride objet-composant COSA : Component Object-based Software Architecture, qui unifie l'ensemble des concepts d'ADLs afin de décrire l'architecture logicielle du système, est présentée dans le chapitre qui suit.

Chapitre 3 : COSA : Une approche hybride de description d'architecture logicielle

3.1 Introduction

L'architecture logicielle occupe une position centrale dans le processus de développement des systèmes complexes. Vu le rôle important que l'architecture joue dans le développement de systèmes complexes, il est devenu indispensable de disposer de méthodes formelles ou semi-formelles. La description de l'architecture logicielle est fondée sur deux techniques de modélisation : la modélisation d'architecture logicielle à base de composants (composants architecturaux) décrite par les ADLs (Architecture Description Languages) (Clements et al. 2002) et la modélisation orientée objet utilisant le langage UML (Unified Modeling Language). Les deux techniques de modélisation sont dites successivement Object-Based Software Architecture (OBSA) et Component-Based Software Architecture (CBSA) (Khammaci, Smeda, Oussalah, 2005 ; Alti, Khammaci, 2005 ; Alti, Djoudi, 2009 ; Alti, Boukerram, Smeda, Maillard, Oussalah, 2010). Ces langages de modélisation d'architectures permettent la formalisation des architectures logicielles, la réduction du coût et l'augmentation de la performance du système logiciel ainsi que la compréhension des gros logiciels en les représentants à un niveau d'abstraction élevé. Aussi, ces langages permettent la réutilisation en se basent sur l'établissement de bibliothèques de composants, la construction de logiciels en fournissent un modèle partiel du développement par indication des composants majeurs et de leurs connexions, l'analyse pour vérifier la conformité des contraintes imposées par un style architectural et la communication entre les différents intervenants dans le développement du logiciel.

Dans ce chapitre, nous décrivons une approche hybride, basée sur la modélisation par objets et la modélisation par composants, pour décrire des systèmes logiciels. La contribution principale de cette approche est, d'une part d'emprunter le formalisme des ADLs et de les étendre grâce aux concepts et mécanismes objets, et d'autre part décrit les systèmes en termes de classes et d'instances. Les éléments architecturaux (composants, connecteurs, configurations) sont des types qui peuvent être instanciées pour construire plusieurs architectures. Nous présentons également une architecture à trois niveaux de COSA pour guider l'architecte dans son processus de modélisation d'architectures logicielles. Enfin, ce modèle améliore la réutilisation des architectures logicielles en supportent une hiérarchie pour les trois niveaux conceptuels.

3.2 COSA[4] : une approche hybride d'architectures logicielles

COSA (Component Object-based Software Architecture) est une approche hybride basée sur la modélisation à base d'objets et la description architecturale. La modélisation à base d'objets et la description architecturale ont plusieurs points en commun. En effet les deux approches se basent sur des concepts similaires, lesquels sont l'abstraction et les interactions entre les composants. Dans la description architecturale, les composants et les connecteurs sont les principaux concepts pour décrire un système où les composants sont les abstractions de modules et les connecteurs sont des descriptions de la communication et des interactions entre ces composants. Dans les systèmes orientés objets, les classes sont des abstractions de données et les associations permettent de décrire les relations et les communications entre les classes et les objets. En terme d'architecture logicielle en général, la similarité entre les deux domaines est évidente. En terme d'intention, les deux approches ont pour but de réduire les coûts de développement d'applications et d'augmenter la production de lignes de code (Perry, Wolf, 1992 ; Shaw, Garlan, 1996) puisqu'elles permettent la réutilisation et la programmation à base d'objets et/ou composants.

[4] Component-Object based Software Architecture

A cet effet, certains travaux ont montré que ces deux approches pouvait être utilisées pour mieux décrie l'architecture d'un logiciel (Garlan, Cheng, Kompanek, 2002 ; Medvidovic et al. 2002). Khammaci et al. (Khammaci, Smeda, Oussalah 2005), ont présenté une étude sur les travaux qui permettent de faire co-exister les deux approches. (Garlan, Cheng, Kompanek, 2002), ont sélectionné des constructions UML pour représenter des éléments architecturaux. Un compromis intéressant est celui de l'utilisation de quatre stratégies pour modéliser les composants. Ainsi, UML-RT (Selic, 1999), est une variante particulière de ces stratégies, basée sur l'implantation des concepts ADL vers UML-RT (Cheng, Garlan, 2001). Dans le cadre de ce travail, Krutchen (Krutchen, 1995), présente le modèle (4+1) vues des architectures logicielles.

Dans COSA les composants, les connecteurs et les configurations sont des entités architecturaux peuvent être décrites explicitement. COSA prend en compte la plupart des mécanismes opérationnels inhérents à l'approche objet, comme l'instanciation, l'héritage, la généricité ou la composition (Oussalah, Khammaci, Smeda, 2004).

COSA (Component-Object based Software Architecture) est une approche hybride de description d'architecture basé sur la description architecturale et la modélisation par objets (Khammaci, Smeda, Oussalah, 2004 ; Alti, Khammaci, Smeda, 2007a ; Alti, Khammaci, Smeda, 2007b ; Alti & al. 2010). COSA décrit l'architecture logicielle d'un système comme une collection de composants qui interagissent entre eux par l'intermédiaire de connecteurs. Les composants et les connecteurs ont le même niveau d'abstraction et sont définis explicitement par la séparation de leurs interfaces et de leurs implémentations. COSA intègre la plupart des mécanismes opérationnels inhérents à l'approche objet comme l'instanciation, l'héritage, la généricité ou la composition (Oussalah, Khammaci, Smeda, 2004).

Les éléments de base de COSA sont : les composants, les connecteurs, les interfaces, les configurations, les contraintes et les propriétés. Pour ces éléments, on note qu'un nombre de mécanismes opérationnels, ont prouvé leur efficacité dans le paradigme orienté objet (Smeda, Khammaci, Oussalah, 2004b).

3.2.1 Le métamodèle de COSA

L'ADL COSA décrit les systèmes en terme de classes et d'instances. Les éléments architecturaux (les composants, les connecteurs et les configurations) sont des classes qui peuvent être instanciées pour construire plusieurs architectures (Smeda, Khammaci, Oussalah, 2004b). Les concepts de base de l'architecture COSA sont les composants, les connecteurs, les configurations, les interfaces, les contraintes et les propriétés (fonctionnelles et non fonctionnelles).

Le métamodèle COSA sera mis en œuvre dans l'environnement d'UML 2.0 et intégré au sein de MDA (Model Driven Architecture). UML 2.0 fournit une notation de modélisation qui associe à chacun de ces concepts du métamodèle COSA des entités syntaxiques. La démarche vise l'automatisation de la transformation de modèle et de l'architecture vers l'implémentation (Alti, Khammaci, Smeda, 2007a ; Alti, Khammaci, Smeda, 2007b ; Alti, 2008 ; Alti & al. 2010 ; Alti, Boukerram, 2010).

Le métamodèle COSA est décrit en utilisant le diagramme de classes UML. Ainsi, les concepts de COSA sont représentés sous forme des classes, leurs caractéristiques sous forme d'attributs et les différentes sont représentés par des associations. La classe abstraite « Elément architectural » regroupe toutes les informations structurales et comportementales partagées par un composant, un connecteur ou une configuration et n'a pas donc de correspondance conceptuelle dans un ADL classique ; cette classe sert uniquement comme un support de factorisation, et elle est représentée par une classe abstraite.

Les concepts de base de l'architecture logicielle COSA sont les mêmes que dans la plupart des architectures logicielles, à savoir : configurations, composants et connecteurs. La figure 3.1 décrit le métamodèle COSA. Cette figure montre entre autre, que COSA sépare la notion de calcul (*composant*) de la notion d'interaction (*connecteur*) et distingue deux types d'interfaces : l'interface d'un composant (appelé *port*) et l'interface de connecteur (appelée *rôle*) et que les interfaces fournissent des points de connexion entre les composants et les connecteurs. Nous détaillons les concepts de COSA dans les sections suivantes.

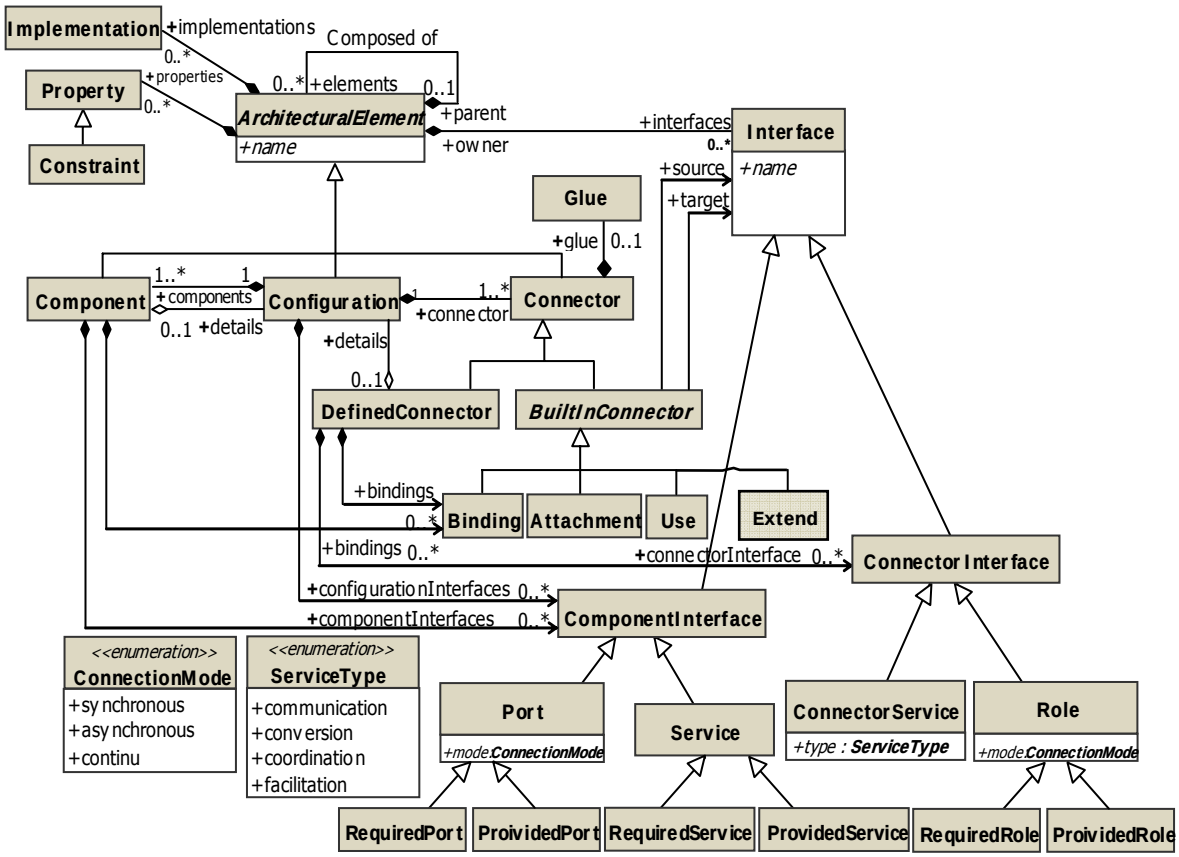


Figure 3.1. *Métamodèle de l'architecture COSA.*

COSA permet de décrire une vue de l'architecture logique afin de générer automatiquement l'architecture physique pour toutes les instances de l'application. L'idée est basée sur le raffinement et la traçabilité des éléments architecturaux. L'architecture physique est une image en mémoire de l'instance de l'architecture logique d'application. COSA permet de faire la distinction entre connecteurs et visualisation des connecteurs au niveau des modèles architecturaux. Cette distinction entre architecture physique et architecture logique permet une bonne gestion de toutes les instances de l'application.

3.2.1.1 Les configurations dans COSA

Dans COSA, les configurations sont des entités de première classe. Une configuration peut avoir zéro ou plusieurs interfaces définissant les ports et les services de la configuration. Les ports sont des points de connexion qui sont reliés aux ports des composants internes ou aux ports des autres configurations. Les services représentent les services requis et les services fournis de la configuration. En général, les

configurations sont structurées de manière hiérarchique : les composants et les connecteurs peuvent représenter des sous-configurations qui disposent de leurs propres architectures. La figure 3.2 définit les principales parties d'une configuration COSA.

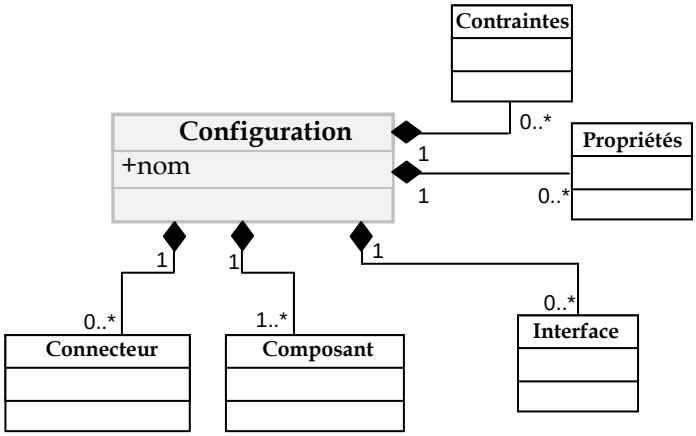


Figure 3.2. *Les configurations dans COSA*

3.2.1.2 Les composants dans COSA

Les composants représentent les éléments de calcul et de stockage de données d'un système logiciel. Chaque composant possède une ou plusieurs interfaces ayant plusieurs ports. Les ports sont les points de connexion entre les composants et le monde extérieur. Un composant est défini par un ensemble de services interagissant pour remplir un rôle et communiquant avec l'environnement via deux interfaces (requise et fournie). Le comportement d'un composant est décrit par ses services requis et fournis. Un composant peut avoir plusieurs implémentations. Un composant peut être primitive ou composite (Smeda, Khammaci, Oussalah. 2004a).La figure 3.3 décrit un composant d'une architecture COSA.

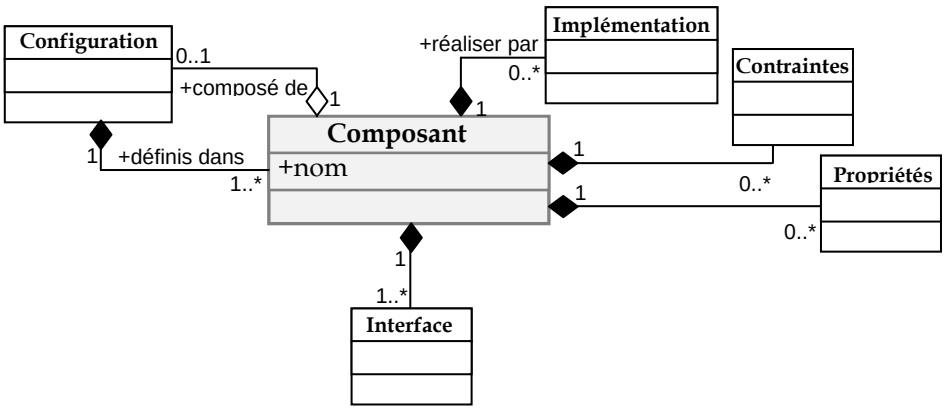


Figure 3.3. *Les composants dans COSA.*

3.2.1.3 *Les connecteurs dans COSA*

Les connecteurs entre les composants sont des entités de première classe. Dans l'approche COSA, on distingue deux types de connecteurs : les connecteurs de construction (i.e. connecteurs d'associations) et les connecteurs utilisateur. Un connecteur de construction est composé de rôles représentant les extrémités du connecteur. Un connecteur utilisateur est principalement définit par une *interface* et une *glu*, comme le montre le diagramme de classes de la figure 3.4.

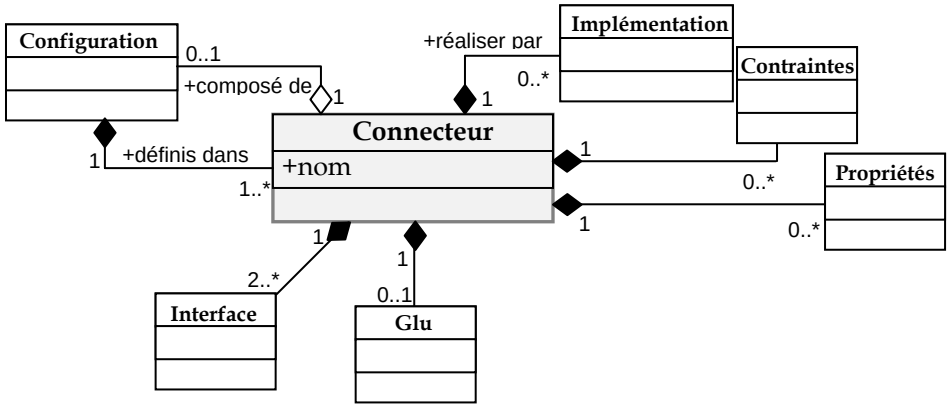


Figure 3.4. *Les connecteurs dans COSA.*

Le connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants, ceci par la définition des règles qui gouvernent ces interactions. Par exemple, un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes tels que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones.

En principe, *l'interface* décrit les informations nécessaires du connecteur, y compris le nombre de rôles, le type de services fourni par le connecteur (communication, conversion, coordination et facilitation), le mode de connexion (synchrone et asynchrone), le mode de transfert (parallèle et série), etc. Les points d'interaction d'une *interface* sont appelés *rôles*.

Un *rôle* est une interface d'un connecteur appelé à être relié à une interface d'un composant (un port de composants). Un *rôle* est soit de type « Besoin » ou de type «

Service ». En principe, un *rôle* est une interface générique d'un connecteur qui sera relié à une interface d'un composant. Un *rôle* «Service» sert comme un point d'entrée dans l'interaction d'un composant représenté par une instance d'un type de connecteur. Il est appelé à être connecté à une interface « Besoin » d'un composant (ou à un *rôle* « Besoin » d'un autre connecteur). De manière similaire, un *rôle* « Besoin » sert comme point de sortie de l'interaction d'un composant représenté par une instance d'un type de connecteur et ce *rôle* a pour but de se connecter à une interface « Service » d'un composant (ou a un rôle « Service » d'un autre connecteur).

Le nombre de *rôles* d'un connecteur représente le *degré* d'un type de connecteur. Par exemple, dans un type de connecteur Client-Serveur, la représentation de l'interaction d'appels de procédures entre les entités Client et Serveur est un connecteur de *degré* 2. Les interactions complexes, entre au moins trois composants, sont représentées par des types de connecteurs de *degré* supérieur à 2. Certains langages de description d'architectures proposent des connecteurs avec uniquement deux *rôles*, et un connecteur ne peut être relié qu'à un autre connecteur (cas des langages C2 (Medvidovic, Taylor, Whitehead, 1996)).

La glu décrit les fonctionnalités attendues d'un connecteur. Elle peut être un simple protocole reliant des rôles ou un protocole complexe ayant plusieurs opérations telles que le lien, la conversion de format de données, le transfert ou l'adaptation. En général, la glu d'un connecteur représente le type de connexions de ce connecteur. Les connecteurs peuvent avoir leur propre architecture interne qui contient des calculs et du stockage de données. Par exemple, un connecteur peut exécuter un algorithme de conversion de données d'un format à un autre. Ainsi, le service fourni par un connecteur est défini par sa glu. Les services d'un connecteur peuvent être de type communication, conversion, coordination ou facilitation.

Un connecteur composite se compose des sous-connecteurs et ses sous-composants. Ce connecteur doit être défini dans la glu ainsi que les liens entre les sous-connecteurs et les sous-composants. La glu peut aussi avoir ses propres propriétés et ses propres contraintes en plus des propriétés et des contraintes du connecteur. Les services fournis

d'un connecteur peuvent être de quatre types : communication, coordination, conversion ou facilitation (Perry 1997 ; Mehta, Medvidovic, Phadke, 2000).

3.2.1.4 *Les interfaces dans COSA*

Les interfaces sont des entités abstraites de première classe. Une interface de COSA spécifie les points de connexion et les services fournis et ceux requis pour un élément architectural (configuration, composant, connecteur). Ces derniers permettent à une interface d'interagir avec son environnement, y compris avec d'autres éléments.

De façon générale, les interfaces sont un support de description des composants permettant de spécifier comment ils peuvent être assemblés ou utilisés au sein d'une architecture. Les interfaces de composants en COSA sont vues comme les points de connexion des composants et sont le support des invocations de services.

Le point de connexion est appelé *port* pour les composants/configurations et *rôle* pour les connecteurs. En plus les ports et les rôles, les interfaces contiennent des services fournis/requis qui expriment le comportement fonctionnel de l'élément architectural (composant, connecteur et configuration). Un service peut utiliser un ou plusieurs points de connexion pour exécuter sa tâche. Du point de vue conceptuel, les ports, les rôles et les services sont des classes concrètes héritées de la classe abstraite interface comme le montre la figure 3.1. La figure 3.5 décrit les interfaces dans COSA.

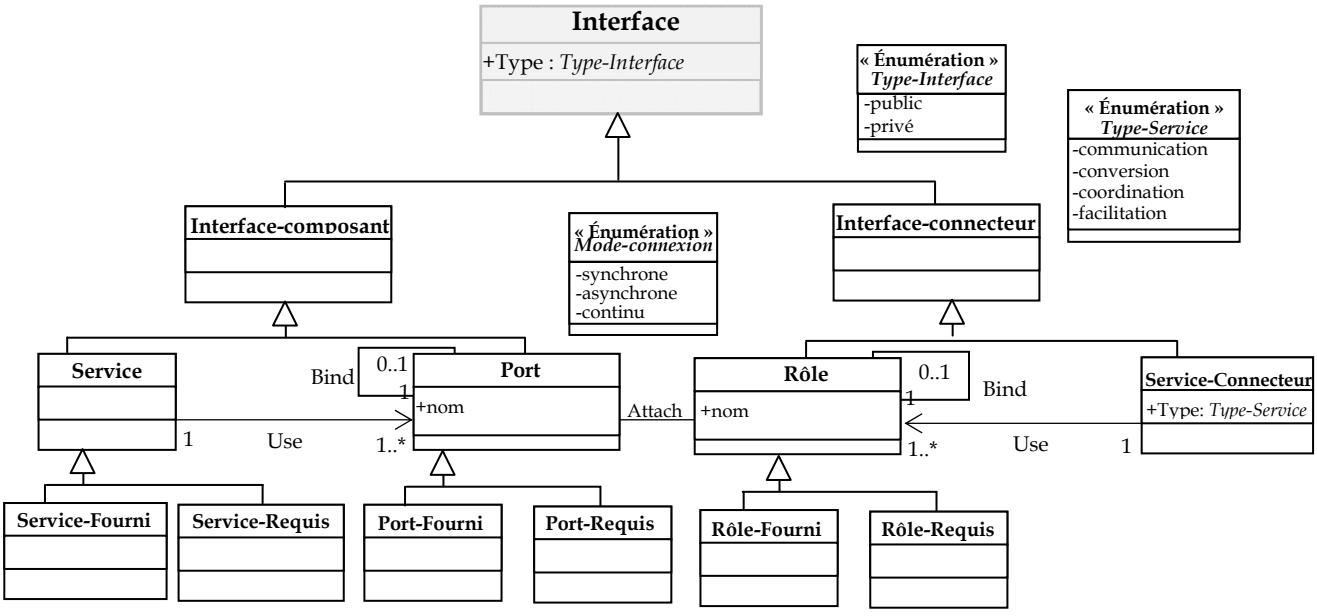


Figure 3.5. *Les interfaces dans COSA.*

3.2.1.5 Les propriétés (fonctionnelles et non fonctionnelles) dans COSA

Selon le paradigme de séparation avancée des préoccupations, un système est un ensemble de préoccupations fonctionnelles et extra-fonctionnelles. Les préoccupations fonctionnelles sont les fonctionnalités métier que le système doit assurer, alors que les préoccupations extra-fonctionnelles sont des services dont le système a besoin pour effectuer les fonctionnalités métier. Comme exemple de préoccupations extra-fonctionnelles on peut citer la sécurité, la gestion de la persistance, etc.

Les propriétés servent en général à documenter les détails des éléments architecturaux (composant, connecteur, configuration) relevant de leur conception et de leur analyse. Elles deviennent utiles lorsqu'elles sont utilisées par des outils à des fins de manipulation, d'affichage ou d'analyse.

Pour un composant (respectivement un connecteur ou une configuration), les propriétés peuvent concerner aussi bien leur structure, leur comportement que leurs fonctionnalités et permettent de caractériser les éléments architecturaux. Elles peuvent être paramétrées et configurées selon un contexte d'exécution particulier. Par ailleurs, il existe d'autres propriétés dites non-fonctionnelles qui représentent les services utilisés par un élément architectural (composant, connecteur, configuration) et qui ne font pas partie des services applicatifs. Ce sont des services de base fournis par l'architecture tels que la sécurité, le traçage et la fiabilité.

Du point de vue de COSA, les propriétés sont des valeurs non interprétées, c'est-à-dire, qu'elles n'ont aucune sémantique intrinsèque. Les propriétés deviennent utiles quand les outils les utilisent pour l'analyse, la traduction et la manipulation. Toutes les entités architecturales de COSA peuvent être annotées avec une liste des propriétés. Chaque propriété a un nom, un type facultatif et une valeur.

Dans COSA, les propriétés sont simplement représentées en tant qu'attributs. COSA ne fournit donc pas de syntaxe spécifique pour définir les propriétés, et les types de propriété sont définis en utilisant les types primitifs (entier, chaîne de caractère, booléen), et des constructeurs pour les enregistrements, les ensembles et les listes.

3.2.1.6 Les contraintes dans COSA

Un des principaux concepts d'une description architecturale est un ensemble de contraintes de conception qui déterminent comment une architecture est autorisée à évoluer dans le temps. Les contraintes incluent des restrictions aux valeurs permises des propriétés et de topologie. Par exemple, une architecture peut contraindre sa conception pour que le nombre de clients d'un serveur soit inférieur à une certaine valeur. Un autre exemple concerne le débit d'un connecteur qui ne dépasse pas une certaine valeur. Les contraintes peuvent également être considérées comme un type spécial de propriété. Mais comme en général, elles jouent un rôle clé dans la conception d'architectures à base de composants, les auteurs et développeurs de logiciels fournissent souvent une syntaxe spéciale pour les décrire. Dans COSA, elles sont définies en tant qu'attributs afin de simplifier la syntaxe de l'approche.

3.2.1.7 Les associations de COSA : Attachement, Binding, Use et Extend

- ***Attachement :*** dans COSA, la topologie d'un système est définie en énumérant un ensemble d'attachements qui lient les ports d'un composant ou d'une configuration aux rôles d'un connecteur. Dans ce cas, le port requis d'un composant (configuration) est lié au rôle fourni d'un connecteur et le port fourni d'un composant (configuration) est lié au rôle requis d'un connecteur.
- ***Bindings :*** Binding est une association entre les ports (respectivement rôles) internes et les ports (respectivement rôles) externes des composants et configurations (respectivement connecteurs).
- ***Use (utilise) :*** l'association Use, relie des services aux ports (rôles pour les connecteurs). Par exemple, un port fourni (ports fournis) d'un composant est (sont) associé(s) au service fourni (ou aux services fournis) de ce composant. Un port requis (ports requis) est (sont) associé(s) à un service requis (aux services requis).
- ***Extend (hérite) :*** cette association peut être utilisé pour définir des composants concrets (respectivement connecteurs concrets ou configurations concrètes) en tant que sous-éléments de composants abstraits (connecteurs/configurations abstraites).

3.2.2 Le métamodèle d'instance de COSA

La figure 3.6 décrit le métamodèle d'instance COSA (Alti & al. 2010). Dans le monde réel, les applications sont des instances de modèle architectural. Les éléments architecturaux COSA sont instanciés pour décrire une application complète. Les instances sont créées à partir des types qui sont définis dans le métamodèle COSA. Les éléments sont créés et assemblés sous différentes contraintes définies dans les modèles d'architecture COSA (Alti, Boukerram, Smeda, Maillard, Oussalah, 2010).

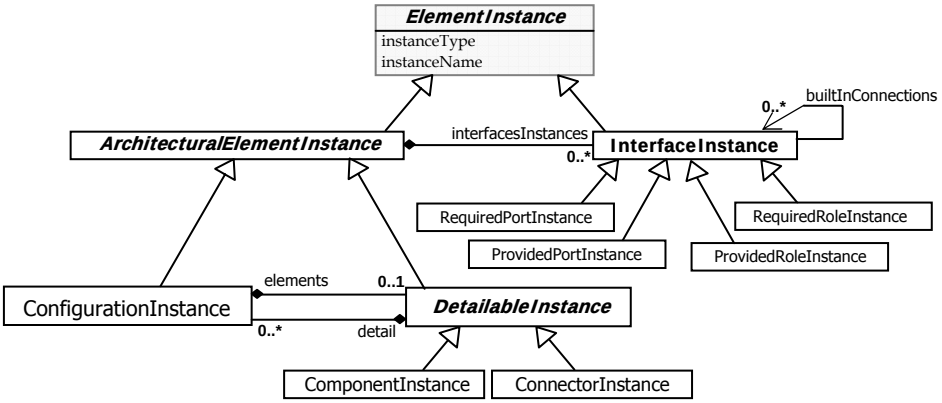


Figure 3.6. *Métamodèle instance de l'architecture COSA.*

3.2.3 Les mécanismes opérationnels de COSA

Définir les composants, les connecteurs et les configurations en tant que concepts de première classe permet de réutiliser, de les redéfinir et de les faire évoluer efficacement par des mécanismes opérationnels bien définis tels que l'instanciation, l'héritage ou la composition (Smeda, Khammaci, Oussalah, 2003). Les mécanismes que nous utilisons dans COSA sont inspirés des mécanismes du paradigme objet. Ces mécanismes sont : l'instanciation, l'héritage, la composition.

3.2.3.1 L'instanciation

L'instance d'un composant (d'un connecteur ou d'une configuration) est un objet particulier, qui est créé suivant un plan donné par sa classe génératrice. Toutes les instances d'une classe composant (connecteur ou configuration) doivent inclure la structure définie par la classe et se comporter exactement tel que cela est défini dans sa classe. L'architecture logicielle COSA distingue entre les types des composants et les types des connecteurs. Les types des composants sont des abstractions qui contiennent

des fonctionnalités de stockage de données dans des blocs réutilisables alors que les connecteurs définissent des abstractions qui englobent les mécanismes de communication, de coordination et de conversion entre les composants (Smeda, Khammaci, Oussalah, 2004a). La figure 3.7 décrit l'exemple du système Client-Serveur en COSA. Le système possède deux composants (Client et Serveur) et un connecteur (RPC). Les composants Client et Serveur qui communiquent par l'intermédiaire du connecteur RPC constituent la configuration *Client-Serveur*. Le composant *Client* possède un port de type requis (*besoin*) et le composant *serveur* possède un port fourni (*service*). Le connecteur *RPC* représente un médiateur auquel un *client* peut passer une requête et un *serveur* envoie une réponse. Il possède un rôle fourni (*particpateur-1*) et un rôle requis (*participateur-2*) et son service est de type communication.

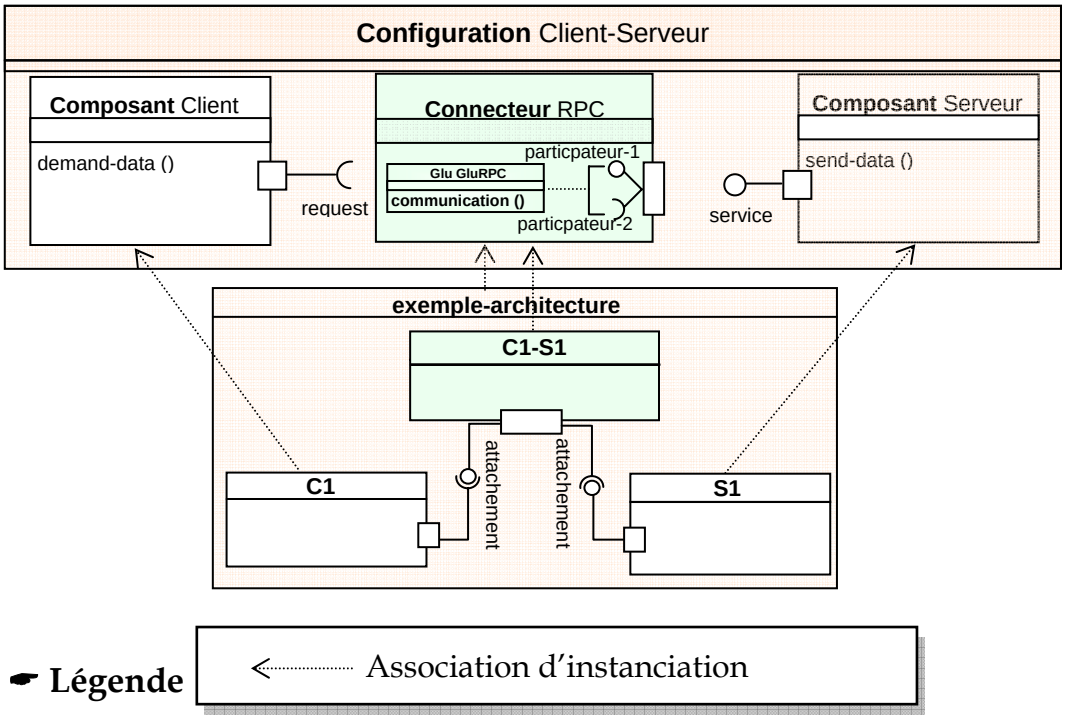


Figure 3.7. *Exemple d'instanciation dans COSA*.

3.2.3.2 L'héritage

Dans COSA, les composants (respectivement connecteurs ou configurations) peuvent être définis par extension d'autres composants (connecteurs ou configurations). Le mécanisme d'extension est semblable au mécanisme d'héritage des classes, c'est-à-dire un sous-élément peut ajouter et surcharger (override) des éléments de son super élément. Ce mécanisme peut être utilisé pour définir des composants concrets

(respectivement connecteurs concrets ou configurations concrètes) en tant que sous-éléments de composants abstraits (connecteurs abstraits ou configurations abstraites). Enfin, COSA ne supporte pas l'héritage multiple pour éviter les conflits sémantiques des sous éléments. Enfin, COSA ne supporte pas l'héritage multiple pour éviter les conflits sémantiques des sous éléments. La figure 3.8 montre la spécialisation de connecteur RPC par deux rôles caller2 et caller3. Le connecteur RPC-1 peut être utilisé pour connecter trois clients avec le serveur via le rôle request.

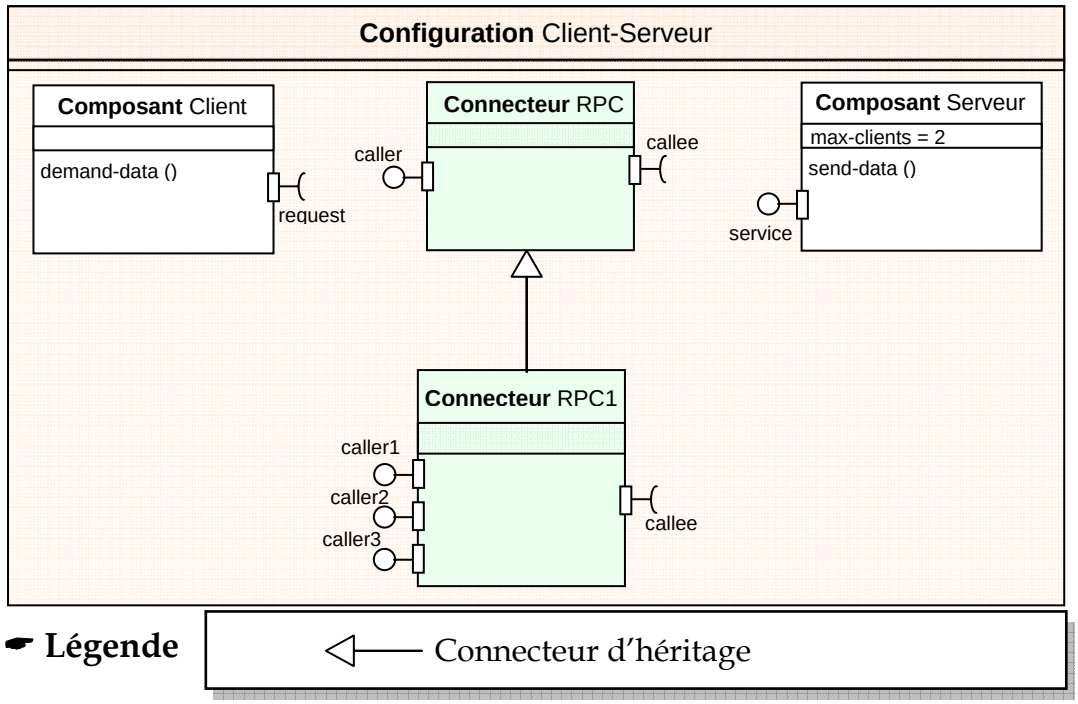


Figure 3.8. *Exemple d'héritage en COSA.*

3.2.3.3 La composition

Un composant (respectivement un connecteur) peut être composé de configurations. Il est dit *composant* (respectivement connecteur) *composite*. Dans la suite de cette thèse, on appellera composant interne (respectivement connecteur interne), les composants et les connecteurs inclus dans un composant composite (respectivement connecteur composite).

Les composants internes (respectivement connecteurs internes) peuvent être assemblés afin de remplir des services du composant composite (respectivement connecteur composite). Le composite dispose de ses propres interfaces et donc de ses propres points de connexion et utilise les liaisons (*binding*) pour faire le lien avec les points de connexion des interfaces des éléments internes. Ainsi, pour les services du composite

rendus par les composants internes, on utilise les correspondances pour indiquer quels points de connexion d'un élément interne redent les services de l'élément composite. COSA supporte la composition de ses éléments architecturaux (composants, connecteurs). En effet, avec la définition explicite de ces éléments, on peut composer non seulement les composants, mais également les connecteurs et même les configurations (une configuration est vue comme un composant composite au niveau d'abstraction supérieur). Dans COSA, les composants composites et les connecteurs composites sont définis comme des configurations. Les connecteurs composites ne possèdent pas de glu, puisque leurs services sont définis au niveau de leurs structures internes.

Par exemple, le schéma de la figure 3.10, montre la structure d'un composant composite (serveur qui possède le port provide de type fourni et le service send-data) constitué de trois composants : *ConnectionManager*, *SecurityManager* et *DataBase* et trois connecteurs : *SQLQuery*, *ClearanceRequest* et *SecurityQuery.*

- Le composant *ConnectionManager* possède deux ports fournis *externalSocket* et *DBQueryIntf* et un port requis *securtiyCheck*.
- Le composant SecurityManager possède un port fourni *securityAuthorization* et un port requis *credentialQuery.*
- Le composant DataBase possède un port fourni *securityManagement* et un port requis *queryIntf.*
- Le connecteur *SQLQuery* possède un rôle fourni *callee* et un rôle requis *caller.*
- Le connecteur *ClearanceRequest* possède un rôle fourni *requestor* et un rôle requis *grantor.*
- Le connecteur *SecurityQuery* possède un rôle fourni *securityManager* et un rôle requis *requestor.*
- Le composant Server utilise une configuration (*Database-Manager*) pour définir la composition.

La correspondance (*binding*) entre les composants internes et leur composant composite est nécessaire pour exporter les services des composants internes. La figure 3.9, montre la description de l'exemple.

3.3 L'architecture à trois niveaux de COSA

Dans cette section, nous présentons l'architecture à trois niveaux de COSA pour guider l'architecte dans son processus de modélisation d'architectures logicielles. Cette architecture a trois niveaux d'abstraction pour l'élaboration des modèles d'architecture logicielle et leurs interrelations : niveau méta-architecture, niveau architecture et niveau application. Il présente une spécification complète et structurée de l'architecture logicielle COSA. Il améliore la réutilisation des architectures logicielles en supportant une hiérarchie pour les trois niveaux conceptuels.

3.3.1 Niveaux d'abstractions pour l'élaboration des architectures logicielles

Une hiérarchie de trois niveaux d'abstraction pour l'élaboration des architectures logicielles est illustrée dans la figure 3.11. Le niveau le plus élevé (M_2) contient les différents concepts de base pour définir les architectures logicielles COSA, le second niveau (M_1) contient les modèles d'architectures et le troisième niveau (M_0) contient les instances de ces modèles. La relation entre M_2-M_1 est nécessaire pour vérifier la cohérence des modèles, alors que la relation entre les niveaux M_1-M_0 permet de générer plusieurs instances d'architectures (Alti, Khammaci, Smeda, 2006; Alti, Khammaci, Smeda, 2007a). Les trois niveaux d'abstraction de l'architecture logicielle sont inspirés de trois niveaux d'abstraction de l'OMG (MOF 2002) comme le montre le tableau 3.1.

3.3.1.1 *Le niveau méta-architecture (M_2)*

Selon OMG (Object Mangement Group, 2003), Le métamodèle définit la structure de tout modèle UML conforme à ce métamodèle. Par exemple, le métamodèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc.

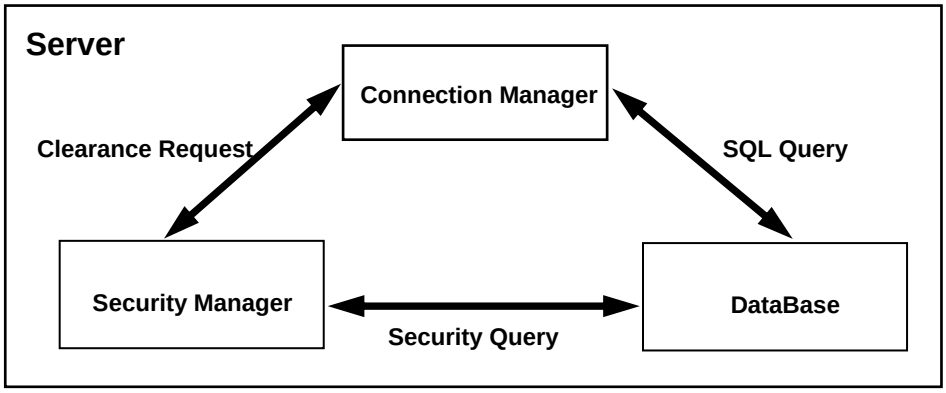
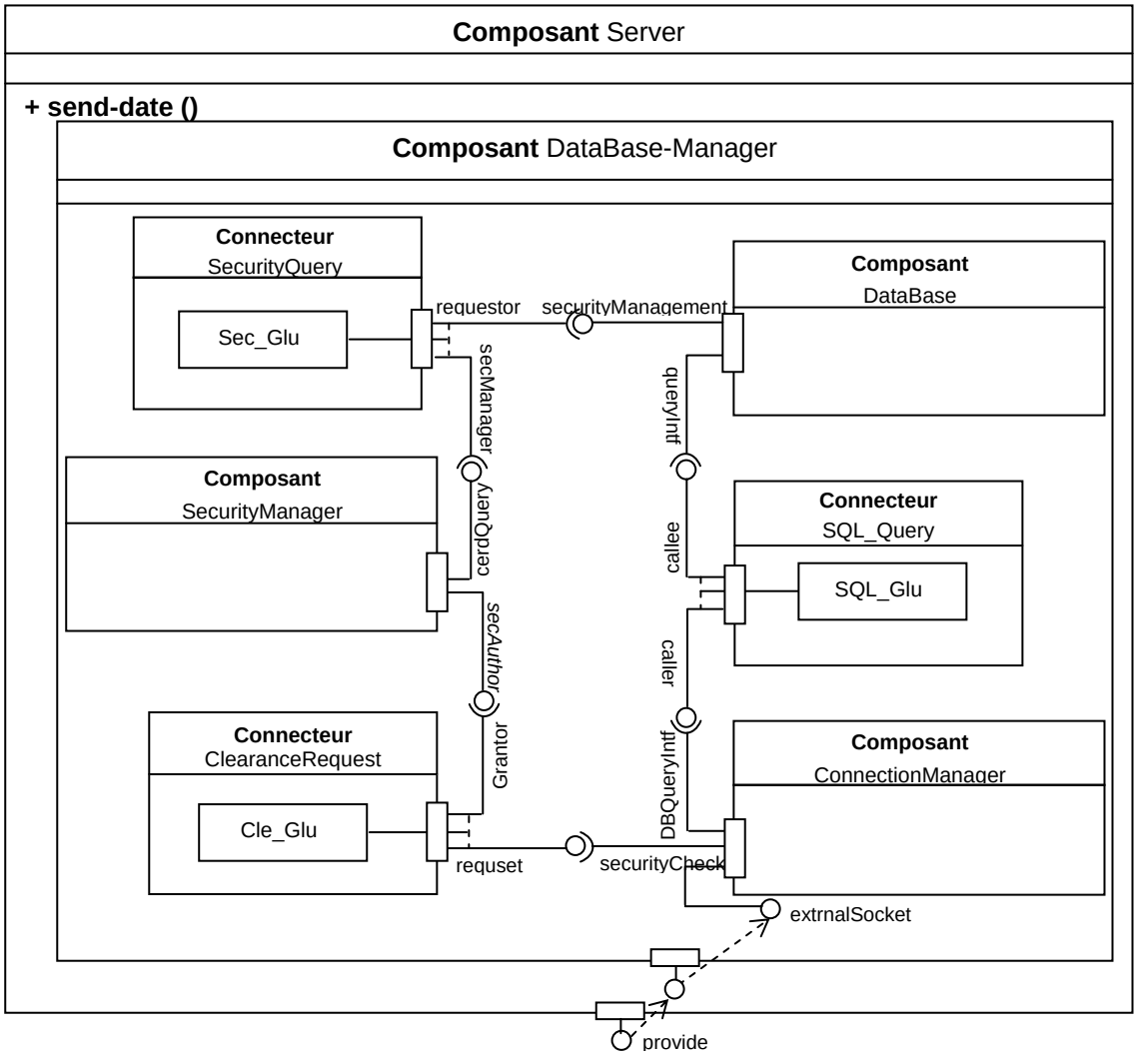


Figure 3.9. *La structure du composant Serveur.*



Légende

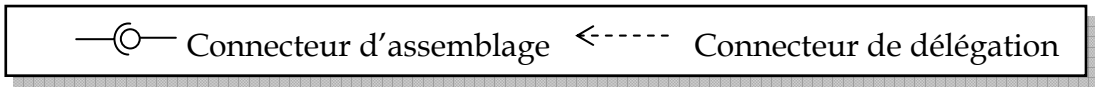


Figure 3.10. *Exemple de composition en COSA.*

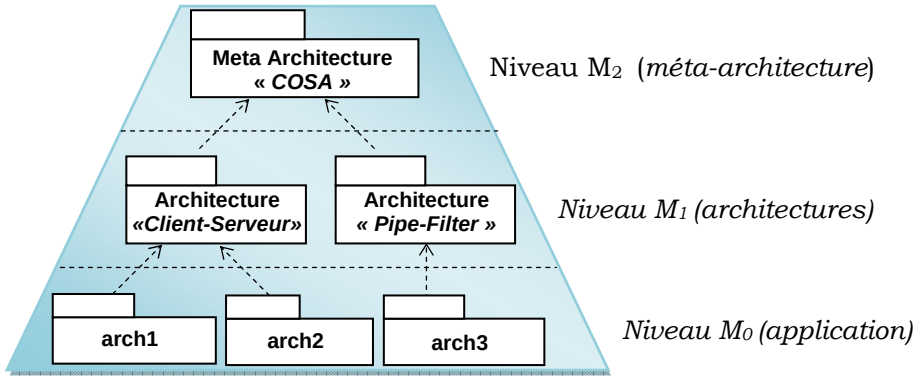


Figure 3.11. *Les trois niveaux d'abstraction de l'architecture logicielle.*

	Modélisation par objet	Modélisation par composants
Le niveau métamodèle M_2	UML	Acme, COSA, Fractal,…
Le niveau modèle M_1	Modèles	Architectures
Le niveau application M_0	Instances	Instances

Table 3.1 *Les trois niveaux conceptuels dans la modélisation objets et la modélisation composants.*

Dans le contexte de l'architecture logicielle une méta-architecture fournit la définition des concepts de base d'une architecture, ainsi que les propriétés de leurs connexions et leurs règles de cohérence. Par exemple, la méta-architecture COSA définit que les architectures COSA contient des configurations, leurs configurations des composants et des connecteurs, leurs composants des interfaces et des propriétés, etc. Une méta-architecture fournit les éléments de base pour la description d'architecture logicielle (ADL) : composant, connecteur, configuration, ports, rôles, …etc. Ces concepts de base permettent de définir différentes architectures.

3.3.1.2 Le niveau architecture (M_1)

Une architecture contient plusieurs types de composants, de connecteurs et d'architectures sont décrits. Les architectures se conforment aux méta-architectures (ADLs), donc chaque élément de M_1 associé à un élément de M_2. Par exemple dans le système Client-Serveur de la figure 3.12, client et serveur sont des composants, le RPC est un connecteur et le Client-Serveur est une configuration.

3.3.1.3 Le niveau application (M_0)

M_0 c'est le lieu où les instances d'exécution sont localisées. Une application est vue comme un ensemble d'instances de types de composants, de connecteurs et

d'architectures. Les applications sont conformes aux architectures. Chaque élément d'M_0 est associé à un élément de M_1. Par exemple, dans la figure 3.11, C1 est une instance de client, S1 est une instance de serveur, et C1-S1 est une instance de RPC et le arch1 est une instance de Client-Serveur.

3.3.2 Exemple applicatif

La figure 3.12 décrit l'architecture Client-Serveur en COSA. Les composants Client et Serveur qui communiquent par l'intermédiaire du connecteur RPC constituent le système Client-Serveur. Le connecteur RPC représente un médiateur auquel un client peut passer une requête et un serveur envoie une réponse.

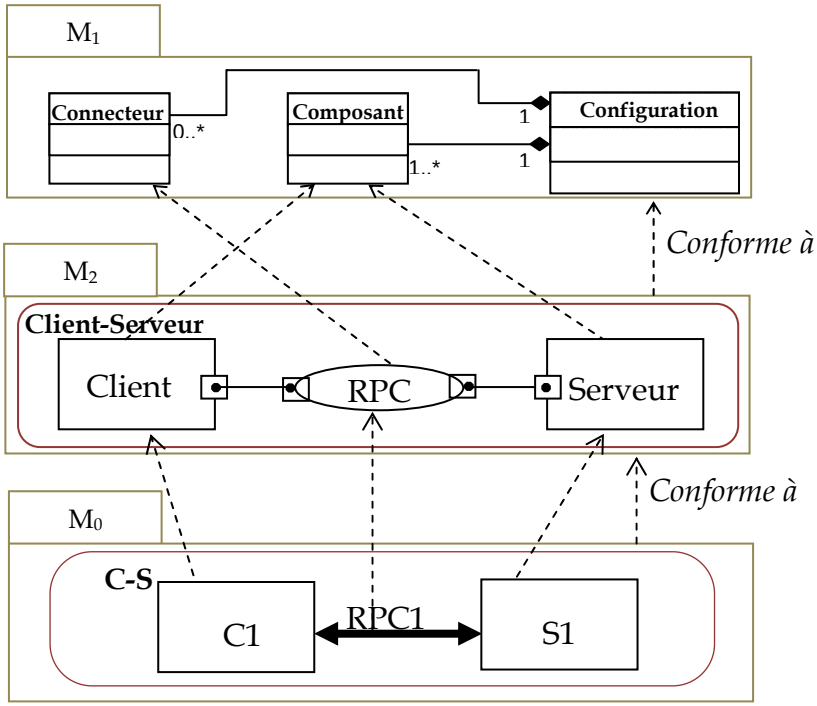


Figure 3.12 *Le système Client – Serveur dans l'architecture à trois niveaux de COSA.*

3.3.3 COSA et les langages de description d'architecture

COSA propose une présentation de l'architecture à partir d'un ensemble de composants, de connecteurs et des configurations. Un composant propose un ensemble de services à travers son interface « fournie » et demande un ensemble de services à travers son interface « requise ». Un connecteur est chargé d'assurer la communication entre les services des composants connectés et peut être utilisé pour assurer plusieurs connexions. Les mécanismes opérationnels de COSA sont inspirés des mécanismes du paradigme objet qui permettent une réutilisation croissante et un support direct des

évolutions statiques des systèmes. La prise en compte des propriétés sémantiques proposées dans le cadre de l'ADL COSA (Component-Object based Software Architecture) présente un avantage par rapport aux autres ADLs. Ainsi COSA est un ADL hybride qui réifie les concepts communément admis par la majorité des langages de description d'architectures logicielles. Dans COSA, les connecteurs sont définis comme des entités de première classe et sont utilisés pour connecter des composants, des configurations et des interfaces. COSA distingue deux catégories de connecteurs : les connecteurs utilisateur et les connecteurs de construction. Nous reprenons les critères liés à la description architecturale que nous avons appliquée aux différents ADLs (cf. chapitre 2), pour les appliquer à COSA. La comparaison selon les concepts de description d'architectures illustrée dans le tableau 3.2. Le tableau 3.3 résume la comparaison selon les critères des ADLs et le tableau 3.4 illustre la comparaison selon les supports d'évolution d'un modèle décrit par COSA.

	Composants	Connecteurs	Interface	Structure
COSA	Composants	Connecteurs (groupe 3)	Interfaces de composants et interfaces des connecteurs	Configuration

Table 3.2. *Comparaison selon les concepts de description d'architectures*

	Spécification structurelle	Séparation des préoccupations	Bibliothèques des composants	Communication	Point d'accès à un composant
COSA	Oui	Oui	Oui	Connecteur	Port

Table 3.3. *Comparaison selon les critères des ADLs.*

	Héritage	Sous-typage	Généricité	Composition	Raffinement
COSA	√	x	√	√	√

√ pris en compte x non pris en compte

Table 3.4. *Comparaison selon les supports d'évolution.*

3.3.4 Les avantages et les inconvénients de COSA

3.3.4.1 Avantages

COSA est un métamodèle générique pour la description des architectures logicielles tout en intégrant des mécanismes opérationnels inhérents à l'approche objet comme

l'instanciation, l'héritage, la composition, la généricité (Oussalah, Smeda, Khammaci, 2004). Les points forts de COSA vis-à-vis des autres ADLs sont :

1. Déploiement facile et multiforme de l'instanciation de plusieurs architectures du même système. COSA établit l'architecture par l'instanciation des composants, des connecteurs et des configurations. En premier lieu, nous commençons par instancier la configuration pour établir l'architecture. Nous pouvons avoir plusieurs instances d'une configuration, nous pouvons donc établir différentes architectures avec différentes topologies du même système. La définition des configurations comme des classes instanciables permet la construction de différentes architectures du même système.

2. Un support direct de distribution et de maintenance du système issu des connecteurs explicites. Obliger les composants de communiquer par l'intermédiaire des connecteurs donne plusieurs avantages (Smeda, Oussalah, Khammaci, 2005):

- une réutilisation croissante : le même composant peut être employé dans une variété d'environnements, chacun d'eux supportant une communication spécifique,
- l'amélioration de la maintenance du système,
- un support direct de réutilisation de manière transversale à des organisations.

En plus, plusieurs applications peuvent être décrites plus facilement en utilisant une approche dans laquelle les composants, les connecteurs et les configurations sont définis de manière uniforme et explicite.

3. COSA autorise l'évolution statique d'un système du fait l'emprunte des mécanismes opérationnels objets par COSA et ainsi l'évolution dynamique grâce aux connecteurs actifs qui sont utilisés dans le modèle d'évolution SAEV (Sadou, Oussalah Tamzalit, 2005 ; Sadou, Tamzalit, Oussalah, 2005). Le positionnement de SAEV lui permet de gérer l'évolution à la fois au niveau *Architecture*, mais aussi au niveau *Application*.

4. Dans COSA, les fonctionnalités des composants, des connecteurs et des configurations sont exprimées par les services. Ils sont représentés au niveau interface. Ceci facilite la recherche d'un composant (d'un connecteur ou d'une configuration) dans une bibliothèque donnée.

3.3.4.2 Inconvénients

1. COSA est un ADL qui évolue avec des objectifs académiques plutôt qu'avec des objectifs industriels. COSA n'est utile que si nous fournissons des outils de support.

2. COSA est un ADL semi formel et reste ainsi limité pour l'expression de la sémantique.

3. Dans COSA, les propriétés et les contraintes sont représentées en tant qu'attributs. Par conséquent, COSA manque de spécifications formelles explicites pour représenter les propriétés et les contraintes.

4. COSA ne propose pas explicitement une manière de définir des styles architecturaux. Les styles architecturaux permettent la réutilisabilité et l'intégration des architectures hétérogènes en appliquant des méthodes connues à de nouveaux problèmes (Ratcliffe, 2005).

3.4 Conclusion

Le modèle hybride COSA (Component-Object Software Architecture), permet de décrire les architectures logicielles à base de composants de systèmes informatiques. COSA est un modèle hybride basé sur l'approche objet et l'approche composant. Les composants et les connecteurs ont le même niveau d'abstraction et sont définis explicitement et il sépare la notation de calcul de la notion d'interaction. On peut conclure que les points forts de COSA par rapport aux autres ADLs sont l'évolution dynamique et statique possibles grâce aux mécanismes opérationnels objets (héritage, composition) et le support direct de distribution et de maintenance issue des connecteurs explicites. C'est ce qui permet de développer des systèmes à base de composants de meilleure qualité tout en offrant un meilleur potentiel de réutilisation. L'architecture à trois niveaux, permet de définir une spécification complète et structurée de l'architecture logicielle COSA.

Le profil UML 2.0 pour COSA et l'intégration de ce dernier dans la démarche MDA fait l'objet du chapitre qui suit.

Chapitre 4 : Intégration de l'architecture logicielle COSA au sein de la démarche MDA

4.1 Introduction

Les développements récents des systèmes logiciels sont appelés Component-Based Software Developement (CBSD). Ils sont basés sur l'assemblage des composants préfabriqués. Le CBSD permet aux développeurs de faire abstraction des détails d'implémentation et de faciliter la manipulation et la réutilisation des composants. Actuellement, il y a plusieurs plates-formes d'exécution qui se focalisent sur le développement des systèmes à base de composants. Les plus connus d'entre eux sont CORBA, EJB, .Net et les Web Services. La communication entre les composants est complexe dans des plates-formes hétérogènes et la réutilisation des composants au niveau d'implémentation est limitée.

Les langages de description des architectures logicielles (ADLs) fournissent une représentation abstraite des systèmes logiciels. Avoir une projection concrète d'une telle représentation avec une séparation de l'architecture de l'implémentation est un des principaux aspects de MDA (Model Driven Architecture). La démarche MDA (Frankel, 2003) a été proposée par l'OMG. L'idée de MDA est de proposer un modèle stable indépendant de l'intergiciel, à partir duquel il est possible de dériver différents outils. L'objectif est de capitaliser et de réutiliser ce modèle au lieu de migrer incessamment, de façon laborieuse d'un intergiciel à l'autre.

Par ailleurs, la démarche MDA est très souvent associée à la notion du profil UML. Ce dernier constitue le fondement de la démarche MDA. Les profils UML existants sont dédiés à un type d'application (système distribués, temps réel, etc.), alors que la notre

sera indépendant et spécialisé dans la modélisation des architectures logicielles.

L'intégration de l'architecture logicielle COSA au sein de MDA rentre dans le cadre des nouvelles orientations des travaux de recherche de la communauté des architectures logicielles. Nous souhaitons allier les avantages de l'approche ADL et de l'approche MDA de l'OMG. De l'approche ADL, nous intégrons les notions et les mécanismes inhérents aux composants, connecteurs et architectures qui confèrent aux modèles objets exécutables à un niveau d'abstraction haut et à un degré de réutilisation équivalent à celui d'ADL. Quant à l'approche MDA basée sur la métamodélisation, elle présente l'avantage de pouvoir considérer les modèles (architectures et méta-architectures) comme entité de première classe, et de pouvoir leur appliquer diverses opérations, en particulier, les transformations permettant l'implémentation du système sur différentes plates-formes d'exécution.

Dans ce chapitre, nous présentons une approche d'intégration de l'architecture logicielle COSA au sein de MDA par l'utilisation d'un nouveau profil UML dédiée à l'architecture logicielle COSA dont l'originalité, outre le fait qu'il traite le problème de méta-architecture peu étudié dans les ADLs. Il résiderait dans la nature de sa spécialisation. Ainsi nous définissons une stratégie de transformation directe par l'utilisation de celui-ci, pour l'élaboration des transformations PIM (Platform Indpendent Model) vers PSM (Platform Specific Model) et PSM vers PSM.

4.2 La projection de COSA en UML 2.0

Plusieurs travaux ont été réalisés sur la modélisation des concepts des ADLs en UML. Ces travaux peuvent être classés en deux catégories : des travaux visant des ADL particuliers comme par exemple ACME vers UML, Wright vers UML, et d'autres ciblant des ADLs génériques, c'est-à-dire comportant des concepts architecturaux communs à plusieurs ADLs. Dans cette section nous nous concentrons principalement sur la construction d'un profil UML 2.0 de COSA. Nous définissons un ensemble de base de stéréotypes en utilisant les possibilités du profil UML (métamodèle et modèle) pour définir une spécification complète et structurée des architectures logicielles (Alti, Khammaci, Smeda, 2006 ; Alti, Khammaci, Smeda, 2007a). En COSA, nous considérons

qu'une architecture est sémantiquement valide si ses éléments architecturaux (composants, connecteurs, configurations) sont sémantiquement valides.

4.2.1 Pourquoi la projection de COSA en UML?

Durant la dernière décade, UML est devenu un langage standard de spécification, de visualisation, de construction et de documentation des systèmes logiciels. En plus des concepts proposés dans UML 1.4, la nouvelle version UML 2.0 propose de nouveaux concepts et raffine plusieurs autres concepts déjà existants. Les nouveaux concepts et les modifications sur les concepts dans UML 2.0, fournissent un vocabulaire riche pour documenter une architecture logicielle et aider à résoudre la plupart des problèmes dus à l'utilisation des précédentes versions du langage UML. Les concepts d'UML 2.0 (Object Management Group, 2004a), sont suffisamment génériques pour être utilisés dans la description d'architecture logicielle, y compris :

1. La définition des composants comme genre de *Classifier,* ainsi les composants peuvent avoir des instances et avoir accès à certains mécanismes tels que le sous typage par la relation de généralisation, la description comportementale, la structure interne, les interfaces et les ports.
2. La redéfinition des interfaces qui peuvent inclure non seulement des interfaces *fournies* mais des interfaces *requises.*
3. L'introduction des ports comme des points d'interaction pour les *Classifiers*.
4. L'introduction des *Structured Classifiers* pour représenter la structure interne (décomposition) des *Classifiers.*
5. L'introduction des connecteurs pour représenter une liaison entre deux ou plusieurs instances. Néanmoins, les connecteurs dans UML 2.0 sont définis par une liaison simple entre deux éléments. Ils ne peuvent pas être associés à une description comportementale ou aux attributs qui caractérisent la connexion.

COSA supporte les composants composites et définit explicitement les connecteurs comme des concepts architecturaux abstraits. Donc, il est très utile de définir une projection des concepts de COSA vers UML. L'intérêt fondamental est d'enrichir UML de charges sémantiques propres à l'architecture logicielle COSA, mais aussi, de profiter de:

- la variété des avantages de COSA (cf. section 3.2.4) telle que la définition explicite des connecteurs, le support fort de la réutilisation et les vues multiples de déploiement.

- niveau d'abstraction haut et degré de réutilisation de COSA.

- sémantique riche des concepts architecturaux COSA et de capitalisation des architectures logicielles.

- résolution des problèmes d'ambiguïtés et des problèmes d'interaction des services.

Comparé à COSA, UML est une norme dominante pour l'analyse et la conception des systèmes logiciels et fournit :

- une variété d'outils qui sont mis en application pour UML, par exemple, Rational Rose, Microsoft Visual Studio, Poseidon pour UML, IBM Rational Software Modeler, Entreprise Architecte. La plupart de ces outils fournissent des services de génération de code dans différents langages tels que C++, Java, C#, etc.

- l'approche MDA (Model Driven Architecture), qui vise à fournir un cadre précis et efficace pour la production et la maintenance du logiciel (Frankel, 2003).

- la variété des avantages d'UML (cf. section 2.3.4) telles que les vues multiples, une sémantique semi-formelle et des mécanismes d'extension exprimés comme des profils UML, un langage associé puissant pour exprimer des contraintes (OCL : Object Constraint Language (Warmer, Kleppe, 1998)) et un raffinement jusqu'à l'implémentation.

Si le nombre de modèles de base (core model) disponibles est faible, il n'est cependant pas figé. Le dispositif MDA est « extensible » par le biais de nouveaux modèles, d'autant que la démarche MDA laisse en suspens de nombreux challenges. La prise en compte des applications héritées (legacy applications) de précédents développements et la gestion de l'interopérabilité entre intergiciels sont des points cités fréquemment par l'OMG au nombre des pistes à explorer.

Plus concrètement, l'approche « UML Profile » consiste à étendre le métamodèle UML en ajoutant de nouveaux concepts des ADLs. Cette opération se fonde sur l'utilisation de stéréotypes, de valeurs marquées (tagged values) et de contraintes OCL (Warmer,

Kleppe, 1998), L'avantage reconnu de cette approche est l'utilisation des outils UML. L'inconvénient majeur réside dans la difficulté à appréhender la séparation entre la méta-méta-architecture, la méta-architecture et l'architecture. Le terme « dialecte UML » est utilisé pour qualifier cette situation. A l'heure actuelle, l'approche profil UML est une approche populaire.

Actuellement, certaines recherches menées par la communauté ADL portent sur le développement de langages génériques de « seconde génération ». L'approche ADL issue des milieux académiques ne s'est pas imposée : il lui est souvent reproché d'utiliser des notations formelles difficiles à mettre en œuvre et de fournir des outils peu exploitables.

Les recherches prennent ainsi une nouvelle orientation, en direction des travaux de l'OMG. Ces deux communautés ont travaillé en parallèle ces dernières années, la pauvreté d'UML en matière de concepts architecturaux logiciels expliquant probablement cela ou du moins en partie. Aujourd'hui, l'intégration de la notion de composant dans UML 2.0, peut aussi être interprétée comme un signe de cette évolution et laisse entrevoir la possibilité de définir des architectures (à base de composants) dont le passage vers les plates-formes d'exécution adaptées sera facilité.

Nous souhaitons allier les avantages de l'approche ADL et de l'approche MDA de l'OMG. De l'approche ADL, nous retenons les notions et mécanismes inhérents aux composants, connecteurs et architectures. Quant à l'approche MDA basée sur les profils UML, elle présente l'avantage de pouvoir considérer les modèles (dans notre cas, architectures et méta-architectures) comme des entités de première classe, en particulier, des transformations permettant l'implantation du système sur différentes plates-formes d'exécution. Nous situons notre proposition dans cette dernière tendance. Concernant le point de vue technique de notre profil UML 2.0 COSA, nous avons opté pour l'approche de « UML profile » qui répond explicitement aux lacunes de l'approche MDA, au regard des langages de description d'architecture à savoir : l'absence des concepts représentant les connecteurs, l'absence des concepts représentant les configurations, l'absence des concepts représentant la glu, …etc.

4.2.2 Projection d'un ADL vers UML via les mécanismes d'extensibilités

UML est un langage générique pouvant être adapté aux langages de description des architectures logicielles grâce aux mécanismes d'extensibilité offerts par ce langage tels que stéréotypes, valeurs marquées et contraintes. Les mécanismes d'extensibilité offerts par UML permettent d'étendre UML sans modifier le métamodèle UML. Plusieurs travaux permettent d'adapter aussi bien UML 1.x que UML 2.0 aux architectures logicielles. Ces travaux peuvent être classés en deux catégories : des travaux visant un ADL particulier comme par exemple Acme vers UML, Wright vers UML, et d'autres ciblant des ADL génériques c'est-à-dire comportant des concepts architecturaux communs à plusieurs ADL.

Dans (Medvidovic et al. 2002), les auteurs ont présenté deux approches pour exprimer les éléments architecturaux avec les notations du langage UML 1.4. La première approche utilise le langage UML « tel qu'il est », alors que la seconde propose d'utiliser des extensions d'UML (stéréotypes, valeurs marquées, contraintes) pour incorporer les concepts de trois langages de description d'architectures (C2, Wright et Rapide). La projection de C2 est présentée dans le tableau 4.1 et la projection de Wright est dans le tableau 4.2.

Dans (Garlan, Cheng, Kompanek, 2002), les auteurs ont sélectionné des notations UML 1.4 pour représenter des éléments architecturaux, en prenant en considération les avantages et les limites de chaque notation et ont constaté que les aspects d'architecture logicielle à base de composants sont difficilement représentables dans UML 1.4. Ce qui nous conduisons à dire que les premières versions d'UML ne sont pas suffisamment adéquates pour représenter les concepts architecturaux tels que les composants, les connecteurs, les configurations, les interfaces (ports et rôles) ou les styles architecturaux.

UML 2.0 (Object Management Group, 2004a) a été enrichi par de nouveaux concepts architecturaux comme les connecteurs, les ports, classifieurs structuraux et a redéfini le concept de composants qui devient une sous-classe de la classe de méta-modèle UML. En outre, un composant a aussi plus de caractères expressifs que les classes (il peut avoir des interfaces et contenir d'autres composants ou classes), etc.

Dans (Goulão, Abreu, 2003), les auteurs établissent un profil UML 2.0 pour ACME. Ils regroupent des concepts communs et minimaux des ADLs. Ils considèrent aussi les connecteurs comme des composants stéréotypés sans autres interfaces, que celles qui sont définies par leurs rôles et leurs propriétés. Cependant, décrire les connecteurs comme des composants peut encombrer la conception et rendre sa structure globale difficile à comprendre et les rôles de connecteurs et les ports de composants difficiles à distinguer. De ce fait, ils ne profitent pas des nouvelles notations du langage UML 2.0. Le tableau 4.3 illustre la projection de d'ACME vers UML 2.0.

Dans (Roh, Kim, Jeon, 2004), les auteurs signalent quelques faiblesses de ce travail, liées notamment à la représentation proposée du connecteur (au sens d'ADL) en UML 2.0 et proposent un ADL générique sous forme d'un profil UML 2.0. Dans ce travail, on note notamment l'utilisation des collaborations UML 2.0 pour représenter des connecteurs ADL. De plus, un autre aspect intéressent se dégage : type et instance de connecteur sont modélisés par deux stéréotypes. En effet, le type de connecteur est défini comme stéréotype à base de métaclasse *Collaboration* d'UML 2.0. L'instance de connecteur est définie comme un stéréotype à base de métaclasse *Connector* d'UML 2.0. Mais ce travail ne traite pas les aspects comportementaux des ADLs.

Dans (Ivers et al. 2004), Ivers et al. ont étudié la convenance des nouvelles notations UML 2.0 pour la projection de la vue composants et connecteurs (C&C) de l'architecture logicielle, en particulier le langage de description d'architecture ACME. Ainsi, ils ont étudié la projection de chaque concept du langage ACME lié à la vue C&C (composant, connecteur, ports, rôles) vers UML 2.0. Ils ont choisi la correspondance sémantique, la clarté visuelle et le support par des outils comme critère de base pour choisir les notations d'UML qui peuvent représenter la description architecturale. Pour chaque notation, il a été proposé deux choix, sauf pour les attachements qui n'ont pas été considérés, comme le montre le tableau 4.4. Il a été conclu, que même si les nouvelles notations ont amélioré la description de l'architecture logicielle en utilisant UML, elles présentent toujours des inconvénients majeurs. D'ailleurs, des aspects de description architecturale continuent à être problématiques. Par exemple, UML 2.0 manque de possibilité à associer une information sémantique à un connecteur pour décrire son

comportement. Aussi, UML 2.0 ne distingue pas, les rôles qui sont les interfaces des connecteurs et les ports qui sont les interfaces des composants.

Oquendo (Oquendo, 2006) a proposé un profil UML 2.0 pour l'ADL formel ArchWare. Il fournit des notations visuelles pour modéliser formellement les architectures logicielles statiques et dynamiques en UML 2.0 tout en utilisant un outil avec une génération de code vers java (Alloui, Oquendo, 2004).

(Graiet et al. 2006) ont proposé un profil UML 2.0 pour l'ADL formel Wright. Il décrit les expressions CSP de Wright par une machine abstraite à états de description de protocole (Protocol State Machine) UML 2.0 stéréotypés. D'ailleurs, les opérations des ports et les rôles sont modélisées explicitement et sont exprimées plus formellement via une machine abstraite à états de description de protocole. Ce travail est diminué,par le manque partiel des moyens et des techniques de projection des composants et des connecteurs composites et styles d'architectures. Plus récemment, (Amirat, Oussalah, 2009) ont proposé un profil UML pour C3. Ce travail est diminué des techniques de projection des composants et des connecteurs architecturaux vers les plates-formes d'exécution objet (CORBA, EJB, etc.).

4.2.3 Profil UML 2.0 pour l'architecture logicielle COSA

4.2.3.1 Pourquoi la définition d'un profil UML 2.0 pour COSA ?

L'intérêt primordial d'un profil UML 2.0 de COSA est de fournir des moyens standards pour exprimer la sémantique de ses concepts en utilisant les nouvelles notations UML 2. En d'autre terme, l'utilisation des extensions et des standards d'UML 2.0, permet précisément de mieux capturer les concepts du modèle de COSA en fournissant un ensemble de stéréotypes. Ainsi, nous profitons des possibilités offertes par le profil UML (métamodèle et modèle) pour définir une spécification complète et structurée de l'architecture logicielle COSA. Le tableau 4.5 résume la différence entre les concepts de COSA et ceux d'UML 2.0.

C2	UML 1.4	OCL/Valeurs marqués
Component (Type)	« C2Component » (instances de métaclasse Class)	C2Component doit implémenter exactement deux interfaces.
Connector (Type)	« C2Connector » (instances de métaclasse Class) « C2AttachOverComp » (instances de métaclasse Association) « C2AttachUnderComp » instances de métaclasse Association) « C2AttachConnConn » (instances de métaclasse Association)	C2Connector doit implémenter exactement deux interfaces. C2Attachments sont des associations binaires, une extrémité d'attachement doit être un C2Compoenent et l'autre extrémité un C2Connector.
Port	« C2Interface» (instances de métaclasse Interface)	
Role	« C2Interface» (instances de métaclasse Interface)	
System	« C2Architecture» (instances de métaclasse Model) « C2Attach»	L'architecture est un réseau de concepts C2.
Message	« C2Operation» (instances de métaclasse Operation)	C2Operation sont étiquetées en tant que notifications ou demande et comme entrantes ou sortantes
Interface	« C2Interface»	

Table 4.1 *Projection de C2 vers UML 1.4.*

Wright	UML 1.4	OCL/Valeurs marqués
Component (Type)	« WrightComponent » (instances de métaclasse Class)	WrightComponent doit implémenter au moins une WrightInterface.
Connector (Type)	« WrightConnector» (instances de métaclasse Class) « WrightGlu » (instances de métaclasse Operation)	WrightConnector doit implémenter au moins une WrightInterface. WrightGlu contient un WrightStateMachine.
Port	« WrightInterface» (instances de métaclasse Interface)	
Role	« WrightInterface» (instances de métaclasse Interface)	
System	« WrightArchitecture» (instances de métaclasse Model) « WrightAttachment» (instances de métaclasse Association)	Architecture est composée des instances de composants et de connecteurs.
CSP protocol (state machine)	« WSMTransition» (instances de métaclasse Transition) « WrightStateMachine» (instances de métaclasse StateMachine)	Toutes les transitions composées dans un WrightState doivent être des WSMTransitions.
Interface	« WrightInterface»(instances de métaclasse Interface)	WrightInterface sont étiquetées en tant que ports ou rôles.

Table 4.2 *Projection de Wright vers UML 1.4.*

ACME	UML 2.0	OCL/Valeurs marqués
Component (Type)	«AcmeComponent »	Les composants ont seulement des interfaces dites des ports ou des propriétés.
Connector (Type)	«AcmeConnector»	Les connecteurs n'ont aucune autre interface que celles définies par leurs rôles.
Port	Port	Les ports peuvent seulement être utilisés avec des composants d'Acme et ils ont une interface fournie et une interface requise.
Role	«AcmeRole»	Les rôles sont liés aux connecteurs d'Acme et ils ont une interface fournie et une interface requise.
System	«AcmeSystem»	Les systèmes représentent un graphe de composants qui communiquent entre eux.
Rep-maps	Delegation connector	Tous les connecteurs de délégation relient des ports et des rôles.
Properties	«AcmeProperties»	Un port de «AcmeProperty» possède une interface fournie qui doit fournir les opérations de *get* et de *set* pour la valeur et le type de la propriété.
Properties	«AcmeConstraints»	Ce stéréotype doit avoir un attribut énuméré avec deux valeurs permises : invariable et heuristique.
Style (Family)	Package	Tous les connecteurs utilisés dans un système de pipe-filtre doivent conformes à PipeT.

Table 4.3 *Projection d'ACME vers UML 2.0.*

ACME	Choix 1	Choix 2
Component (Type)	Object (Classe)	Composant instance (Composant)
Connector (Type)	Lien Objet (Classe d'Association)	Objet (Classe)
Port	Port	Port
Role	Port	Port
Attachment	-	Connecteur d'assemblage

Table 4.4 *Projection d'ACME vers UML 2.0.*

Le langage UML 2.0 offre des moyens plus explicites pour représenter les ports, les propriétés et les contraintes, et d'autres moyens pour spécialiser et étendre les concepts UML 2.0 pour documenter des concepts architecturaux comme les composants, les connecteurs ou les configurations, en utilisant des correspondances avec les concepts d'UML 2.0. Ainsi, la définition de stéréotypes appliqués sur des métaclasses d'UML 2.0 reflète mieux la sémantique des caractéristiques de modèle COSA (Alti, Khammaci, Smeda, 2007a). Ainsi, il est nécessaire de définir un profil UML 2.0 dans le domaine des architectures logicielles.

	COSA	UML 2.0
Composant	Une composant fournit des services via des ports. Ces services doivent faire parties de son interface.	Une classe enrichie.
Connecteur	Un composant spécifique assurant des services de communication, de facilitation, de conversion des composants métier.	Une relation simple de communication.
configuration	Un graphe des composants et des connecteurs. Elle considérer comme étant un type.	Un assemblage des composants (composants composites)
Interface	Contenant les ports d'un composant/configuration ou les rôles d'un connecteur.	Permet d'assurer les méthodes requis/fourni des composants
Port	Associé à un ou plusieurs services des composants ou des configurations.	Associé à une ou plusieurs interfaces.
Rôle	Associé à un ou plusieurs services des connecteurs.	/
Service	Jouer un rôle d'un composant, d'un connecteur ou d'une configuration.	/
Glu	Assurer la communication des composants.	/
Attachement	Un rattachement physique entre un composant et un connecteur.	Une relation d'association entre deux composants.
Binding	Un rattachement physique entre un port externe (rôle externe) et un port interne (rôle interne).	Une relation de dépendance entre une interface (port) externe et une interface interne (port).
Use	Un rattachement physique entre un port/rôle et un service.	/
Extend	Un composant architectural peut être héritée d'une ou de plusieurs composants architecturaux de même type.	Une classe peut être héritée d'une ou de plusieurs classes.

Table 4.5 *COSA vs UML 2.0.*

L'intérêt primordial de définition d'un profil UML est de représenter les concepts COSA en utilisant les notations UML 2.0. Par conséquent, la représentation et la formalisation du modèle COSA, en utilisant les profils UML permettent l'intégration de l'architecture logicielle COSA au sein de la démarche MDA (Model Driven Architecture) qui unifie toutes les approches de modélisation. De même, l'utilisation des stéréotypes et tagged-values (valeurs marquées), et les contraintes permettent de mieux capturer le sémantique des concepts de COSA. Donc, nous profitons des profils UML 2.0, pour définir une spécification complète et structurée de l'architecture logicielle COSA et la réalisation de la projection des concepts de COSA vers UML 2.0 (Alti, Khammaci, Smeda, 2007a).

4.2.3.2 Définition du profil UML 2.0 pour COSA

A l'instar de la plupart des autres langages de description d'architectures (Medvidovic, Taylor, 2000), le langage COSA perd tous ses atouts dès qu'il est projeté au niveau implémentation dans la mesure où il est ne permet que de décrire un système informatique de manière abstraite. Comme le passage du langage UML2.0 vers des plates-formes objets est évident, une solution serait de réaliser un « mapping » des concepts de COSA vers les concepts d'UML2.0. L'intérêt de cette solution est de permettre la projection de l'architecture abstraite COSA vers une architecture concrète objet et de pouvoir utiliser tous les outils UML. Les nouveaux concepts et les modifications sur les concepts dans UML 2.0 fournissent un vocabulaire riche pour documenter une architecture logicielle et aider à résoudre la plupart des problèmes dus à l'utilisation des précédentes versions du langage UML. Avec la version UML 2.0, il y a plusieurs manières de représenter tout concept architectural, ce qui conduit à plusieurs stratégies de transformation qui peuvent être utilisées pour représenter les différents concepts de COSA. La projection d'une architecture abstraite telle que COSA sur une autre architecture concrète comme UML 2.0, doit suivre un processus progressif qui transforme la plupart des éléments architecturaux abstraits par de nouveaux éléments architecturaux concrets. Nous proposons de transformer chaque concept architectural dans le modèle COSA à l'aide du profil UML 2.0. En se basant sur certains critères comme la clarté, les détails à capturer et le degré de sémantique pour représenter les différents concepts de COSA, nous réalisons la projection des concepts architecturaux de COSA vers les concepts UML 2.0.

Nous avons besoin de supporter la définition explicite des connecteurs, indépendante de n'importe quel usage et de supporter, les capacités d'association des propriétés sémantiques, et de trouver une meilleure représentation pour décrire des interactions entre les sous composants à travers des sous connecteurs.

4.2.3.2.1 Les niveaux d'abstraction

Le but du profil COSA est d'étendre UML 2.0 pour représenter les concepts architecturaux COSA. Ce profil fournit une manière pratique pour intégrer l'architecture logicielle au sein de framework MDA (Modèle Driven Architecture), qui unifie toutes les approches de modélisation.

Pour définir un profil UML 2.0 de COSA, nous proposons une hiérarchie à trois niveaux d'abstraction comme le montre la figure 4.1 (Alti, Khammaci, Smeda, 2007a) :

- Le plus haut niveau (M_0) contient les différents concepts de base pour définir l'architecture COSA,
- le second niveau (M_1) contient les modèles d'architectures,
- le troisième niveau contient les instances de ces modèles (A_0).

La relation entre les niveaux M_0-M_1 est nécessaire pour vérifier la cohérence des modèles, alors que la relation entre les niveaux M_1-A_0 permet de générer plusieurs instances d'architectures.

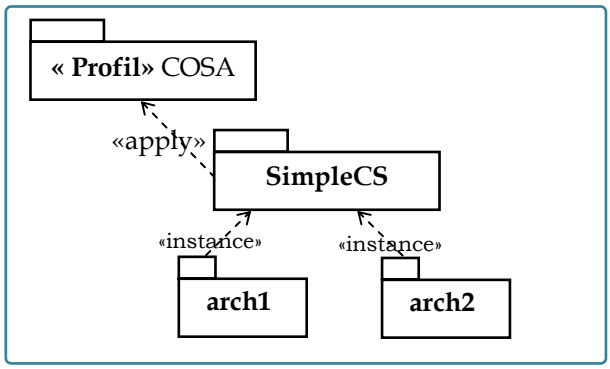


Figure 4.1 *Exemple d'instances pour le système Client-Serveur.*

4.2.3.2.2 Le niveau de métamodèle (M_0)

Le niveau le plus élevé contient les différents concepts de base pour définir l'architecture COSA Le métamodèle de COSA est décrit comme un package UML stéréotypé nommé «COSA». Ce package doit inclure un ensemble de stéréotypes : *«COSAComponent»*, *«COSAConnector»*, *«COSAConfiguration»*, etc. Ces stéréotypes correspondent aux métaclasses du métamodèle UML avec toutes ses valeurs marquées et ses contraintes OCL (Object Constraint Language) (Object Management Group, 2005). La figure 4.2 présente ce métamodèle.

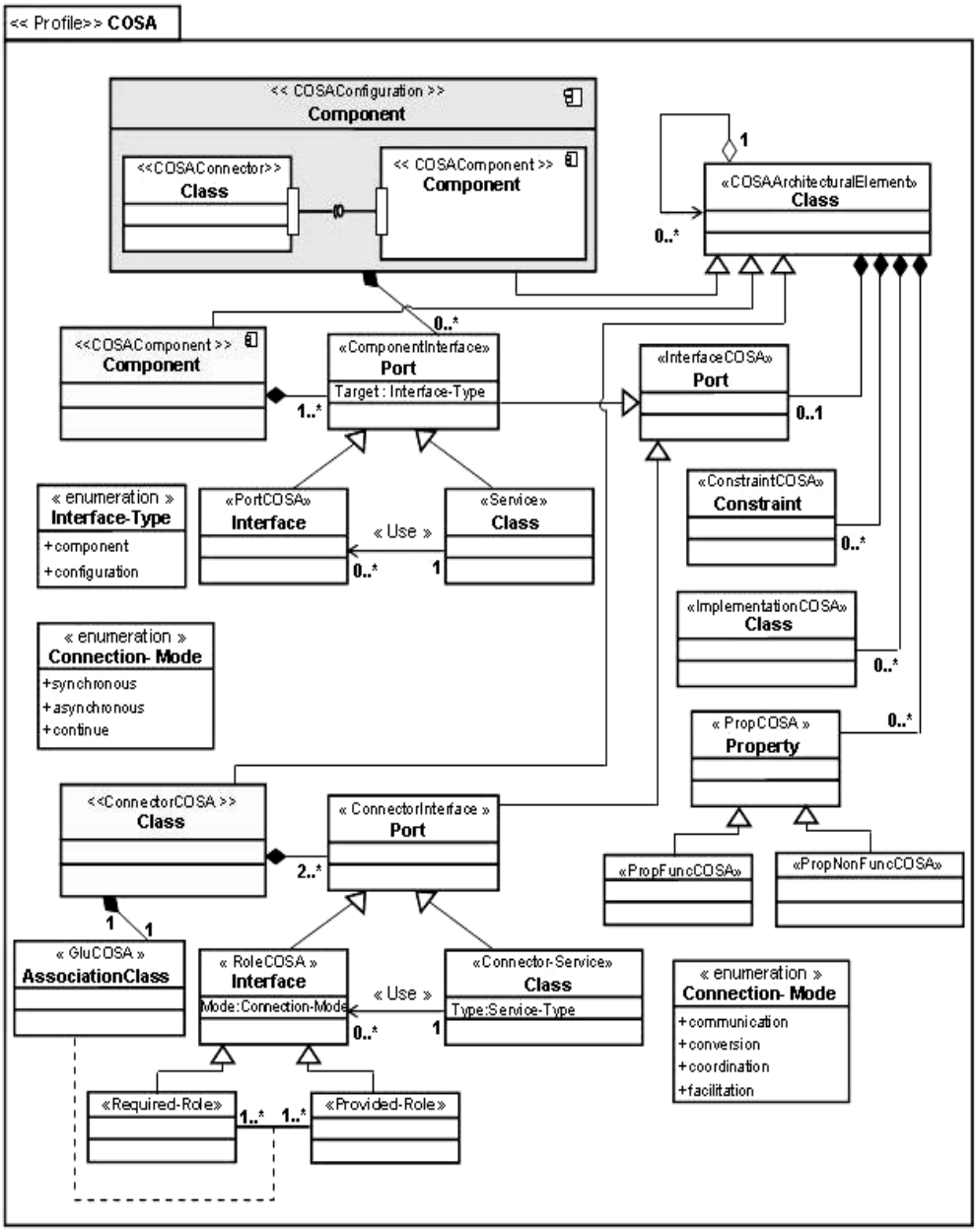


Figure 4.2 *Le profil COSA.*

☛ Critère de choix de projection

Les conventions et règles de transformation vers UML 2.0, ont été choisies en fonction des critères par Garlan (Garlan, Cheng, Kompanek 2002 ; Ivers et al. 2004 ; Medvidovic et al. 2002), pour comparer leurs différentes stratégies de représentations des concepts architecturaux en UML. Les deux approches comportent chacune des avantages et aussi des inconvénients, que nous résumons ci-dessous en fonction des quatre critères suivants :

– ***l'utilisation des nouveaux concepts architecturaux d'UML 2.0 :*** les règles de transformation doivent s'appuyer sur l'utilisation des concepts tels que les composants ou connecteurs UML nouvellement introduits dans UML 2.0, et proposés par Garlan (Ivers et al. 2004) ;

– ***la séparation visuelle entre composant et connecteur :*** la volonté de Garlan (Garlan, Cheng, Kompanek 2002), de bien séparer de composant logiciel de celui de connecteur logiciel, est une des conséquences du critère de 'lisibilité visuelle'.

– ***la limitation au domaine des composants logiciels :*** UML est découpé en diagrammes, qui proposent pour chacun d'eux, un ensemble de concepts spécifiques au domaine associé. Alors, il faut choisir un type de diagramme, et par voie de conséquence ses concepts associés. Les règles de transformation s'orientent donc vers le diagramme UML de composants, et permettent ainsi d'obtenir une meilleure compatibilité avec les outils support.

☛ **Spécification des correspondances via un profil UML**

La transformation du modèle COSA vers celui d'UML 2.0 doit suivre un processus progressif qui transforme la plupart des éléments architecturaux abstraits en de nouveaux éléments architecturaux concrets. Nous proposons de transformer chaque concept architectural du modèle COSA à l'aide de profil UML 2.0. Pour cela, les choix possibles varient suivant certains critères comme la clarté, les détails à capturer et le degré de sémantique entre les concepts des deux types d'architectures.

Le choix est porté sur le composant d'UML 2.0 pour la représentation des composants et des configurations COSA, mais ces derniers restent bien distincts grâce aux stéréotypes qu'on leur associe. Les connecteurs COSA sont des types explicites et peuvent être réutilisés, ils sont représentés par des classes stéréotypées d'UML 2.0.

Une telle transformation des concepts Composants/Connecteurs est alors très lisible, permet de capturer tous les détails des concepts de COSA et représente fidèlement la sémantique associée à chaque concept de COSA. Le tableau 4.6 résume les correspondances entre chaque concept de COSA et un concept d'UML.

COSA	UML 2.0	OCL/Valeurs marqués
Architectural-Element	«ArchitecturalElement » (instances de métaclasse Class)	Les éléments architecturaux COSA ayant seulement des propriétés, contraintes et implémentations COSA.
Component (Type)	«COSAComponent » (instances de métaclasse Component)	- COSAComponent est une spécialisation de Architectural-Element - COSAComponent doit avoir seulement des ports ComponentInterface.
ComponentInterface	<<ComponentInterface>> (instances de métaclasse Port)	Marquée par le cible : **Type-Interface**
		- ComponentInterface est portée seulement par COSAComponent ou COSAConfiguration - ComponentInterface doit avoir au moins un COSARequiredPort ou COSAProvidedPort - ComponentInterface est une spécialisation RequiredService ou ProvidedService
Port	«COSAPort» (instances de métaclasse Ineterface)	Marquée par le Mode : **Mode-Connexion**
		COSAPort est porté par ComponentInterface
Connector (Type)	«COSAConnector » (instances de métaclasse Class)	- COSAConnector est une spécialisation de Architectural-Element - COSAConnector doit avoir seulement des rôles ConnectorInterface et une seule COSAGlu.
ConnectorInterface	<<ConnectorInterface>> (instances de métaclasse Port)	- ConnectorInterface est portée seulement par les ports de ConnectorCOSA - ConnectorInterface doit avoir au moins un COSARequired-Role et COSAProvidedRole - ConnectorInterface est une spécialisation de Connector-Service
Role	«COSARole » (instances de métaclasse Ineterface)	Marquée par le Mode : **Mode-Connexion**
		COSARole est porté par ConnectorInterface
Glu	«COSAGlu» (instances de métaclasse AssociationClass)	Relie seulement des COSA-RequiredRole et des COSA-ProvidedRole.
Configuration (Type)	«COSAConfiguration» (instances de métaclasse Component)	- COSAConfiguration peut avoir des ports Component-Interface. - Configuration se compose de types (composants et connecteurs), contient au moins un COSAComponent

Attachment	Assemnbly connector	Tous les connecteurs d'assemblage relient des ports et des rôles.
Binding	Delegation connector	Tous les connecteurs de délégation relient deux ports ou deux rôles.
Use	«Use» (instances of meta class Association)	Use doit relier seulement les ports et les services (ou rôles et Connecteur-Service)
Service	«Service» (instances de métaclasses Class)	Le propriétaire est ComponentInterface
Connector-Service	«Connector-Service» (instances de métaclasses Class)	Marqué par un attribut énumérée type : **Type-Service**
		Le propriétaire est ConnectorInterface
Properties	«COSANonFuncProp» (instances de métaclasses Property)	COSANonFuncProp correspond à un attribut d'un élément architectural COSA.
Implementation	«Implementation» (instances de métaclasses Class)	COSAImplementation est une classe d'implémentation d'un élément de COSA.
Constraint	«COSAConstraint» (instances de métaclasses Constraint)	Toutes les contraintes de COSA doivent être invariantes

Table 4.6 *Projection de COSA vers UML 2.0.*

L'élément architectural

L'élément architectural est un concept qui définit tous les concepts architecturaux de COSA. Ce concept n'a pas de correspond explicite dans UML. Ainsi, le profil UML doit inclure un stéréotype pour le représenter. Nous appelons ce stéréotype «*COSAArchitecturalElement*». Il correspond à la métaclasse Class du métamodèle UML. Ce dernier introduit le nouveau concept *EncapsuledClassifier*, qui permet à une classe, de pouvoir contenir des ports en tant que point d'interaction, associés par des interfaces « Fournies » et « Requises ». Une classe UML peut avoir des propriétés et des contraintes et peut être implémentée par une autre classe. Ceci peut être décrite dans OCL comme suit :

```
context UML::InfrastructureLibrary::Core::Constructs::Class
inv :self.isStereotyped("COSAArchitecturalElement")
implies
   (self.ownedAttribute->forAll->(a|a.isStereotyped("COSAProp"))) and
   (self.ownedRule->forAll->(r|r.isStereotyped("COSAConstraint")))and
   (self.clientDependency.target->forAll->(t|t.isStereotyped("COSAImp"))
```

- **Les propriétés**

Les propriétés fonctionnelles de COSA sont équivalentes au concept d'attributs d'UML alors que les propriétés non fonctionnelles sont représentées par des attributs stéréotypés «*COSANonFoncProp*» dans la mesure où elles sont différentes au sens sémantique puisque les attributs en UML sont des propriétés structurelles qui constituent une classe. Chaque attribut ayant un type et auquel on peut donner une valeur pour l'instance de l'élément architectural.

- **Les contraintes**

Chaque élément architectural peut avoir des contraintes, qui sont des restrictions et conditions qui doivent être vérifiés à tout moment. Dans UML 2.0, l'OMG définit formellement les contraintes OCL 2.0 (Object Constraint Language) (Object Management Group, 2005). Les contraintes UML exprimées via le standard OCL sont utilisées pour documenter les contraintes de COSA.

- **L'implémentation**

La classe correspondante au concept d'implémentation COSA dans UML 2.0. Une classe stéréotypée «COSAImplementation » est définit seulement par un élément architectural COSA et ne possède pas des ports. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Class
inv :self.isStereotyped("COSAImplementation")
    implies
    (self.ownedPort->isEmpty()) and
    (self.owner->forAll->o|o.isStereotyped("COSAArchitecturalElement")))
```

- **Les composants**

Un composant en UML 2.0 est plus expressif qu'une classe UML 2.0 et offre des services via les ports associés par des interfaces « Fournies » et « Requises ». Un composant est considéré comme un type remplaçable, possède la spécification de déploiement et il convient mieux aux composants COSA. Le type de composant de COSA est le type de composant dans UML 2.0, et les instances de composant de COSA correspondent aux instances de composant d'UML 2.0.

Tout composant stéréotypé «*COSAComponent*» doit avoir au moins un port stéréotypé «*ComponentInterface*». Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Component
inv :self.isStereotyped("COSAComponent")
    implies
    (self.ownedPort->size()>=1) and
    (self.ownedPort->forAll-(r.isStereotyped("ComponentInterface")))and
    (self.clientDependency.target->select
                          ->(t|t.oclIsKindOf(Interface)))->isEmpty())
```

- **Les connecteurs**

Les types de connecteurs COSA sont explicites et peuvent être réutilisés. La classe UML définit clairement le type de connecteur dans COSA et les instances du type de connecteur correspondent aux instances d'une classe UML 2.0. Toute classe stéréotypée « *COSAConnector* » doit avoir au moins un port stéréotypé « *ConnectorInterface* » et possède une seule classe d'association stéréotypée «*COSAGlu*». Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Class
inv :self.isStereotyped("COSAConnector")
implies
(self.ownedPort->size()>=1) and
(self.ownedPort->forAll->r.isStereotyped("ComponentInterface"))) and
(self.member->select(m|m.oclIsTypeOf(AssociationClass))
        ->forAll->(isStereotyped("COSAGlu")))->size()=1) and
      (self.clientDependency.target->select
                ->(t|t.oclIsKindOf(Interface)))->isEmpty())
```

- **La glu**

Un connecteur COSA définit le comportement local de chacune des parties en interactions. Ces comportements sont combinés pour former une communication via le concept de glu. Dans UML, le concept de classe association stéréotypée « *COSAGlu* » est parfaitement identique mais il reste à définir sa sémantique avec la contrainte OCL suivante:

```
context UML::InfrastructureLibrary::Core::Constructs::AssociationClass
inv :self.isStereotyped("COSAGlu")
     implies
      (self.owner.isStereotyped("COSAConnector")) and
      (self.memberEnd->select->
         (t|t.type.isStereotyped("COSAProvidedRole"))->size ()>=1)and
       self.memberEnd->select->
        (t|t.type.isStereotyped("COSARequiredRole"))->size ()>=1)and
        (self.memberEnd->forAll->(m|m.lowerBound()=2)
```

- **Les configurations**

Un aspect important de l'architecture COSA est celui de configuration qui est un graphe de composants et de connecteurs. Comme un composant UML peut contenir des sous-

composants et des sous-classes, les configurations COSA sont projetées vers un graphe de composants UML avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Component
inv :self.isStereotyped("COSAConfiguration")
implies
(self.ownedPort->forAll->(r.isStereotyped("ComponentInterface")))and
(self.member->select(m|m.oclIsKindOf(Component))->forAll
    ->(c|c.isStereotyped("COSAComponent")))->size()>=1)and
(self.member->select(m|m.oclIsTypeOf(Class))->forAll
    ->(c|c.isStereotyped("COSAConnector")))->size()>=1)and
(self.clientDependency.target->select
    ->(t|t.oclIsKindOf(Interface)))->isEmpty())
```

- **Les interfaces**

Les ports dans le standard UML 2.0 *superstructure* décrivent une vision externe d'un *encapsulated classifier* et offre des interfaces de types « fournis » et « requis » garantissant la documentation des interfaces COSA. Nous distinguons deux types d'interfaces (composant et connecteur) et nous intégrons dans le profil deux stéréotypes : un stéréotype nommé «*ComponentInterface*», qui sera porté par les ports d'un composant et un stéréotype nommé « *ConnectorInterface* » qui sera porté par les ports d'une classe.

Les interfaces des composants et des configurations : Le stéréotype «ComponentInterface» possède une valeur marquée portée sur les ports est nommé cible. Ces valeurs marquées peuvent être composant ou configuration et permettent de préciser le type de port à construire, à savoir composant ou configuration. En plus de ces valeurs marquées, un port stéréotypé «ComponentInterface» est défini seulement par un COSAComponent ou COSAConfiguration. Elle possède un ensemble non vide de ports. Nous spécifions la restriction par la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Port
inv :self.isStereotyped("ComponentInterface")
implies
   (self.owner.isStereotyped("COSAComponent") or
   (self.owner.isStereotyped("COSAConfiguration")) and
   (self.required->size()>=1 or self.provided->size()>=1)and
   (self.required->forAll->(p|p.isStereotyped("COSARequiredPort")) and
   (self.provided->forAll->(p|p.isStereotyped("COSAProvidedPort")) and
   (self.type->size()=1) and
       (self.type.oclAsType(Class).general->forAll->
         (g|g.isStereotyped("RequiredService") or
             g.isStereotyped("ProvidedService"))
```

Les interfaces des connecteurs : chaque port stéréotypée *«ConnectorInterface»* est défini seulement par un COSAConnector et possède un ensemble non vide de rôles COSA, nous spécifions la restriction par la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Port
inv :self.isStereotyped("ConnectorInterface")
     implies
      (self.owner.isStereotyped("COSAConnector") and
       (self.required->size()>=1 or self.provided->size()>=1)and
      (self.required->forAll->
         (p|p.isStereotyped("COSARequiredRole")) and
      (self.provided->forAll->
         (p|p.isStereotyped("COSAProvidedRole")) and
      (self.type->size()=1) and
      (self.type.oclAsType(Class).general->forAll->
         (g|g.isStereotyped("RequiredConnector-Service") or
              g.isStereotyped("ProvidedConnector-Service"))
```

Les ports et rôles

Les concepts d'interfaces « *Fournies* » et « *Requises* » de UML 2.0 permettent de documenter correctement les interfaces «Fournies» et «Requises» de COSA (les ports et les rôles), mais elles restent toujours distinctes avec leurs stéréotypes associés.

Un port stéréotypé «*COSAProvided-Port*» est défini seulement par une interface «ComponentInterface» et possède seulement des interfaces fournies. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Interface
inv :self.isStereotyped("COSAProvidedPort")
     implies
      (self.owner.isStereotyped("ComponentInterface"))and
      (self.required->size()->isEmpty()) and
      (self.provided->size()>=1))
```

Un port stéréotypé «*COSARequired-Port*» est défini seulement par une interface «*ComponentInterface*» et possède seulement des interfaces requises. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Interface
inv :self.isStereotyped("COSARerquiredPort")
     implies
      (self.owner.isStereotyped("ComponentInterface"))and
      (self.required->size()>=1) and
      (self.provided->size()->isEmpty())
```

Un port stéréotypé «*COSAProvided-Role*» est défini seulement par une interface «*ConnectorInterface*» et possède seulement des interfaces fournies. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Interface
inv :self.isStereotyped("COSAProvidedRole")
     implies
      (self.owner.isStereotyped("ConnectorInterface"))and
      (self.required->size()->isEmpty()) and
      (self.provided->size()>=1))
```

Un port stéréotypé *«COSARequired-Role»* est défini seulement par une interface *«ConnectorInterface»* et possède seulement des interfaces requises. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Interface
inv :self.isStereotyped("COSARequiredRole")
     implies
      (self.owner.isStereotyped("ConnectorInterface"))and
      (self.required->size()>=1) and
      (self.provided->size()->isEmpty())
```

- **Attachment**

Un connecteur d'assemblage UML est équivalent au concept d'attachement de COSA qui est défini entre un port et un rôle COSA. Le port du connecteur COSA de type fourni est relié avec le rôle COSA de type requis ou le port COSA de type requis est relié avec le rôle COSA de type fourni avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Connector
inv :self.isStereotyped("Attachment")
 implies
 (self.kind = #assembly) and
 ((self.end.exists (cp|cp.role.IsCOSAProvidedPort()) and
           (self.end.exists (cr|cr.role.IsCOSARequiredRole()) or
 (self.end.exists(cp|cp.role.IsCOSARequiredPort()) and
           (self.end.exists (cr|cr.role.IsCOSProvidedRole())
```

- **Binding**

Un connecteur de délégation UML correspond au concept de Binding COSA défini seulement entre une interface interne et une interface externe. Binding COSA relie deux ports COSA fournis (ou requis) de deux différents sous composants d'un même composant COSA avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Connector
inv :self.isStereotyped("Binding")
 implies
 (self.kind = #delegate) and
 (self.end.exists (cp1, cp2|cp1<>cp2 and
     ((cp1.role.IsCOSAProvidedPort()and cp2.role.IsCOSAProvidedPort())
    or(cp1.role.IsCOSARequiredPort()and cp2.role.IsCOSARequiredPort()))
    and (cp1.owner = cp2.owner))
```

Binding COSA relie deux rôles COSA fournis (ou requis) de deux différents sous connecteurs d'un même connecteur COSA avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Connector
inv :self.isStereotyped("Binding")
 implies
 (self.kind = #delegate) and
 (self.end.exists (cr1, cr2|cr1<>cr2 and
     ((cr1.role.IsCOSAProvidedRole() and cr2.role.IsCOSAProvidedRole())
     or(cr1.role.IsCOSARequiredRole() and
      cr2.role.IsCOSARequiredRole()))
     and (cr1.owner = cr2.owner))
```

- **Use**

Le concept d'association correspond au concept de Use dans UML 2.0. Il représente le rattachement physique entre un port/rôle et un service. Use COSA est définie entre un port COSA et Service COSA ou entre un rôle COSA et un Service-Connecteur COSA avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Association
inv :self.isStereotyped("Binding")
 implies
  ((self.end.exists (cp1|cp1.role.IsCOSAPort()) and
           (self.end.exists (cp2|cp2.role.IsService()) or
   (self.end.exists(cr1|cr1.role.IsCOSARole()) and
           (self.end.exists (cr2|cr2.role.IsService-Connector())
```

Attachement et Binding sont définies en utilisant les connecteurs d'UML 2.0, le connecteur d'assemblage pour l'attachement et le connecteur de délégation pour le Binding. Puisque UML 2.0 n'offre pas la capacité d'associer une information sémantique aux connecteurs (par exemple, une description comportementale) ou de décrire clairement les rôles de connecteur, on peut donner une sémantique plus riche aux Attachment, Binding et Use en les exprimant par des associations reliées au ConnecteurCOSA. Dans ce cas, Attachment, Binding et Use sont des sous-types du ConnecteurCOSA et ils peuvent avoir des rôles bien définies.

- **Extend**

Le concept d'association d'héritage correspond au concept de Extend dans UML 2.0. Il représente le. Extend COSA est définie entre un composant COSA et un autre composant COSA ou entre un connecteur COSA et un autre connecteur COSA avec la contrainte OCL suivante :

```
context UML::InfrastructureLibrary::Core::Constructs::Association
inv :self.isStereotyped("Extend")
 implies
 (self.kind = #extend) and
 (self.end.exists (cr1, cr2|cr1<>cr2 and
          and (cr1.type = cr2.type))
```

4.2.3.3.1.1 Le niveau modèle (M_1)

Le niveau M_1 permet de décrire un modèle d'architecture spécifique (ex : système client-serveur) avec l'application du profil UML 2.0 de COSA. Il peut permettre de définir les valeurs marquées des stéréotypes. Dans ce niveau, les contraintes OCL sont vérifiées et le système final projeté doit être conforme au profil UML, comme le montre la figure 4.3.

4.2.3.3.1.2 Le niveau application (A)

Ce niveau est un ensemble d'instances des types de composant, connecteur et configuration. Ces instances sont définies dans M_1. La figure 4.1 illustre deux instances *arch1* et *arch2* pour le système Client-Serveur.

4.2.3.3.2 Un exemple de projection : cas du système Client-Serveur

Pour illustrer notre stratégie de projection, nous proposons d'utiliser l'exemple du système Client-Serveur. La figure 4.4 décrit l'exemple du système Client-Serveur en COSA et la figure 4.5 montre la représentation de ce même système après l'application du profil UML proposé.

Les valeurs marquées de chaque stéréotype du système Client-Serveur permettent de préciser les caractéristiques des interfaces, des ports et services de COSA, comme le montre le tableau 4.7.

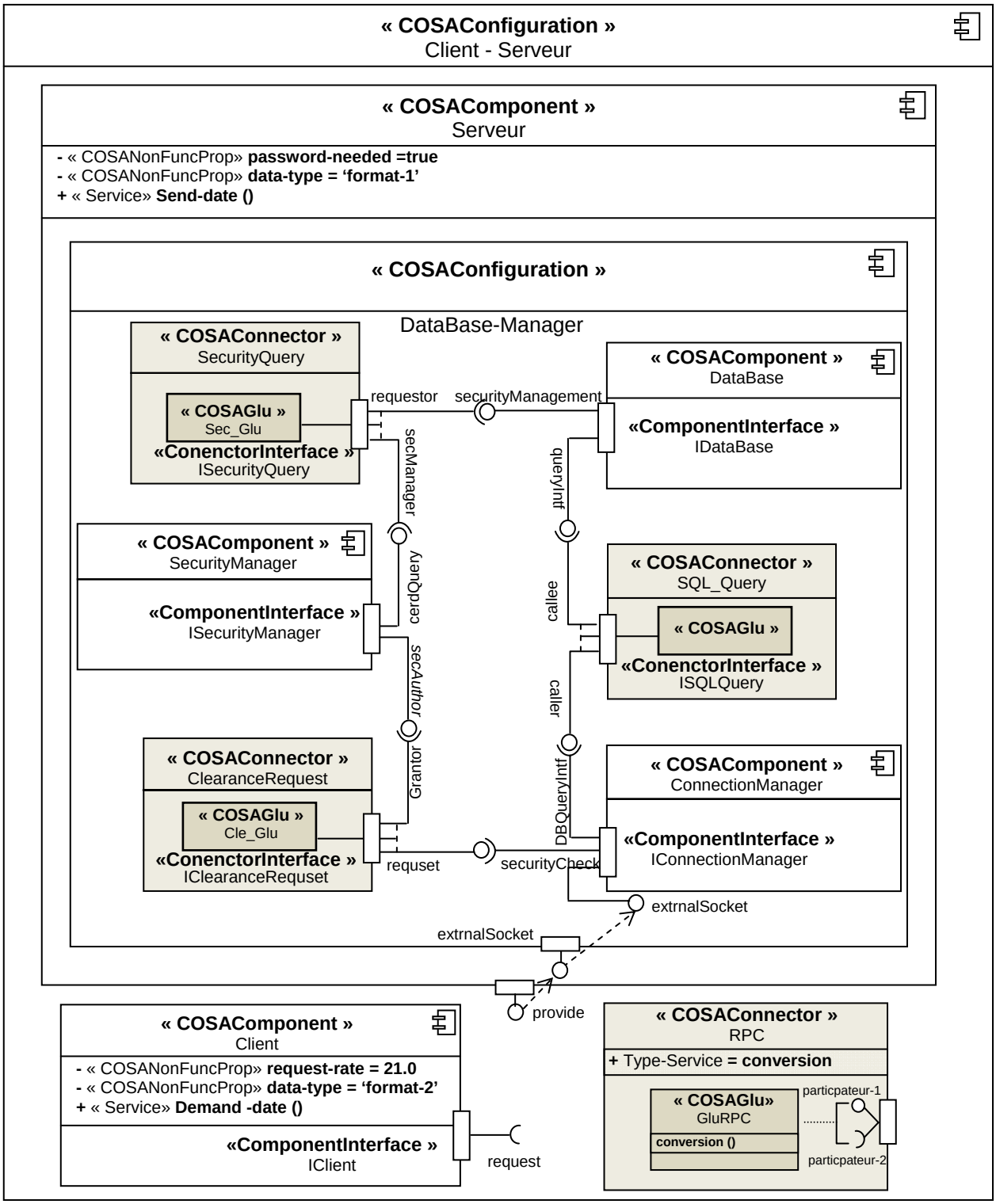


Figure 4.3 *L'application du profil pour le système Client-Serveur*

```
Class Configuration  client-serveur {
  Class Component Serveur {
    Properties {password-needed = true;  data-type = format-1;}
    Constraints {max-clients=1;}
    Interface { Connection-Mode : synchronous ;
              Ports provide{provide;}
              Services provide {Send-data;}
              Use {Send-data to provide;} }

   Define-Composition {
     Class Components {
       Class Component ConnectionManager {
         Interface { Ports provide {externalSocket; DBQueryIntf;}
                   Ports request {securityCheck;}}
       Class Component SecurityManager {
         Interface { Ports provide {securityAuto;CerdentialQuery;}}}
       Class Component Database {
         Interface { Ports provide{queryIntf;}
                   Ports request{securityManagement;}}}
       Class Connector SQLQuery {
         Interface { Ports provide{callee;} Ports request{caller;}}
       Class Connector CleranceRequest {
         Interface { Ports {grantor; requestor;}}
       Class Connector SecurityQuery {
         Interface { Ports provide{securityManager; requestor }
     }
     Attachments {
       ConnectionManager. SecurityCheck to CleranceRequest.requestor;
       SecurityManager. SecurityAutothorization to CleranceRequest.grantor;
       ConnectionManager.DBQueryIntf to SQLQuery.caller;
       Database. queryIntf  to SQLQuery.callee;
       SecurityManager.cerdentialQuery to SecurityQuery.SecurityManager;
       Database. securityManagement to SecurityQuery. Requestor;
     }
     Bindings {ConnectionManager. ExternalSocket to server.provide; }
   }
  }

  Class Component Client {
     Properties {Request-rate = 21.0;  data-type = format-2;}
    Interface { Connection-Mode : synchronous ;
              Ports request{request;}
              Services request {Damend-data;}
              Use { Damend -data to request;} }
  }

  Class Connector RPC {
     Interface {Roles {callee; caller}}
    Constraints {max-roles = 2;}
     Glue {Definition of service….}
  }

  Instance client-serveur arch-1 {
       Instances {
              S1: server;
              C1: client;
              C1-S1: RPC;}
       Attachments {C1.request to C1-S1. callee;
                    S1.provide to C1-S1. caller; }
  }
}
```

Figure 4.4. *Système Client-Serveur dans COSA.*

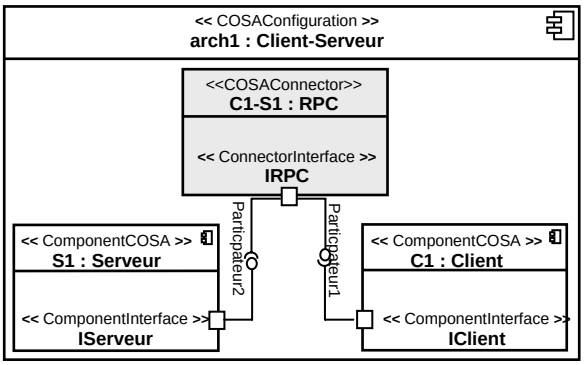


Figure 4.5. *Exemple du Client-Serveur en UML 2.0*

Concepts UML	Valeurs Marquées
IClient	Cible = composant
Besoin	Mode = synchrone
IServeur	Cible = composant
Service	Mode = synchrone
IRPC	Mode = synchrone

Table 4.7 *Définition des valeurs marquées de chaque stéréotype.*

4.2.4 Synthèse

Nous avons présenté la projection des concepts architecturaux COSA vers ceux des concepts d'UML 2.0 et ce, en précisant la façon dont elle est prise en compte par UML. Cette prise en compte passe par l'utilisation du profil UML pour COSA. Notre objectif était de représenter les concepts COSA en utilisant les notations UML 2.0. Grâce à ce profil, il est possible d'appliquer l'approche MDA à des modèles élaborés sous forme de composants UML 2.0 afin, de générer des modèles spécifiques pour les plates-formes d'exécution. Ces derniers facilitent la génération de code.

Le caractère expressif du langage UML 2.0 est le résultat de la disponibilité du concept de composant avec les ports qui sont associés par des interfaces « Requises » et « Fournies » et du concept de connecteur de types délégation ou d'assemblage. Cependant, UML manque de support explicite pour certains concepts architecturaux tels que les configurations ou les rôles. Il ne considère pas les connecteurs comme des entités de première classe, mais de simples liens entre composants et il ne peut pas modéliser directement un composant de communication. Nous avons vu l'intérêt du profil UML COSA pour préciser le sémantique des concepts d'architecture logicielle à projeter et ainsi d'affiner la façon dont ils sont exploités dans UML.

4.3 Architecture logicielle et MDA

En vu, de préciser les notations fondamentales de cette partie de notre travail, nous décrivons d'abord la démarche MDA, la notion de transformation dans MDA, les profils UML et MDA et ensuite les étapes de la démarche MDA.

4.3.1 La démarche MDA (Model Driven Architecture)

L'approche de l'architecture dirigée par les modèles (MDA) est une démarche de développement proposée par l'OMG (Object management Group). Les modèles de base de MDA sont les entités capables d'unifier et de supporter le développement de systèmes informatiques en assurant l'interopérabilité et la portabilité. Les modèles de base comme UML (Unified Modeling Language), CWM (Common Warehouse Metamodel), MOF (Meta Object Facility) représentent le noyau de MDA.

Les intergiciels représentent le niveau d'exécution des applications (par exemple CORBA, J2EE, .NET, etc.) et les services normalisés fournissent le support à l'exécution de ces applications (par exemple la sécurité des transactions et des évènements). Tout cet ensemble, de technologies est utilisé pour supporter différents domaines tels que la finance, le commerce électronique ou la télécommunication. Si le nombre de modèles de base disponibles est faible, il n'est pas cependant figé.

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'un logiciel. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (CIM), d'analyse et de conception (PIM) et d'implémentation (PSM). L'objectif primordial de MDA est l'élaboration des modèles pérennes, indépendants des détails techniques des plates-formes (J2EE, .Net, PHP ou autres), afin de permettre la génération automatique de l'implémentation des applications et d'obtenir un gain significatif de productivité.

4.3.2 Transformation et MDA

La notion de transformation est un aspect de base pour MDA (Model Driven Architecture) (MDA, 2002). Elle vise l'automatisation des transformations de modèles. En outre, les profils UML peuvent être intégrés dans le contexte MDA pour définir une

chaîne de transformations de modèles de l'architecture vers l'implémentation (Frankel 2003 ; MDA, 2002). Dans MDA, les modèles sont définis comme des entités de première classe. La mise en production de MDA passe par la mise en place, de ces modèles en relation et leurs transformations. MDA définit une transformation entre des modèles d'exigences (CIM), des modèles indépendants des plates-formes (PIM) et des modèles spécifiques (PSM).

- **Transformation de modèles CIM vers PIM :** permet de construire des PIM partiels à partir des informations contenues dans les CIM. L'objectif est de s'assurer que les besoins exprimés dans les CIM sont retranscrits dans les PIM. Ces transformations sont essentielles à la pérennité des modèles. Ce sont elles, qui garantissent les liens de traçabilité entre les modèles et les exigences exprimées par un architecte.
- **Transformation de modèles PIM vers PIM,** permet de raffiner les PIM afin d'améliorer la précision des informations qu'ils contiennent. En architecture logicielle, de telles transformations peuvent être, par exemple la création des modèles comportementaux à partir des modèles structuraux.
- **Transformation de modèles PIM vers PSM,** permet de construire une bonne partie des modèles PSM à partir des modèles PIM. Ces transformations sont les plus importantes dans le domaine de l'architecture logicielle, car elles garantissent la productivité des modèles et leur lien avec les plates-formes d'exécution.
- **Transformation de modèles PSM vers code,** permet de générer la totalité du code. Ces transformations ne sont pas considérées comme étant des transformations de modèles.

4.3.3 MDA et UML

MDA propose plusieurs métamodèles « core model » de base, correspondant au niveau M_2 de l'architecture de méta-modélisation à quatre niveaux (cf. section 2.3.2.3 de chapitre 1). Ces modèles sont également appelés « UML profiles » dans la terminologie MDA. Ils sont dédiés à une classe de problèmes, comme par exemple, « Entreprise Computing » ou « Real-Time Computing ». Certains de ces modèles sont en phase de standardisation. Ces modèles sont indépendants de toute plate-forme. Leur nombre devrait croître pour couvrir d'autres besoins. Cependant, il a vocation à rester

relativement peu important car chacun se veut « généraliste » et regroupe les caractéristiques communes à tous les problèmes de sa catégorie.

UML est préconisé par l'approche MDA comme étant un langage à utiliser pour réaliser des modèles d'analyse et de conception indépendamment des plates-formes d'implémentation. Une stratégie pour définir un métamodèle consiste à utiliser le mécanisme des profils UML. MDA (Model Driven Architecture), recommande cette approche car le support industriel d'UML la rend facile à mettre en oeuvre. La définition d'un profil est une spécification conforme à un ou à plusieurs points :

– identification d'un sous-ensemble du métamodèle UML, qui peut être le métamodèle UML entier,
– définition de règles de bonne construction (« well-formedness rules »), en plus de celles contenues dans le sous-ensemble du métamodèle UML. Le terme « well-formedness rules » est utilisé dans la spécification normalisée du métamodèle. De telles règles permettent de décrire un ensemble de contraintes en langage naturel, et à l'aide du langage OCL d'UML (Warmer, Kleppe, 1998), afin de définir un élément du métamodèle.
– définition d'éléments standards (« standard elements»), en plus de ceux contenues dans le sous-ensemble du métamodèle UML. Le terme « standard elements » est utilisé dans la spécification du métamodèle UML pour décrire une instance standard d'un stéréotype UML ou une contrainte.
– définition de nouvelles sémantiques, exprimées en langage naturel, en plus de celles contenues dans le sous-ensemble du métamodèle UML.

- **UML pour les PIM** : le métamodèle UML contient tous les concepts relatifs aux modèles UML. UML définit plusieurs diagrammes permettant de décrire les différentes parties d'une application. Les modèles UML sont indépendants des plates-formes d'exécution. En outre, le métamodèle UML constitue le métamodèle idéal pour l'élaboration des PIM (Platform Independent Model).

- **UML pour les PSM** : les profils UML ciblent les plates-formes d'exécution. Grâce aux profils UML, il est possible d'utiliser UML pour élaborer des PSM. MDA

recommande l'utilisation de profils UML pour l'élaboration de PSM et la transformation PIM vers PSM puisque PIM et PSM sont tous deux, des modèles UML.

4.3.4 Les étapes de la démarche MDA

La première étape du développement d'une application basée sur MDA est la création d'un modèle d'application indépendamment de la plate-forme (PIM – Platform Independent Model) par instanciation du métamodèle utilisé. Dans un second temps, des spécialistes de la plate-forme cible sont chargés de la conversion de ce modèle d'application général vers CCM, EJB ou vers une autre plate-forme. Des transformations standards (standard mappings), en fonction du « core model », permettent d'envisager une automatisation partielle de la conversion. A l'occasion, ils génèrent, non seulement des artefacts propres à la plate-forme (en langage IDL et autres), mais aussi un modèle spécifique (PSM). Ce modèle est également décrit à l'aide du profil UML. On parle alors de « dialecte d'UML», laissant apparaître, du fait de la transformation des éléments liés à l'exécution sur la plateforme cible. Ceci permet d'exprimer de façon plus riche la sémantique de la solution qu'avec IDL ou XML. L'étape suivante est la génération du code lui-même. Plus le dialecte UML du PSM reflète la plate-forme cible, plus, la sémantique et le comportement de l'application peuvent être intégrés dans le PSM, et le code généré est plus complet peut être. Parmi les (nombreux) avantages de la démarche MDA, cités par l'OMG, on peut signaler la facilité à gérer l'interopérabilité entre applications à l'aide du même « core model ».

4.4 Intégration de l'architecture logicielle COSA au sein de MDA

4.4.1 Les étapes de la démarche MDA

L'architecture logicielle fournit un haut niveau d'abstraction pour représenter la structure d'un système, ce qui permet de réduire sa complexité. Le non respect de l'architecture logicielle dans les plates-formes d'exécution comme CORBA, J2EE, .NET, etc. sont les conséquences des nouveaux problèmes d'ambiguïtés et d'interopérabilité. Les intergiciels sont totalement intégrés dans les plates-formes d'exécution. Quelque soit le degré d'intégration de l'intergiciel, il reste toujours difficile de distinguer l'architecture (structure du système) de l'implémentation (code). Cela diminue le degré

de réutilisation des logiciels et augmente le coût d'un changement d'intergiciel. Il s'agit dans tous les cas de séparer les aspects d'architecture et des aspects d'implémentation. Les aspects d'architecture sont définis en haut niveau d'abstraction en terme de composants de systèmes et de leurs interconnexions sans aucune dépendance envers la technologie. Par contre, les aspects d'implémentation sont définis en bas niveau d'abstraction en terme de composants du système spécifiques aux technologies des plates-formes. La prise en compte de l'architecture logicielle dans les plates-formes d'exécution permet de résoudre l'hétérogénéité et faciliter la communication et la coordination des composants distribués.

4.4.2 Intégration des ADLs au sein de MDA

Plusieurs propositions d'intégration des concepts architecturaux ont vu le jour ces dernières années. Garlan (Garlan, 2000b), persiste à dire que dans le monde de développement du logiciel et dans le contexte d'utilisation du logiciel les méthodes changent considérablement. Ces changements sont le résultat de l'impact majeur des manières d'utiliser une architecture.

Le projet ACCORD RNTL, (ACCORD RNTL Project, 2002) est un environnement ouvert et distribué, fournit une simplicité de développement par l'assemblage des composants. Il définit une correspondance semi-automatique des concepts et une transformation automatique de modèle ACCORD vers Component CORBA Model (CCM). Ce travail se base sur les profils UML pour représenter les concepts ACCORD et les concepts CCM. Il définit un filtre intermédiaire pour l'adaptation du processus de transformation. Ensuite l'assemblage des composants est définis via les fichiers XML, et donc difficile de garantir la réutilisation des composants.

Dans (Rodrigues, Lucena, Batista, 2004), les auteurs définis des règles de projection pour transformer une description de l'architecture ACME vers une spécification CORBA IDL (Barlett, 2004). Ils focalisent la composition des systèmes par l'exploration des extensions ACME pour inclure les ports d'entrée/sortie dans la spécification ACME. Ils transforment chaque concept du langage ACME en une interface IDL, et de ce fait, ne profitent pas réellement des concepts disponibles dans CORBA IDL.

Dans (Manset, Verjus, McClatchey, Oquendo, 2006), les auteurs exploitent les capacités du langage de description d'architecture Archware et les plates-formes d'exécution dans le processus de développement ACMDD (Architecture-Centric Model Driven Developement). Ils intègrent effectivement la vision architecture au sein de MDA.

Dans (Marcos, Acuna, Cuesta, 2006), les auteurs proposent une extension de l'approche MDA nommée ACMDA (Architecture-Centric Model Driven Architecture) par l'intégration effective des aspects d'architecture. Cette approche considère une architecture comme une entité de première classe et défini des modèles d'architecture dans deux niveaux d'abstraction distincts; un niveau des modèles d'architecture indépendant de la plate-forme PIA (Platform Independent Architecture) et un autre niveau des modèles d'architecture spécifiques PSA (Platform Specific Architecture). L'architecture logicielle est considérée comme perspective qui permet une meilleure séparation des différents aspects (contenu, hypertexte, comportement) envers les niveaux d'abstractions, et elle devient un gestionnaire des instances des modèles d'architecture. Elle garantit un modèle approprié lors de l'instanciation du composant. Ce travail est diminué des moyens et des techniques de projection des architectures indépendantes de plate-forme vers architectures spécifiques à cause de l'absence des langages de description des architectures (ADLs) qui fournissent tous les concepts et les styles architecturaux.

Dans (Sánchez et al. 2006), les auteurs expriment des liens de traçabilité et de raffinement par une transformation des aspects de besoins vers des aspects architecturaux. Ce travail fournit une structure de framework MDA confortable, une meilleure séparation des préoccupations (architecture, exigences) et de plus la prise en compte des aspects d'architecture au sein de framework MDA.

4.4.3 Intégration de l'ADL COSA au sein de MDA

L'absence de quelques concepts architecturaux (connecteur, configuration, etc.), dans UML 2.0 ainsi que le non respect de l'architecture logicielle dans les plates-formes d'exécution comme CORBA, J2EE, .NET, etc. sont les conséquences des nouveaux problèmes d'ambiguïtés et d'interopérabilité. La solution recommandée est la démarche MDA qui fournit des moyens de projection rapide des concepts architecturaux COSA

vers UML 2.0 par une transformation de l'architecture abstraite COSA en une architecture concrète UML et des moyens de séparation des préoccupations des aspects d'architecture et des aspects d'implémentation par une transformation automatique de l'architecture vers l'implémentation dans les plates-formes d'exécution.

L'intégration des langages de description de l'architecture logicielle (ADLs) telle que COSA; au sein de l'approche MDA est une perspective qui offre aux intervenants de système et aux plates-formes d'exécutions une autre vue de gestion et de maintenance de l'implémentation, par conséquent l'implémentation devient un résultat de l'architecture qui permet de bénéficier de la concrétisation des plans architecturales et de l'évolution des architectures logicielles.

Les plates-formes MDA offrent une simplicité de développement par l'assemblage des composants préfabriqués mais il ne supporte pas des niveaux d'abstraction élevés, le concept de connecteur et composant composite. D'ailleurs, COSA supporte les composants composites et définit explicitement les connecteurs comme des concepts architecturaux abstraits. Donc, il très utile à définir une transformation automatique de modèle d'architecture logicielle, le PIM (Platform Independent Model) vers un modèle de plate-forme d'exécution le PSM (Platform Specific Model). L'intérêt fondamental est la projection rapide et une meilleure intégration des concepts d'architecture logicielle vers les plates-formes MDA pour aboutir un niveau d'abstraction plus élevé et aider à résoudre les problèmes d'interactions entre les composants des plates-formes d'exécutions.

MDA prend en compte les langages de description d'architecture en intégrant leur description aux modèles PIM et lors de la transformation PIM vers PSM. L'objectif de COSA au sein de MDA est de permettre la décomposition et l'organisation de système en éléments architecturaux compréhensible et facilement manipulables, de migrer facilement d'une architecture vers une autre, et donc capitaliser l'exploitation des concepts architecturaux COSA et faciliter le passage vers les plates-formes d'exécution.

La prise en compte des langages de description d'architecture par MDA telle que COSA; se fait sur deux niveaux, dans les PIM (Platform Independent Model) et dans les transformations PIM vers PSM (Platform Specific Model) (voir figure 4.6). Les CIMs (Computation Independent Model) ne sont pas nécessaires ni pour la définition des

concepts architecturaux COSA ni pour sa projection vers les plates-formes d'exécution (Alti, Khammaci, Smeda, Bennouar, 2007c ; Alti, Smeda, 2007).

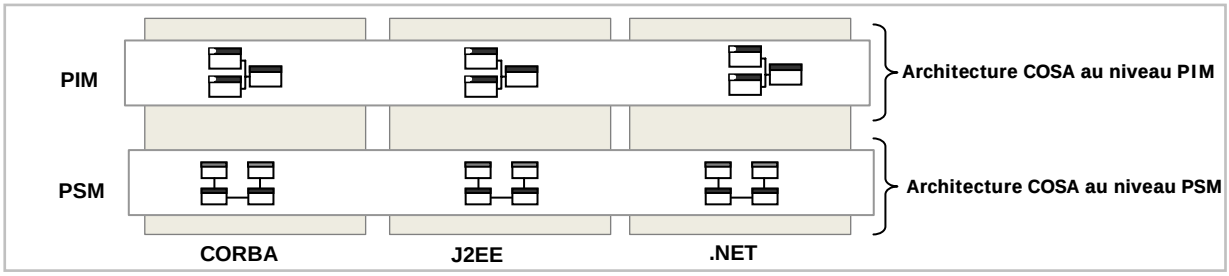


Figure 4.6 *Intégration de l'architecture COSA au sein de la démarche MDA.*

- **l'architecture COSA au niveau PIM:** le métamodèle PIM contient tous les concepts architecturaux relatifs à l'approche COSA. Grâce au mécanisme du profil UML, on réalise des transformations PIM vers PSM et on intègre les concepts COSA dans UML et dans les plates-formes d'exécution. Tous les concepts COSA se retrouvent dans ce métamodèle (PIM).

- **l'architecture COSA au niveau PSM:** les transformations PIM vers PSM précisent la façon dont le standard UML et les plateformes cibles (CORBA, J2EE, etc.), utilisant les modèles d'architectures COSA contiennent tous les concepts architecturaux projetés et disponibles à l'exploitation. Cette concrétisation de la transformation des PIM vers PSM préserve la séparation des préoccupations. Le modèle d'architecture et le modèle d'implémentation se trouvent séparés dans les modèles PIM et PSM et le lien entre ces deux modèles réside toujours dans les règles de transformation.

4.4.3.1 Stratégie de transformation des profils

Supposez-nous transformer le modèle COSA (PIM) qui conforme au métamodèle COSA, vers un autre modèle spécifique de la plateforme MDA (PSM) qui conforme à un autre métamodèle. PIM et PSM n'ont pas les mêmes concepts d'architecture. Cela implique que la définition des règles de transformations est assez difficile. Par conséquent, nous proposons le moyen de la transformation directe des profils qui facilite l'élaboration rapide des règles de transformations. Les techniques et les mécanismes d'extensibilité, fournis par les profils UML sont convenables à être utilisés dans la description d'architecture logicielle. La clé pour achever la vision MDA, est les règles de transformation de modèles. Ces règles sont définies au niveau des

métamodèles PIM (COSA) et PSM (profil UML pour COSA et profil UML pour une plate-forme MDA). L'idée d'élaboration des règles de transformation de PIM vers PSM est de prendre chaque élément architectural dans le métamodèle PIM et chercher ses correspondances dans le métamodèle PSM (la même sémantique des éléments UML de PIM). Chaque élément de correspondance contient une expression OCL 2.0 (OMG, 2005), permettant d'effectuer la transformation entre les éléments des profils UML pour COSA et la plate-forme MDA, et un filtre pour faire distinction entre eux. En plus, si les éléments du profil COSA inclus les spécifications des relations de ses éléments, la transformation doit inclure des opérations qui traduisent bien ses relations.

4.4.3.2 Exemple illustrative : la transformation de COSA-UML vers CORBA

Pour illustrer la stratégie de transformations systématiques de transformation de profils, nous l'appliquons, de COSA-UML (PSM) vers CORBA (Object Management Group, 2002) (PSM). La figure 4.7 présente le processus de transformation de l'architecture logicielle COSA vers la plate-forme CORBA. Nous proposons de transformer chaque concept architectural du modèle COSA vers COBA à l'aide du profil UML 1.4 pour CORBA (Object Mangement Group, 2001). Les composants COSA sont représentés par des composants UML 2.0. A chaque composant UML 2.0 correspond une classe UML 1.4 (la classe porte le même nom que le composant), le composant UML 2.0 «COSAComponent» peut être transformé vers une classe UML «CORBAHome».

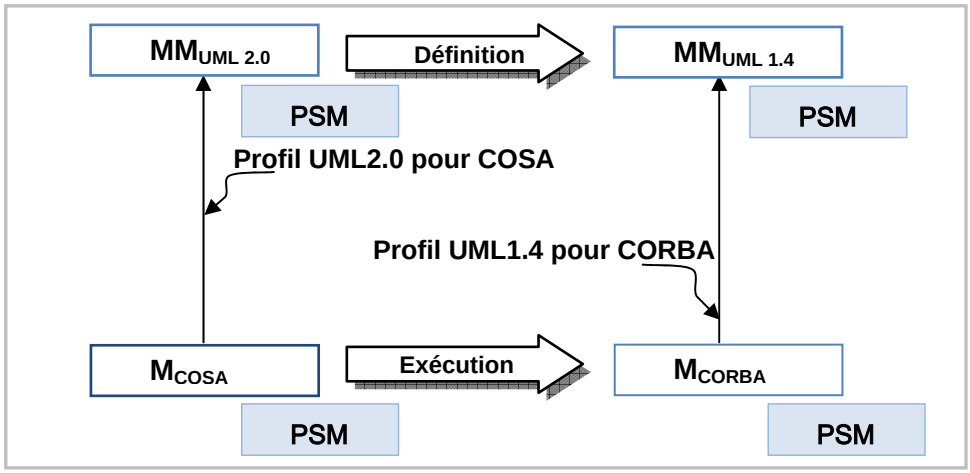


Figure 4.7 *Transformation de COSA-UML (PSM) vers CORBA-UML (PSM).*

Les connecteurs COSA sont représentés par des classes UML 2.0. Les classes UML 2.0 correspondent aux classes UML 1.4. Ainsi, la classe «COSAConnector» est transformée vers une classe «CORBAHome». Les interfaces de composants et des connecteurs correspondent aux ports UML 2.0. Les ports correspondent aux classes UML (le nom de la classe correspond au nom du port). Ces classes réalisent les interfaces fournies du port et dépendent des interfaces requises par le port. Une classe UML (qui représente un composant UML 2.0) doit être reliée à une autre classe (qui représente des ports). Chaque classe représentant un composant ou un connecteur (composants et connecteurs «CORBAHome») doit être rattachée à des associations d'agrégation vers les classes des ports (qui correspondent à une classe «CORBAComponent»). Le tableau 4.8 résume les principales correspondances entre les concepts de COSA et les concepts de CORBA (Alti, Khammaci, Smeda, Bennouar, 2007 ; Alti, Khammaci, Smeda, 2007b).

Concepts COSA UML 2.0	Concepts CORBA UML 1.4
«COSAComponent » Composant	**«CORBAHome»** Classe
«COSAConnector » Classe	
«COSAConfiguration» Composant	**«CORBAModule»** Package
«ComponentInterface» Port	**«CORBAComponent»** Classe
«ConnectorInterface» Port	

Table 4.8 *Correspondances COSA – CORBA.*

4.4.4 Approche MDA centrée sur la décision architecturale

Actuellement, le développement dirigé par les modèles (Model Driven Development (MDD)) est généralement basé sur des modèles qui sont dédiés à la production des architectures de qualité. Les décisions architecturales sont inclus implicitement dans l'approche MDA ce qui conduit à un manque de la traçabilité des décisions prises par le concepteur sur les architectures et ainsi à un manque de la traçabilité des décisions prises par le développeur sur les implémentations. Cela nécessite une définition explicite de cette notion et une considération de première classe. La définition explicite des décisions d'architectures vise à contrôler la qualité du processus de développement logiciel. Nous proposons d'étendre le framework MDA en intégrant les aspects de décisions architecturales et nous proposons aussi de modéliser les décisions architecturales comme une nouvelle dimension dans le processus de MDD.

L'intégration des décisions d'architectures va permettre, d'une part, de guider l'architecte pour créer des systèmes de qualité et d'autre part d'intégrer les véritables préoccupations des décisions dans le processus MDD.

Lors de l'application de la stratégie du profil UML sur un système client-serveur on a constaté que la décision architecturale n'était pas prise en compte. L'architecture logicielle a été conçue avec un ADL arbitraire (COSA, Pi-ADL, ACME, etc.) et aussi intégrée dans une plate-forme arbitraire (CORBA, EJB, .Net, etc.). Au niveau PIM, l'utilisation d'un ADL arbitraire à la description des systèmes logiciels ne répond pas aux facteurs de qualité des ADLs tels que la réutilisabilité et l'extensibilité. Au niveau PSM, l'intégration des concepts d'ADLs dans une plateforme donnée ne répond pas aux facteurs de qualités tels que la réutilisabilité et la portabilité. Chaque ADL à ses méthodes de spécification et d'analyses des propriétés structurelles et comportementales des systèmes logiciels. Chaque plateforme d'implémentation à ses facteurs de qualité tels que la performance et l'interopérabilité. On a besoin d'un modèle d'architecture qui permette de décrire clairement un système logiciel. On a besoin également d'intégrer facilement les concepts d'ADLs dans une plateforme interopérable. Pour cela, nous avons étendu MDA afin de supporter les décisions de choix des ADLs et les décisions de choix des plateformes au sein de MDA selon les caractéristiques de qualité des ADLs et les caractéristiques de qualité des plateformes.

4.4.4.1 Intégration des décisions architecturales au sein de MDA

Une bonne technique pour concevoir une architecture logicielle est d'intégrer les heuristiques des décisions de conception d'un modèle d'architecture avec un système donné. En se basant sur les caractéristiques de qualité des ADLs (extensibilité, portabilité, réutilisabilité, etc.), les décisions architecturales permettent de sélectionner un modèle d'architecture adéquat qui représente mieux nos besoins d'architecture logicielle. Cette prise en compte passe par l'intégration des facteurs d'impacts des décisions à l'étape de conception et non pas au cours du processus de transformation. De ce point de vue, nous ne pouvons pas décrire les systèmes de logiciel avec un ADL arbitraire, mais nous devons expliciter les raisons de sélectionner ce dernier parmi autres. Ensuite, nous considérons les métriques de choix des plateformes MDA (comme

la prise en compte de l'intégration facile des concepts d'architecture logicielle) avant la phase d'implémentation.

Les décisions de conception architecturale permettent de déterminer quels sont les modèles d'architectures logicielles pour chaque aspect architectural (structurel et comportemental) pour mieux décrire une architecture logicielle d'un point de vue conceptuel et indépendamment des contraintes technologiques. Après quelques décisions sur les architectures, tous les composants de description des systèmes de logiciel peuvent être implémentés sur plusieurs plateformes d'exécution en fonction de nos besoins spécifiques, et des technologies disponibles, etc. (Alti, Smeda, 2008).

4.4.4.2 Améliorer le processus de développement en utilisant MDD – centré décision

Pour illustrer notre approche centrée décision, nous l'appliquons au système Client/Serveur. La table 4.8 présente un exemple de comparaison entre CORBA et EJB afin de sélectionner la meilleure plate-forme d'implémentation pour le modèle COSA. Les plateformes CORBA et EJB se basent sur différents concepts d'implémentations.

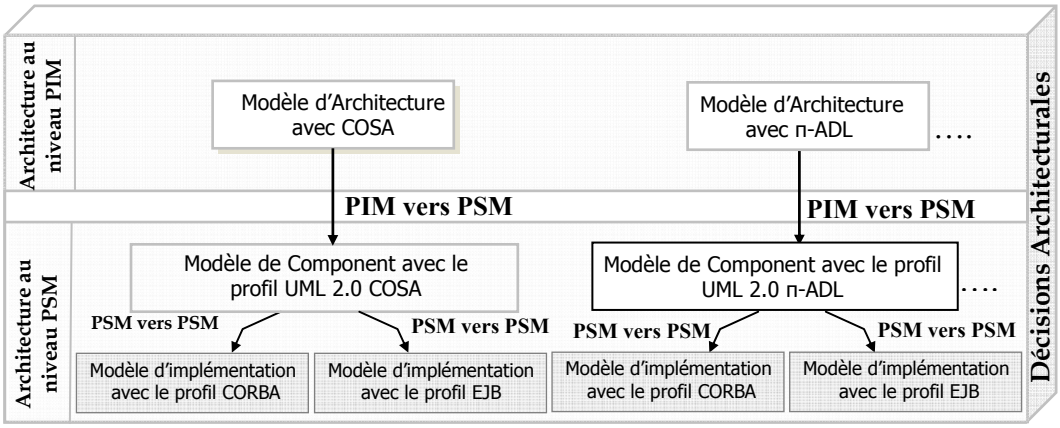


Figure 4.8 *MDD centré sur la décision architecturale (vue structurelle).*

Décision : Nous avons choisi le standard international CORBA qui simplifie le développement par rapport aux autres plates-formes, qui masque l'hétérogénéité par le biais du langage IDL (Interface Defintion Language) et qui facilite la communication entre des composants distribués via le bus ORB (Object Request Broker).

Critères	La plateforme CORBA	La plateforme EJB
Règles de transformation	Nous devons définir les règles de transformation de COSA vers CORBA.	Nous devons définir les règles de transformation de COSA vers EJB.
Contraintes d'implémentation	Le modèle CORBA doit être validé par le profil CORBA	Le modèle EJB doit être validé par le profil EJB.
Conséquences	Le système client / serveur dépend du profil CORBA	Le système client / serveur dépend du profil EJB
Avantages	- Résolution des problèmes d'interactions entre les composants CORBA par la projection des connecteurs COSA. - Simplicité de développement. - Intégration facile des concepts COSA.	- Résolution des problèmes d'interactions des composants EJB par la projection des connecteurs COSA. - Niveau d'abstraction haut celle de COSA.
Inconvénients	La prise en charge d'un nombre relativement faible des concepts d'implémentation.	Difficile de projeter des concepts d'architecture.

Table 4.9 *CORBA vs. EJB.*

4.4.5 Synthèse

Nous avons présenté la prise en compte des concepts d'architecture logicielle COSA au sein de MDA. Cette prise en compte passe par l'utilisation de profil UML. Les profils UML représentent tous les concepts de modèle d'architecture logicielle COSA et les modèles des plates-formes MDA dans le métamodèle UML. Ils facilitent l'élaboration des correspondances, l'automatisation des règles de transformations et la génération des modèles instances des plates-formes MDA. Nous avons présenté brièvement la transformation COSA vers CORBA. Cette transformation est une transformation PSM vers PSM. Elle permet une transformation COSA profil UML (PSM) vers CORBA profil UML (PSM). Notre objectif est uniquement de démontrer la faisabilité de l'approche de transformation des profils.

4.5 Conclusion

COSA est un ADL hybride qui englobe les concepts communément admis par la majorité des langages de description d'architectures logicielles. Un nouveau profil UML dédié à l'architecture logicielle COSA et une stratégie de projection directe des

concepts architecturaux COSA vers ceux des concepts d'UML 2.0 à l'aide ce profil est bien explicité. On peut conclure que les possibilités offertes par le profil UML, permet de définir une spécification complète et structurée de l'architecture logicielle COSA en UML 2.0 et permet de mieux capturer, plus précisément les concepts de l'architecture COSA en UML 2.0 (stéréotypes, valeurs marquées et contraintes OCL).

L'intégration de l'architecture logicielle COSA au sein de la démarche MDA par l'utilisation du profil UML offre aux plates-formes MDA un niveau d'abstraction élevé équivalent à celui de COSA et élimine les problèmes d'hétérogénéités. Crâce aux profils, il est possible d'appliquer l'approche MDA à des modèles élaborés sous forme de composants UML 2.0, afin de générer des modèles spécifiques pour les plates-formes d'exécution. Ils représentent tous les concepts de modèle d'architecture logicielle COSA et les modèles des plates-formes MDA dans le métamodèle UML. Ils facilitent l'élaboration des correspondances, l'automatisation des règles de transformation et la génération automatique des modèles d'instance des plates-formes d'implémentation.

Tant qu'à la réalisation des transformations, COSA vers UML 2.0, COSA vers eCore et COSA vers CORBA, elle fera objet dans le chapitre qui suit.

Chapitre 5 : Réalisations et expérimentations

5.1 Introduction

COSA est un ADL qui évolue avec des objectifs académiques plutôt qu'avec des objectifs industriels. Un ADL n'est utile que si nous fournissons des outils de support. Une validation pratique de COSA peut être effectuée grâce à une implémentation objet. Pour ce faire, nous avons opté pour une approche de type « profil UML » pour bénéficier des outils de modélisation UML et pour générer rapidement des outils graphiques pour l'architecture logicielle COSA. Elle permet ainsi de comprendre et contrôler les interactions et les interconnexions entre composants. L'objectif est la projection rapide et une meilleure intégration des concepts d'ADLs dans les plates-formes d'exécution pour accomplir un niveau d'abstraction plus élevé. Dans ce chapitre, nous exposons notre mise en œuvre de MDA sur l'intégration de l'architecture logicielle COSA avec l'exploration de trois outils de transformations :

COSAPlugin (Alti, Khammaci, Smeda, 2007a) : fournit un profil UML 2.0 pour COSA qui contient un ensemble de stéréotypes avec toutes ses valeurs marquées et contraintes OCL 2.0 nécessaire à la validation pratique de la projection COSA vers UML 2.0.

COSABuilder & COSAInstantiator (Maillard & al. 2007, Alti & al. 2010) : l'outil COSABuilder vise à permettre aux architectes de décrire graphiquement leurs architectures avec MOF et de valider automatiquement ses sémantiques structurelles selon l'approche COSA. L'aspect clé de l'outil COSAInstantiator est la possibilité de générer automatiquement un éditeur graphique pour un modèle d'architecture COSA et de créer des architectures d'applications correctes.

COSA2CORBAPlugin (Alti, Khammaci, Smeda, 2007b): est un outil de transformation des architectures COSA vers CORBA qui offre une simplicité de transformation des

modèles d'architectures UML 2.0, profilé COSA vers la plate-forme standard CORBA. Plus précisément, l'automatisation de la production des modèles UML des applications CORBA et la génération de code.

Nous explicitons pour cela les mises en œuvre des transformations avec ses résultats pour une application très répandue dans la communauté des architectures logicielles: un système de type Client-Serveur. Nous allons d'abord décrire chacun des outils, pour les appliquer au système Client-Serveur. Nous terminons cette étude, par une comparaison des deux transformations (COSA-UML, COSA-CORBA) suivant les critères de niveau de description, de séparation des aspects de conception et d'implémentation, de raffinement, de degré de lisibilité et de type de transformation.

5.2 Mise en œuvre de l'intégration de COSA au sein de MDA

Pour répondre aux problématiques de complexité, d'évolutivité et d'hétérogénéité des plateformes exécutables objets, il semble nécessaire aujourd'hui de recourir à des stratégies relativement nouvelles, fondées sur des techniques de nature génératives organisées autour de nombreux profils s'appuyant sur les métamodèles standards UML. Pour illustrer l'intégration de l'architecture logicielle COSA au sein de MDA à l'aide du profil COSA UML (Alti, Khammaci, Smeda, Bennouar, 2007) et souligner les avantages concrets qu'elle apporte, nous allons détailler les différents modèles d'architectures et transformations de modèles de l'architecture vers l'implémentation nécessaire à l'intégration du modèle COSA selon les principes MDA. Conformément à l'approche MDA, le modèle COSA représente le modèle PIM. Ce PIM perd tous ses atouts dès qu'il est projeté au niveau d'implémentation dans la mesure où il ne permet que la description d'un système informatique de manière abstraite. Comme le passage du langage UML2.0 vers des plates-formes objets est naturel, une solution est de réaliser une transformation des concepts de COSA vers les concepts d'UML2.0. L'intérêt de cette solution est de permettre la projection de l'architecture abstraite COSA (PIM) vers une architecture concrète UML 2.0 (PSM) et de pouvoir utiliser tous les outils UML. Nous avons sélectionné la plate-forme CORBA et la technologie eCore/MOF. Le fait d'avoir un langage de modélisation standard UML, une technologie standard

eCore/MOF et la plate-forme d'exécution standard CORBA nous permet de mettre en lumière les solutions qu'offre MDA pour résoudre les problèmes d'ambiguïtés, d'interopérabilité et de séparation des préoccupations des aspects d'architecture, des aspects d'implémentation et des aspects d'application. La figure 5.1 illustre les différentes étapes réalisées pour l'intégration de l'architecture COSA au sein de MDA. La transformation COSA vers UML 2.0 (PIM vers PSM), la transformation COSA vers eCore (PSM vers PSM) et la transformation COSA vers CORBA (PSM vers PSM). Le travail reste ouvert vers d'autres plates-formes : EJB (Alti, 2008) et .Net.

Les outils de l'architecture logicielle COSA contribuent à intégrer facilement les concepts d'architectures logicielles (connecteurs, configuration, glu, rôles…etc.) au sein des plates-formes MDA et améliorer plusieurs métamodèles standards et génériques comme UML et MOF, visant à promouvoir l'approche « profil UML » et apporte ainsi des solutions pragmatiques évoquées plus haut.

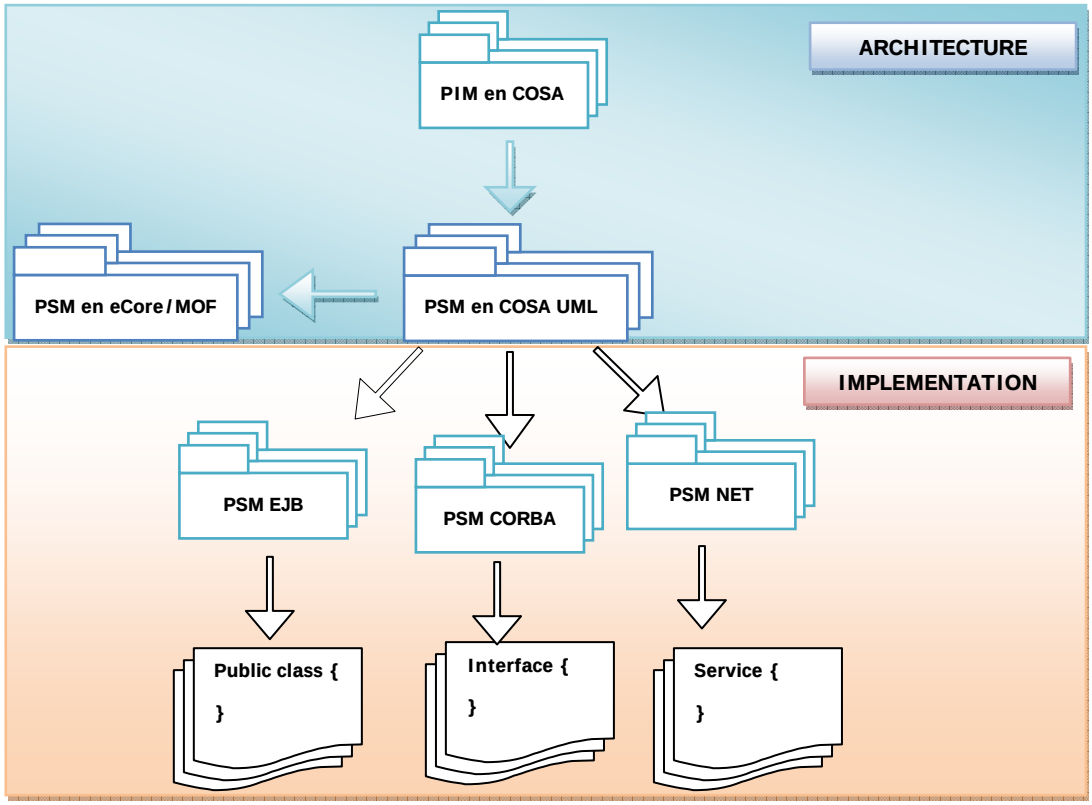


Figure 5.1 *Intégration dans MDA de l'architecture logicielle COSA.*

5.2.1 Transformation COSA (PIM) en UML 2.0 (PSM)

Nous avons réalisé une projection directe de COSA vers UML par l'intermédiaire du profil UML COSA. Ce choix s'explique par le fait que nous ne considérons pas UML comme un vrai ADL, tandis que les ingrédients dont nous avons besoin pour décrire les architectures logicielles se trouvent dans l'ADL COSA (voir le chapitre précédent).

Le profil UML 2.0 pour COSA est écrit en Java a été implémenté dans la plateforme de développement logiciel d'IBM Rational Software Modeler pour Eclipse 3.1. Rational Software Modeler est un environnement de modélisation et de génération de code permettant le développement d'applications et la création d'outils guidés par les modèles. Nous avons réalisé le profil UML 2.0 pour COSA en utilisant le mécanisme de création du profil RSM (Rational Software Modeler). Rappelons que ce profil contient les stéréotypes et les valeurs marquées : cible, mode-connexion, type-service nécessaires à la génération des modèles UML. Afin de profiter des avantages des outils graphiques développés autour d'UML, la plupart des concepts architecturaux sont disponibles dans le composant *COSAPlugin* (ex. connecteurs de construction tels que *Attachement*, *Extends*, *Binding* et *Utilisation*, connecteurs définis par l'utilisateur, des structures telle que la configuration des composants et des connecteurs) (Alti, Khammaci, Smeda, 2007a ; Smeda, Alti, Boukerram, Oussalah, 2009).

COSAPlugin a été développé selon trois niveaux (métamodèle, modèle, application). Dans le niveau haut, le métamodèle COSA avec toutes les valeurs étiquetées et ses contraintes OCL 2.0, est défini par le profil UML 2.0. Ce profil joue un rôle important dans le second niveau quand il est utilisé par le modèle d'architecture logicielle. Une fois que nous nous assurons que le modèle donné est conforme aux contraintes sémantiques définies par le profil, un ensemble d'instances pour les types sont définis et évalués dans ce niveau. L'objectif principal *COSAPlugin* est de montrer la capacité d'appliquer le profil pour des applications complexes. Le plugin offre aux architectes la possibilité de vérifier la cohérence structurelle d'un système donné et de valider sa sémantique avec l'approche COSA. La figure 5.2 présente un aperçu de l'outil *COSAPlugin*.

L'outil *COSAPlugin* inclut les contraintes OCL 2.0, nécessaires à la définition de la transformation COSA vers UML 2.0 et le profil COSA qui contient un ensemble de stéréotypes avec toutes ses valeurs marquées nécessaires à la spécification des correspondances. *COSAPlugin* offre les fonctionnalités suivantes:

- Performance de vérification des modèles d'architecture UML, profilé COSA.
- Simplicité de création et rapidité de modification d'une architecture logicielle par la création du modèle UML 2.0 (diagramme de composants) et ensuite, ajout des contraintes OCL, nécessaires au modèle, après que, le modèle soit évalué par le profil.
- Validation sémantique et vérification de la cohérence structurelle des modèles d'architecture UML 2.0, profilé COSA contre les violations des contraintes du profil.
- Capacité aux architectes de construire facilement des modèles d'architecture UML 2.0 avec l'approche COSA.
- Habilité aux concepteurs de réutiliser et de partager les mêmes concepts architecturaux, durant tout le cycle de vie du système.

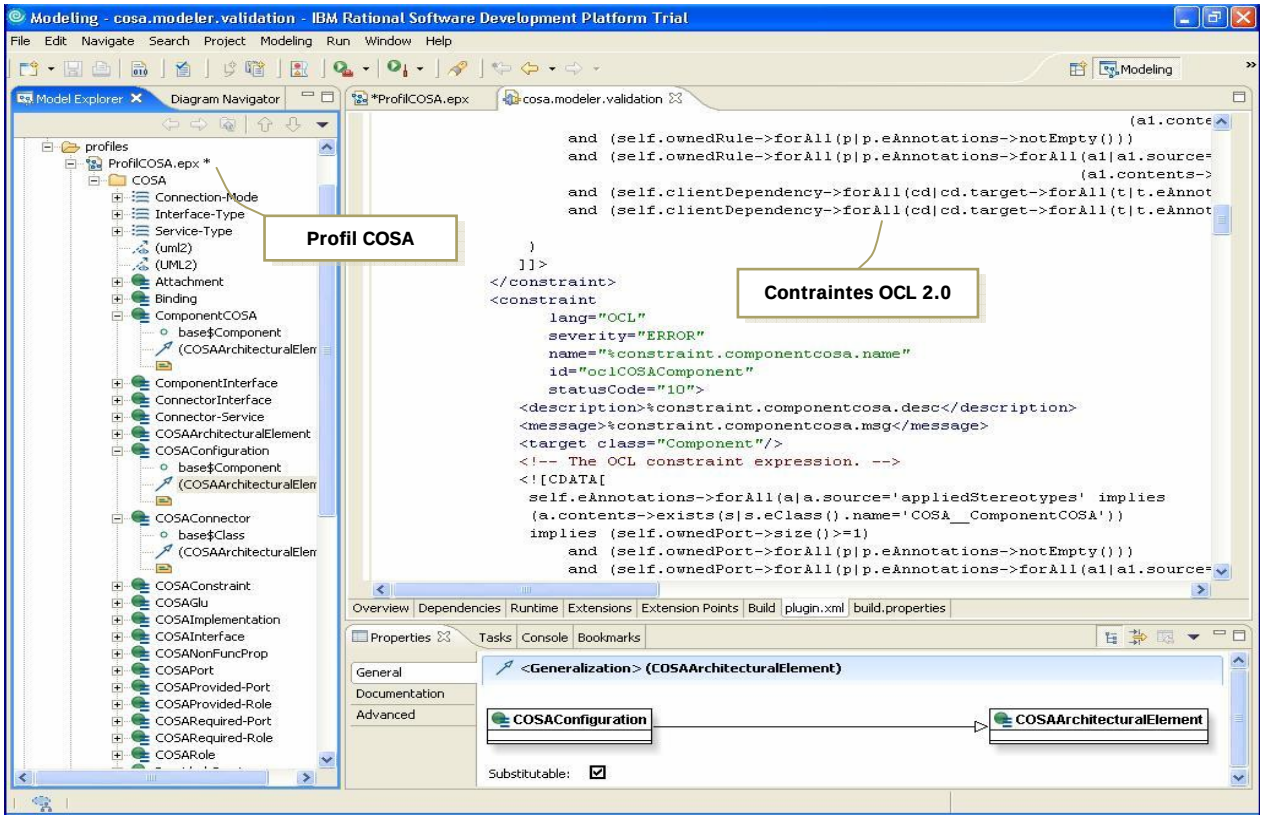


Figure 5.2 *COSAPlugin* sous *Eclipse 3.1*

5.2.2 Transformation COSA-UML (PSM) en eCore (PSM)

Le métamodèle COSA fournit un niveau d'abstraction haut qui permet de spécifier, d'analyser et de manipuler des architectures logicielles. En plus, l'approche de l'ingénierie de modèle (Frankel, 2003 ; Bézivin, 2005) facilite l'automatisation des modèles. Donc, on a besoin d'une stratégie qui permet de générer rapidement des outils graphiques pour la manipulation et la définition des concepts COSA. Plusieurs approches ont été proposées. La plupart des approches permettent le développement manuel des environnements de modélisation et des interfaces graphiques, GEF (Graphical Editing Framework) (GMF, 2006). Le développement manuel des environnements pour les modèles d'architectures n'est pas feusable. Par conséquent, il est devient nécessaire d'automatiser le développement de ces environnements.

COSA définit des abstractions d'architectures, mais il lui reste à supporter la distinction entre un modèle d'architecture (Niveau A_1) et son application (Niveau A_0). Puisque notre objectif est de créer rapidement un outil d'implémentation de métamodèle COSA, nous avons opté pour l'approche « profil UML ». Le but est donc est d'élaborer un métamodèle eCore correspondant parfaitement au métamodèle COSA à l'aide du profil UML, qui va être généré par la transformation COSA-UML vers eCore.

5.2.2.1 Pourquoi choisir eCore pour la transformation ?

EMF (Eclipse Modeling Framework) est un Framework Open Source qui permet de définir des métamodèles (i.e. UML) et de générer, des interfaces dédiées à ces métamodèles, un éditeur graphique et un ensemble de classes permettant de réaliser des applications à partir d'un métamodèle. Le cœur du framework EMF inclue un méta-méta-modèle (eCore). Depuis sa création, eCore n'a cessé d'accroître son influence dans le monde industriel. Les principaux concepts d'eCore sont présentés dans la Figure 5.3.

Nous avons choisi eCore, puisque c'est un MOF, conçu pour être comme l'OMG MOF. Nous pouvons récapituler les raisons qui nous ont fait choisir eCore dans les points suivants :

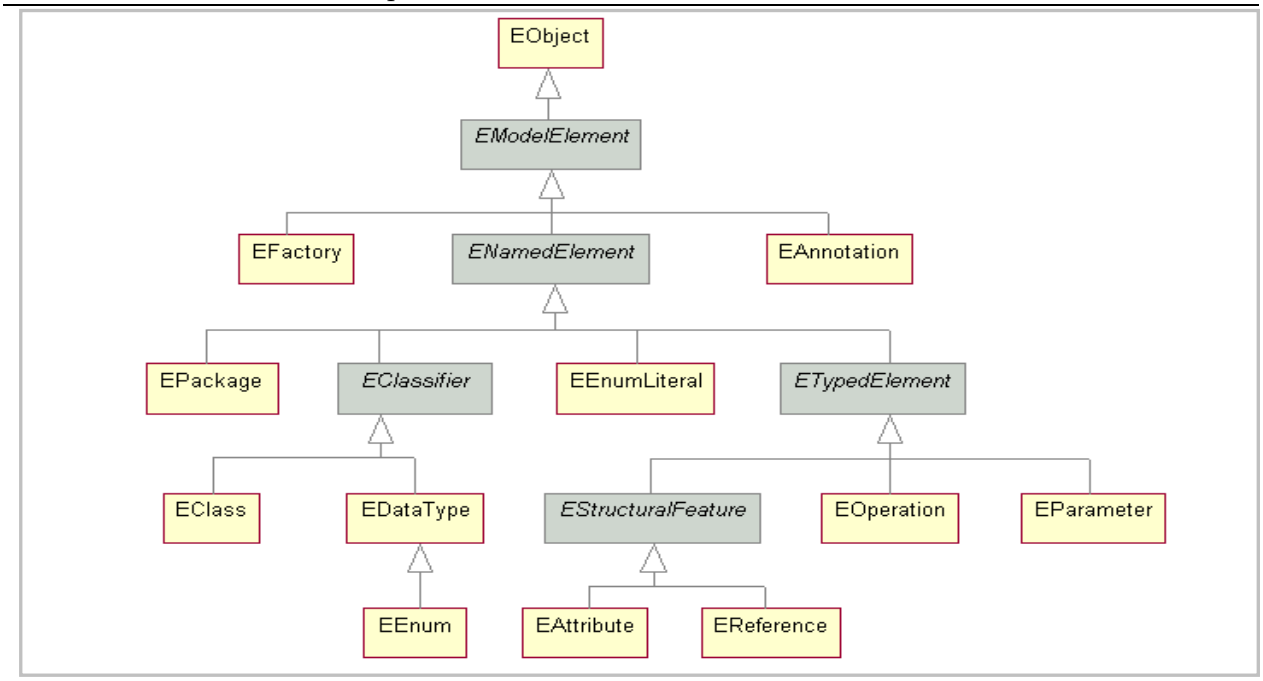


Figure 5.3 *Diagramme de classe de méta-méta-modèle eCore.*

- MOF est un modèle standard bien défini et basé sur le méta-méta-modèle MOF. eCore est l'implémentation de MOF.
- eCore est largement utilisé dans le monde industriel.
- eCore est apprécié dans le développement et la manipulation des modèles (tels que MDA, transformation des modèles, profils, etc.).
- eCore est supporté par plusieurs outils qui offrent diverses facilités, notamment la génération des API et des éditeurs graphiques.

Comme eCore est l'implémentation de MOF, il n'est pas aussi riche qu'UML. Le profil UML pour COSA inclus des éléments architecturaux qui ne sont pas explicitement définis dans eCore (composants, connecteurs, configurations, rôles, glu…). Cela permet de répondre aux lacunes du modèle technologique (eCore) au regard des langages de description d'architectures (ex. COSA) : 1) - absence des méta-entités représentant les architectures, les composants et les connecteurs, 2) - absence des relations explicites entre une architecture et son type, entre un connecteur et son type, entre un composant et son type, 3)- le profil UML pour COSA a plusieurs contraintes, il peut être appliqué aux éléments architecturaux COSA, eCore ne supporte pas directement ces contraintes.

5.2.2.2 Implémentation de la transformation COSA vers eCore

Nous avons utilisé la passerelle profil COSA UML qui jouera le rôle de pivot (Alti, Khammaci, Smeda, Bennouar, 2007) pour élaborer rapidement des outils graphiques pour l'ADL COSA. Le premier concerne un outil de modélisation graphique COSA pour lequel le modèle d'architecture COSA est élaboré et le deuxième concerne un outil d'instanciation COSA dans lequel le modèle d'architecture est instancié. Nous insistons sur la séparation des préoccupations de l'architecture de l'application. Cela permet de les réutiliser de façon indépendante. Donc, on peut aussi réutiliser la représentation graphique afin de produire plusieurs modèles d'instances pour différents modèles d'architectures.

Notre stratégie d'implémentation est réalisée en six étapes (voir figure 5.4) :

1. Projection des éléments du métamodèle COSA en eCore à l'aide du profil UML
2. Génération de l'outil graphique COSABuilder
3. Modélisation et validation de l'architecture COSA en utilisant COSABuilder
4. Projection des éléments des modèles COSA dans eCore via le profil UML
5. Génération de l'éditeur d'instances COSA-eCore
6. Sélectionner et instancier les éléments du modèle d'architecture COSA.

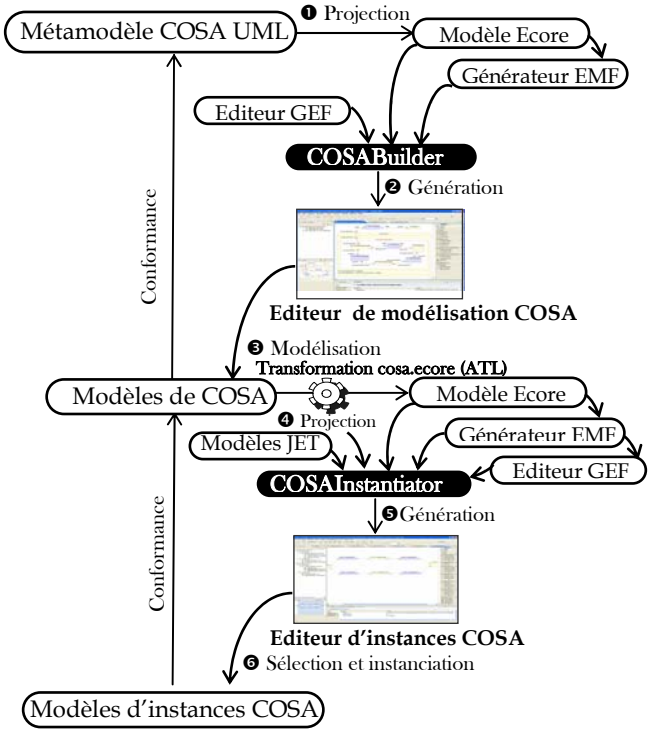


Figure 5.4. *Implémentation du métamodèle COSA vers eCore.*

1. Les règles de passage des concepts de COSA vers eCore

La transformation du modèle COSA vers celui d'eCore doit suivre un processus progressif qui transforme la plupart des éléments architecturaux abstraits en de nouveaux éléments architecturaux concrets. Nous proposons de transformer chaque concept architectural du modèle COSA à l'aide du profil UML 2.0.

Le processus de projection est facilement achevé par l'adaptation de la transformation UML-eCore (Eclipse, 2007), pour projeter les contraintes sémantiques du profil COSA UML, qui ne sont pas projetées parfaitement vers eCore. Les contraintes OCL propre au profil UML COSA sont projetées vers eCore comme des valeurs des attributs non conformes au métamodèle eCore. Pour cela, il est donc nécessaire de transformer toutes les contraintes OCL du profil COSA UML vers eCore afin de respecter les propriétés sémantiques propres au profil COSA-UML. Le choix est porté sur les classes eCore pour la représentation des composants, des connecteurs et des configurations COSA, mais ces derniers restent bien distincts grâce aux annotations qu'on leur associe. Le tableau 5.1 résume la correspondance entre les concepts de COSA et ceux d'UML 2.0 et le tableau 5.2 résume les contraintes OCL de la transformation COSA vers eCore.

Concepts UML 2.0	Concepts eCore
«COSAArchitecturalElement» Classe	«COSA__COSAArchitecturalElement» Classe
Attribut	Attribut
«COSAPropNonFunc» Attribut	«COSA__COSAPropNonFunc» Attribut
«COSAConstraint» Contrainte	«COSA__COSAConstraint» Classe avec deux attributs : nom et corps de contrainte
«COSAImplementation» Classe	«COSA__COSAImplementation» Classe
«COSAComponent » Composant	«COSA__COSAComponent » Classe
«COSAConnector » Classe	«COSA__COSAConnector » Classe
«COSAComponent » Composant	«COSA__COSAComponent » Classe
«ComponentInterface» Port	«COSA__ComponentInterface» Classe
«ConnectorInterface» Port	«COSA__ConnectorInterface» Classe
«COSAPort» Interface	«COSA__COSAPort» Classe
«COSARole» Interface	«COSA__ COSARole» Classe
«COSAGlu» Classe d'Association	«COSA__COSAGlu» Classe
«Service» Opération	«COSA__Service» Classe
«Connector-Service» Opération	«COSA__Connector-Service» Classe
«Attachement» Connecteur d'assemblage	«COSA__Attachement» Classe
«Binding» Connecteur de délégation	«COSA__Binding» Classe
«Use» Association	«COSA__Use» Classe

Table 5.1 *Correspondance COSA – eCore.*

Concepts COSA	Contraintes OCL
Elément-Architectural	Un classe Ecore annoté « ArchitecturalElement » ayant seulement des propriétés et des contraintes COSA.

Composant (type)	Une classe Ecore annoté « COSAComponent » hérite d'un élément architectural COSA.
	`context eCore::EClass -- Specialized Element`
	`self.owner->oclIsKindOf(ArchitecturalElement)`

Interface Composant	InterfaceComposant est définie seulement par COSAComponent ou COSAConfiguration.
	`context eCore::EClass -- ComponentIntf`
	`self.owner->oclIsKindOf(component) or`
	`self.owner->oclIsKindOf(configuration)`

Port	Chaque port doit avoir un attribut énuméré Mode : Mode-connexion

Connecteur (type)	Un connecteur COSA contient un glu ou une configuration.
	`context eCore::EClass -- detailOrGlue`
	`not(self.detail->notEmpty() and`
	`not self.glue.oclIsUndefined()) and`
	`self.detail->size() <= 1`

Interface Connecteur	Interface connecteur définit seulement par un connecteur.
	`context eCore::EClass -- ConnectorInterface`
	`self.owner->oclIsKindOf(UserConnector)`

Rôle	Chaque rôle doit avoir un attribut énuméré Mode : Mode-connexion
Glu	
Configuration (Type)	Les ports d'une configuration héritent d'une InterfaceComposant.

Attachement	Un attachement relie un port requis (rôle requis) avec un port fourni (rôle fourni).
	`context eCore::EClass -- Attachment`
	`(self.source.oclIsKindOf(RequiredPort) and`
	`self.target.oclIsKindOf(ProvidedRole)) or`
	`(self.source.oclIsKindOf(ProvidedPort) and`
	`self.target.oclIsKindOf(RequiredRole)) or`
	`(self.source.oclIsKindOf(RequiredRole) and`
	`self.target.oclIsKindOf(ProvidedPort)) or`
	`(self.source.oclIsKindOf(ProvidedRole)and`
	`self.target.oclIsKindOf(RequiredPort)`

Binding	Binding relie deux rôles fournis (ou requis) de même type.
	`context eCore::EClas -– Binding`
	`(self.source.oclIsKindOf(RequiredPort) and`
	`self.target.oclIsKindOf(RequiredPort)) or`
	`(self.source.oclIsKindOf(ProvidedPort) and`
	`self.target.oclIsKindOf(ProvidedPort)) or`
	`(self.source.oclIsKindOf(RequiredRole) and`
	`self.target.oclIsKindOf(RequiredRole)) or`
	`(self.source.oclIsKindOf(ProvidedRole) and`
	`self.target.oclIsKindOf(ProvidedRole))`

Use	Use est une association entre un port/rôle et un service.
	`context eCore::EClass -- use`
	`self.source.oclIsKindOf(Port)and self.target.oclIsKindOf(Service))`
	`or(self.source.oclIsKindOf(Service)and self.target.oclIsKindOf(Port))`

Propriété	Chaque propriété ayant un nom, type et valeur. Elle est appliquée seulement sur des composants architecturaux COSA (composant, connnecteur et configuration).
Contrainte	Chaque contrainte ayant un nom et un corps.

Table 5.2 *Les Contraintes OCL COSA vers eCore.*

La définition des transformations en JAVA permettant la transformation d'un modèle COSA en un modèle eCore nécessitent des expressions OCL (Object Management Group, 2005), générées manuellement à partir de la spécification des correspondances.

Elément architectural

L'élément architectural est un concept qui définit tous les concepts architecturaux de COSA. Ce concept n'a pas de correspondant explicite dans eCore. Ainsi, le métamodèle COSA-eCore doit inclure une annotation pour le représenter. Il correspond à la métaclasse EClass du métamodèle eCore annoté par *COSA__COSAArchitecturalElement.* Une classe eCore peut avoir des propriétés et des contraintes et peut être implémentée par une autre classe.

Composant

Chaque composant UML stéréotypé «*COSAComponent*» correspond à une classe eCore annotée *COSA__COSAComponent*. Tout composant annoté *COSA__COSAComponent* est une spécialisation de l'élément architectural COSA. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context eCore::EClass
inv :self.owner->oclIsKindOf(ArchitecturalElement)
```

Connecteur

Toute classe stéréotypée « *COSAConnector* » correspond à une classe eCore annotée *COSA__COSAConnector*. Chaque classe eCore annotée *COSA__COSAConnector* doit possède une seule glue ou une configuation mais pas les deux en même temps. Cette contrainte peut se définir dans OCL de la façon suivante :

```
context eCore::EClass
inv : not(self.detail->notEmpty() and not self.glue.oclIsUndefined())
      and self.detail->size() <= 1
```

Configuration

Un aspect important de l'architecture COSA est celui de la configuration qui est un graphe de composants et de connecteurs. Comme un package eCore peut contenir des sous-classes, donc les configurations COSA sont projetées vers un package eCore

Port et rôle

Chaque port annoté *COSA__ConnectorInterface* est défini seulement par un COSAConnector et possède un ensemble non vide de rôles COSA. Une classe annotée *COSA__COSAProvided-Port (COSA__COSARequired-Port)* est définie seulement par une classe annotée *COSA*__ComponentInterface.

Attachment

Une classe eCore annotée COSA__Attachment est équivalent au concept connecteur d'assemblage COSA qui est défini entre un port et un rôle COSA. Le port du connecteur COSA, de type fourni est relié avec le rôle COSA, de type requis ou le port COSA de type requis, est relié avec le rôle COSA de type fourni, avec la contrainte OCL suivante :

```
context eCore::EClass
inv : (self.source.oclIsKindOf(RequiredPort) and
       self.target.oclIsKindOf(ProvidedRole)) or
       (self.source.oclIsKindOf(ProvidedPort) and
       self.target.oclIsKindOf(RequiredRole)) or
       (self.source.oclIsKindOf(RequiredRole) and
       self.target.oclIsKindOf(ProvidedPort)) or
       (self.source.oclIsKindOf(ProvidedRole) and
       self.target.oclIsKindOf(RequiredPort)
```

Binding

Binding COSA relie deux ports COSA fournis (ou requis) de deux différents sous composants d'un même composant COSA avec la contrainte OCL suivante :

```
context  eCore::EClas
inv :(self.source.oclIsKindOf(RequiredPort) and
       self.target.oclIsKindOf(RequiredPort)) or
      (self.source.oclIsKindOf(ProvidedPort) and
       self.target.oclIsKindOf(ProvidedPort)) or
      (self.source.oclIsKindOf(RequiredRole) and
       self.target.oclIsKindOf(RequiredRole)) or
      (self.source.oclIsKindOf(ProvidedRole) and
       self.target.oclIsKindOf(ProvidedRole))
```

Use

Un connecteur d'association UML stéréotypé correspond au concept classe eCore, annotée COSA__Use. Le concept classe correspond au concept de Use dans COSA. Il représente le rattachement physique entre un port/rôle et un service. Use COSA est défini entre un port COSA et un Service COSA ou entre un rôle COSA et un Service-Connecteur COSA avec la contrainte OCL suivante :

```
context eCore::EClass
(self.source.oclIsKindOf(Port)and self.target.oclIsKindOf(Service))or
(self.source.oclIsKindOf(Service)and self.target.oclIsKindOf(Port))
```

Propriétés

Les propriétés fonctionnelles de COSA sont équivalentes au concept d'attributs eCore alors que les propriétés non fonctionnelles sont représentées par des attributs annotés *COSA__COSANonFoncProp* dans la mesure où elles sont différentes au sens sémantique, puisque les attributs en eCore sont des propriétés structurelles qui constituent une classe. Chaque attribut possède un type, auquel on peut donner une valeur pour l'instance de l'élément architectural.

Contraintes

Les contraintes COSA-UML exprimées via le concept EClass de métamodèle eCore sont utilisées pour documenter les contraintes de COSA. Chaque classe eCore contient deux attributs : nom de la contrainte et son corps OCL.

2. Génération de l'outil de modélisation graphique COSABuilder

Afin de profiter des avantages des outils graphiques développés autour eCore et GMF (Graphic Modeling Framework) (GMF, 2006), nous avons développé un outil graphique pour COSA, qu'on a baptisé *COSABuilder* (Smeda, Alti, Oussalah, Boukerram, 2009 ; Alti, Boukerram, Smeda, Maillard, Oussalah, 2010). La plupart des concepts architecturaux sont disponibles dans *COSABuilder* (ex. connecteurs de construction tels que *Attachement*, *Extends*, *Binding* et *Utilisation*, connecteurs définis par l'utilisateur, des structures telle que configuration des composants et des connecteurs). Cet environnement permet aux architectes de définir graphiquement les modèles d'architectures. La figure 5.5 présente l'interface graphique de COSABuilder.

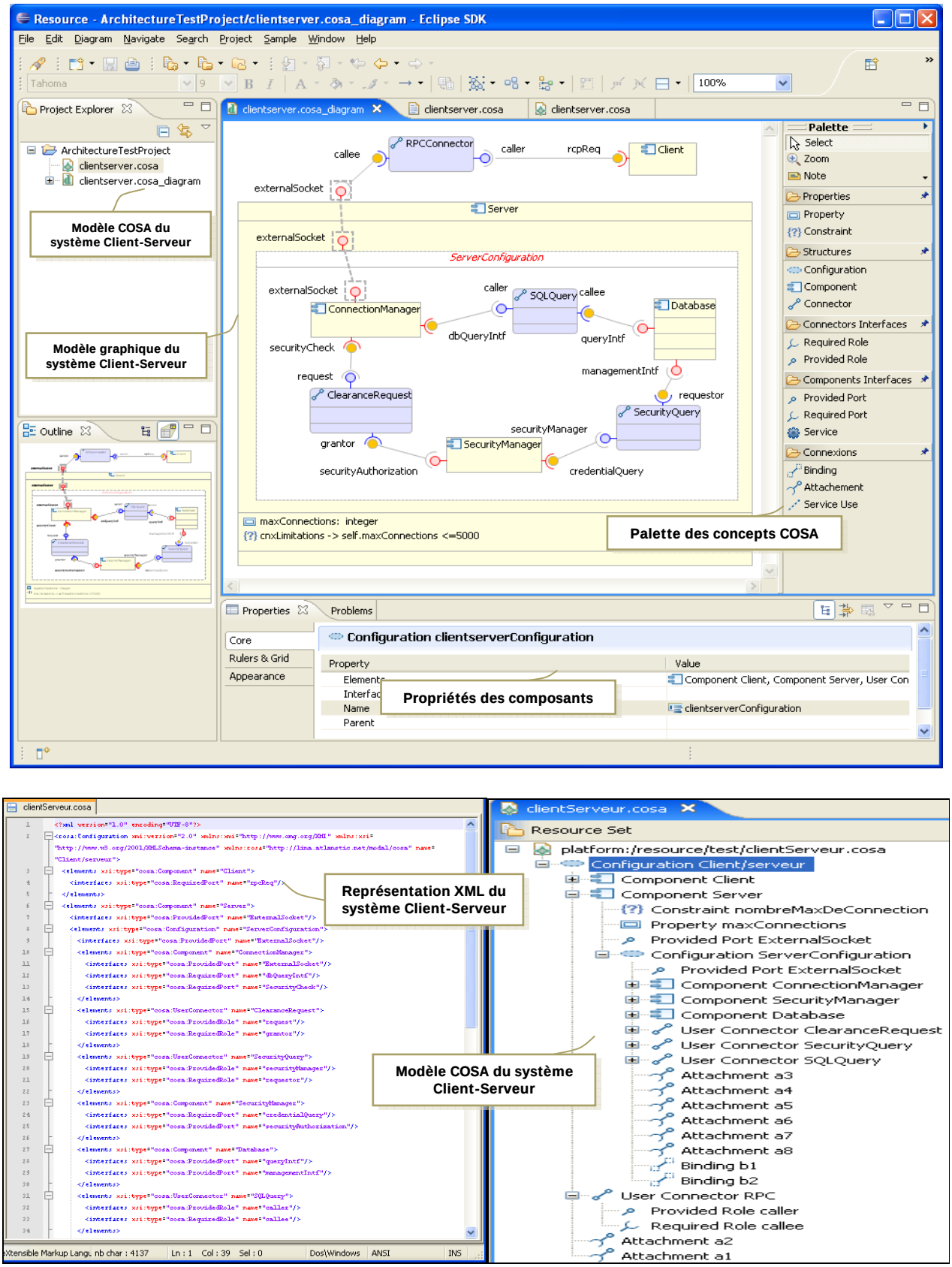


Figure 5.5. *Les différentes vues d'une seule architecture Client/serveur*

3. Modélisation et validation des architectures avec COSABuilder

Dans cette phase, le diagramme de composants COSA décrit un système par des composants reliés entre eux par des connecteurs définis par l'utilisateur ou par des connecteurs de construction. COSABuilder fournit un environnement de modélisation visuel où chaque artefact possède une représentation graphique. Par exemple, les composants composites et les connecteurs composites sont modélisés en utilisant les configurations COSA.

À partir de ce modèle, nous avons appliqué la validation structurelle des modèles d'architecture COSA annoté eCore, contre les violations des contraintes OCL définies par le métamodèle COSA-eCore. Les contraintes OCL définissent les relations entre les éléments de modèle COSA, par exemple le connecteur Binding relie deux ports COSA fournis (ou requis) de deux différents sous composants d'un même composant COSA.

4. Transformation des modèles COSA profilé UML vers eCore

eCore est l'implémentation du MOF. Il peut être utilisé pour définir des métamodèles (UML par exemple). En principe eCore peut être utilisé pour définir un métamodèle d'une architecture logicielle. Cependant, eCore souffre d'un certain nombre des lacunes relatives à la description architecturale. En outre, le concept EClass peut être utilisé pour représenter le concept du composant. Il n'existe pas un concept explicite qui permet de représenter par exemple une instance d'un composant ou une instance d'un connecteur. Par conséquent, il n'y a pas de relation entre un composant et son type ou un connecteur et son type.

En résumé, eCore n'a pas une relation qui puisse être utilisée pour indiquer que l'architecture est définie par une méta-architecture (c'est à dire tous les éléments architecturaux ne peuvent pas être attribués à leur type dans cette méta-architecture).

L'idée d'achever la projection des modèles COSA en eCore, est de prendre chaque concept de métamodèle COSA et essayer de trouver son correspondant dans le métamodèle eCore. Il peut être élaboré de deux manières différentes: manuelle, par la transformation COSA2eCore avec le langage JAVA, ou automatique, par la transformation COSA2eCore avec le langage Java et ATL (ATLAS, LINA, 2007). ATL

permet d'exprimer les transformations des modèles de manière plus simple. En effet, c'est un langage dédié à cette tâche et qui offre deux volets: un volet déclaratif et un volet impératif. La partie impérative garantit le pouvoir de résoudre tous les problèmes en fournissant des constructions classiques (procédures, boucles, etc.) Mais, il 'est recommandé de s'en tenir autant que possible au volet déclaratif.

Le principe de la transformation COSA vers eCore est basé sur la projection de chaque élément du modèle COSA vers son correspond eCore. Le tableau 5.3, illustre la correspondance entre les concepts de modèle COSA et ceux d'eCore, qui nous permettent d'élaborer les règles de transformation entre COSA et le modèle eCore. La transformation peut être étendue à d'autres modèles cibles (par exemple, MOF 2,0). La définition des transformations en JAVA permettant la transformation d'un modèle COSA en un modèle eCore, nécessite des expressions ATL (ATLAS group LINA, INRIA Nantes, 2007), basées sur la spécification de correspondances présentées précédemment. Les règles de transformation en OCL 2.0 sont générées manuellement à partir de la spécification des correspondances (Voir Annexe A).

Concepts UML 2.0	Concepts eCore
Instance Composant **« COSAConfiguration »**	Package annoté COSA__COSAConfiguration si une configuration principale sinon une instance d'une classe annotée COSA__ COSAConfiguration.
Instance Composant **« COSAComponent »**	instance d'une classe annotée COSA__ COSAComponent
Instance Classe **« COSAConnector »**	instance d'une classe annotée COSA__ COSAConnector
Connecteur de construction	Référence
Instance d'un port requis	instance d'une classe annotée COSA__ Required-Port
Instance d'un port fourni	instance d'une classe annotée COSA__ Provided-Port
Instance d'un rôle requis	instance d'une classe annotée COSA__ Required-Role
Instance d'un rôle fourni	instance d'une classe annotée COSA__ Provided-Role

Table 5.3 *Correspondance COSA instance – eCore.*

5. Le générateur COSAInstantiator

En utilisant GMF, nous avons implémenté un outil complet d'instanciation des applications appelé *COSAInstantiator,* qui se base sur le métamodèle instance COSA. Deux façons permettent de générer l'éditeur graphique : 1) - à partir d'un modèle existant COSA-Ecore sous forme de fichier XMI ou 2) - à partir d'une nouvelle architecture COSA en tant que fichier COSA-UML. L'élément clé de *COSAInstantiator* est la capacité à créer rapidement des architectures des applications et aussi introduire un support explicite pour préserver un lien de traçabilité entre l'architecture de l'application et son espace de modélisation. Plusieurs architectures peuvent être instanciées à partir d'un modèle d'architecture. Nous avons obtenu le modèle eCore à partir du modèle d'architecture COSA-UML et le concept EAnnotation de métamodèle eCore, qui décrit les informations sur le méta-type (tous les éléments COSA d'une architecture peuvent être attribués à leurs types, dans la méta-architecture COSA en utilisant les annotations eCore).

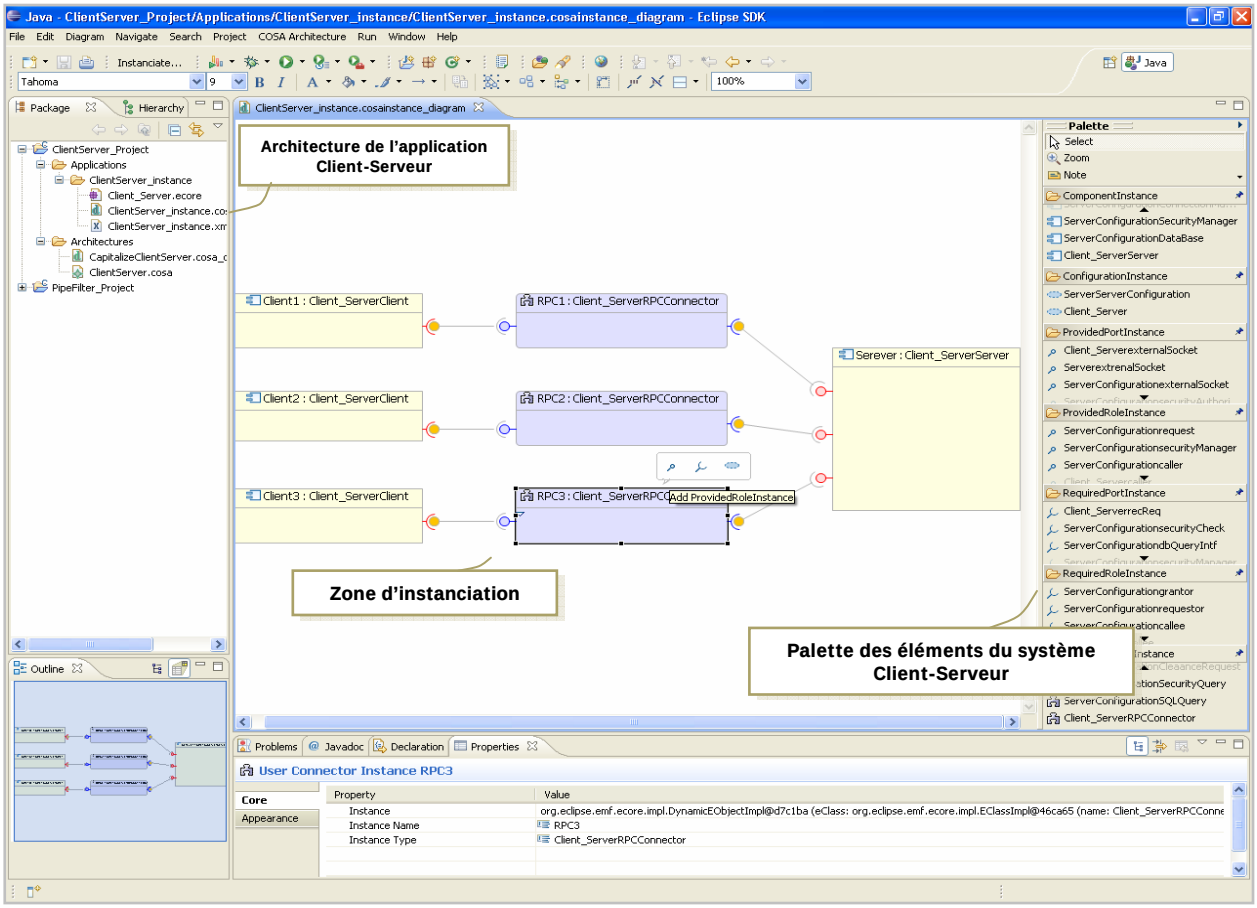


Figure 5.6. *Modèle d'interface utilisateur pour l'outil COSAInstantiator.*

La figure 5.6, présente un aperçu de l'éditeur graphique de l'architecture client-serveur avec les différentes vues: Editeur d'instances COSA avec sa palette, la vue console, et la vue Propriétés. Si l'on regarde la palette, nous allons découvrir que tous les éléments du modèle d'architecture client-serveur sont présentés, de la même sorte que nous venions de faire glisser, l'élément approprié dans la zone qui contient le diagramme d'application client-serveur pour décrire une application complète.

6. Elaboration des architectures des applications

Dans la dernière phase du processus d'instanciation, le développeur a la possibilité de sélectionner et d'instancier des éléments visuels à chaque fois qu'il en a besoin pour décrire une application complète. Les instances sont créées à partir de types d'éléments qui sont obtenus par la projection de COSA vers eCore, lors de la phase précédente.

5.2.3 Transformation COSA-UML (PSM) vers CORBA (PSM)

Parmi les apports de l'intégration de COSA dans MDA, on distingue la séparation des aspects architecture des aspects implémentation, la séparation entre la spécification des correspondances et les règles de transformation. Dans cette section, nous présentons la transformation d'un modèle d'architecture COSA-UML (PSM) vers un autre modèle d'implémentation CORBA (PSM). Ce modèle contient tous les concepts CORBA correspondant aux concepts COSA du modèle source. A partir de ce modèle, nous appliquons la génération des interfaces IDL pour obtenir un squelette des interfaces CORBA.

Nous avons choisi le standard international CORBA qui simplifie le développement de logiciels par rapport aux autres plates-formes. Il supporte l'hétérogénéité et il facilite la communication et la coordination des composants distribués. Nous détaillons la spécification des correspondances COSA-CORBA et nous définissons les règles de transformation COSA-CORBA.

5.2.3.1 CORBA : Standard des plates–formes d'exécution

CORBA[5] décrit des architectures et définit des spécifications pour le traitement d'objets distribués sur le réseau. CORBA est une plate-forme d'exécution standard et internationale. Il fournit une simplicité de développement par rapport aux autres plates-formes (EJB, .NET,… etc.) par le biais des IDLs (Interface Definition Language). Le modèle de composants CORBA est un modèle multi-interface, multi-langage avec IDL (Interface Definition Language).

Les composants CORBA sont interopérables avec les composants EJB. CORBA intègre aussi, des descripteurs pour la configuration, la définition de l'assemblage et le déploiement des composants.

Les éléments de métamodèle CORBA (Object Management Group, 2002), sont les concepts utilisés pour spécifier des composants CORBA. Les concepts clés du type de composant sont les ports. Un composant peut avoir un ou plusieurs attributs qui représentent ses propriétés configurables.

Les *ports* dans CORBA fournissent ou utilisent des services de composants. Le port est principalement représenté par des interfaces. Les *interfaces* de type «Fourni» ou «Requis» sont basés sur l'interface IDL (Interface Description Language). Deux types d'interfaces fournis (requis) : une facette (réceptacle) pour les mécanismes de communication synchrone et puis d'événements (source d'événements) pour le mécanisme de communication asynchrone. Les facettes et les composants ont le même cycle de vie. Chaque interface a sa propre référence de services. Les connections sont utilisées pour l'assemblage des composants. Une *maison* (Home) est un gestionnaire des instances de composants qui fournit des opérations de création. Elle fournit aussi des opérations de recherche pour les composants CORBA existants. Une maison possède ses propres attributs et ses propres opérations.

[5] CORBA est un nom déposé par l'Object Management Group (OMG)

5.2.3.2 Pourquoi la transformation de COSA – UML vers CORBA ?

CORBA est une plate-forme d'exécution standard offrant une simplicité de développement par assemblage des composants préfabriqués, mais il ne supporte pas des niveaux d'abstraction élevés comme les connecteurs et les composants composites. Etant donné que COSA supporte les composants composites et définit explicitement les connecteurs, comme des concepts architecturaux abstraits, il est utile de définir une transformation automatique de modèles d'architectures vers l'implémentation. L'intérêt de cette transformation est la projection rapide et une meilleure intégration des concepts COSA dans la plate-forme CORBA afin d'aboutir à un niveau d'abstraction plus élevé et d'aider ainsi, à résoudre les problèmes d'interactions entre les composants CORBA.

La transformation du modèle COSA vers celui de CORBA est un processus automatique qui produit des concepts CORBA, de ses correspondants COSA. Ce processus définit un modèle UML conforme au métamodèle COSA, il produit un modèle UML vers la plate-forme cible CORBA. Ce modèle est validé par le profil UML pour CORBA (Object Management Group, 2004b).

5.2.3.3 Profil UML 1.4 pour CORBA

Le but du profil UML 1.4 pour CORBA est le développement des applications CORBA en utilisant des notations UML, pour faciliter la génération de code des composants CORBA, et la spécification complète des concepts CORBA (Object Management Group, 2002). Ce profil est décrit comme un package UML stéréotypé nommé «*CORBA*» comme illustré dans la figure 5.7. Ce package inclut une classe stéréotypée «*CORBAComponent*» pour représenter un composant CORBA. Un composant CORBA peut avoir des attributs et des interfaces. Les facettes et les réceptacles sont représentés respectivement par des associations de composition stéréotypées «*CORBAUses*», «*CORBAProvides*» entre «*CORBAComponent*» et «*CORBAInetrface*». En plus, une source d'événements, un puit d'événements et une publication d'événements sont représentées respectivement par des associations de composition stéréotypées «*CORBAEmits*», «*CORBAConsumes*» et «*CORBAPublishes*» entre «CORBAComponent» et «*CORBAEvent*». Un Home est décrit par une classe stéréotypée «CORBAHome». Ces stéréotypes sont documentés en utilisant des valeurs marquées et des contraintes OCL.

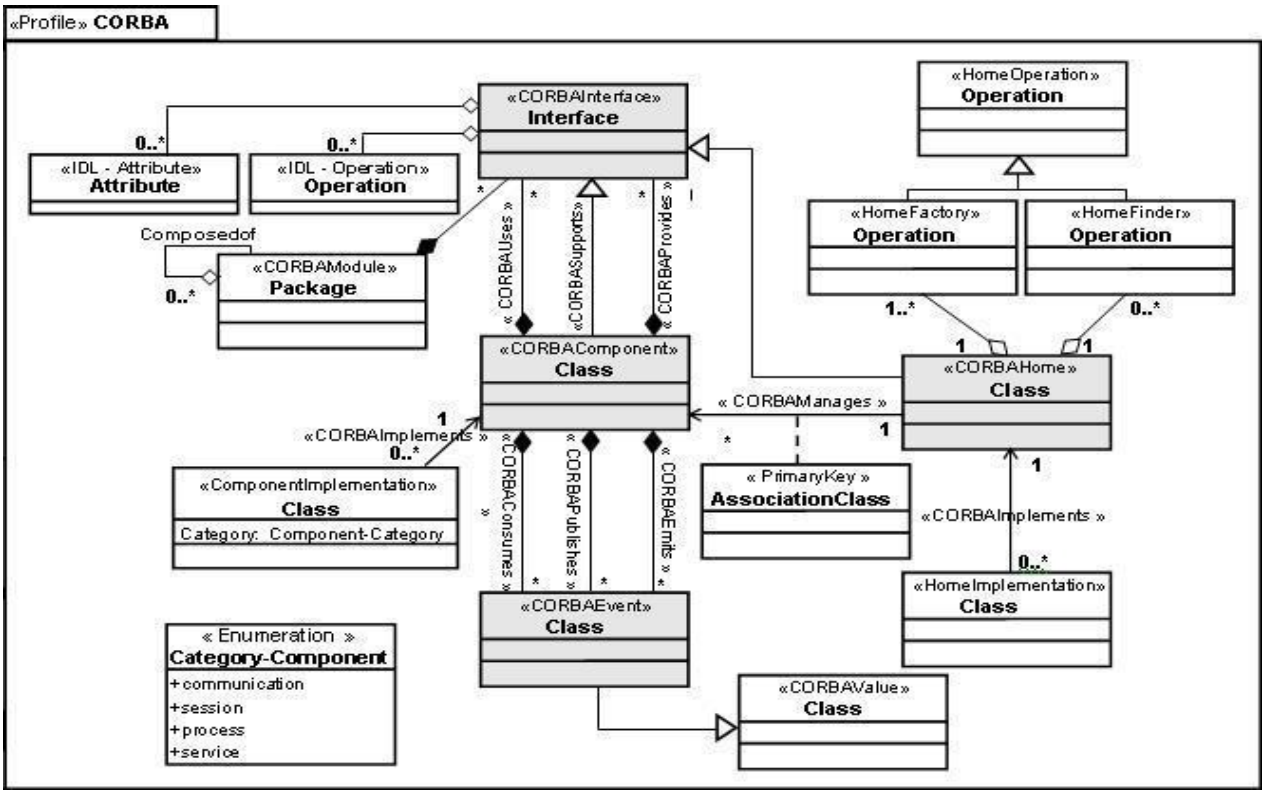


Figure 5.7 *Le profil CORBA*

5.2.3.4 Les règles de passage des concepts COSA vers CORBA

Nous proposons de transformer chaque concept architectural du modèle COSA vers CORBA à l'aide du profil UML 1.4 pour CORBA (Object Management Group, 2004b). Les composants COSA, sont représentés par des composants UML 2.0. A chaque composant UML 2.0, correspond une classe UML 1.4 (la classe, porte le même nom que le composant), le composant UML 2.0 «COSAComponent» peut être transformé vers une classe UML «CORBAHome». Les connecteurs COSA sont représentés par des classes UML 2.0. Les classes UML 2.0 correspondent aux classes UML 1.4. Ainsi, la classe «COSAConnector» est transformée vers une classe «CORBAHome». Les interfaces de composants et des connecteurs correspondent aux ports UML 2.0. Les ports correspondent aux classes UML (le nom de la classe correspond au nom du port). Chaque classe représentant un composant ou un connecteur (composants et connecteurs «CORBAHome») doit être rattachée à des associations d'agrégation vers les classes des ports (qui correspondent à une classe «CORBAComponent»).

Le tableau 5.4 montre la correspondance entre les concepts de COSA et ceux de CORBA, qui nous permet d'élaborer les règles de transformation entre COSA et la

plate-forme CORBA (Alti, Khammaci, Smeda, 2007b).

Concepts COSA UML 2.0	Concepts CORBA UML 1.4
«COSAArchitecturalElement» Classe	«CORBAHome» Classe
«COSAProp» Attribut	«IDL-Attribute» attribut
«COSAConstraint» Contrainte	«CORBAException»
«COSAImplementation» Classe	«CORBAImplementation»
«COSAComponent » Composant	«CORBAHome» Classe
«COSAConnector » Classe	
«COSAConfiguration» Composant	«CORBAHome» Package
«ComponentInterface» Port	«CORBAComponent» Classe
«ConnectorInterface» Port	
«COSAPort » Interface	«CORBAInterface» synchrone «CORBAEvent» asynchrone
«COSARole» Interface	
«COSAGlu» Classe d'Association	«IDL-Operation» Opération
«Service» Classe	«CORBAEvent» Classe
«Attachement» Connecteur d'assemblage	«CORBAComponent» Classe
«Binding» Connecteur de délégation	
«Use » Association	

Table 5.4 *Correspondance COSA – CORBA*

La définition des transformations en JAVA et (ATL) Atlas Transformation Language (ATLAS group LINA, INRIA Nantes, 2007), permettant la transformation d'un modèle COSA profil UML vers CORBA profil UML doit être basée sur la spécification de correspondances décrites précédemment. Chaque élément de correspondance contient une *expression OCL* (Object Management Group, 2005), permettant d'effectuer la *transformation* entre les éléments des profils UML pour COSA et CORBA, et un *filtre* pour faire la distinction entre eux. Nous *transformons* le filtre dans un *helper* '*hasStereotype*' exprimé dans OCL, comme suit:

```
helper context UML2!Element def : hasStereotype(name:String): Boolean =
     self.eAnnotations->forAll(a|a.source='appliedStereotypes' implies
          (a.contents->exists(s|s.eClass().name = name)));
```

Les concepts de base tels que les propriétés, les contraintes et les implémentations sont projetées dans d'autre *helper's (méthodes auxiliaires*) comme suit:

```
helper context UML2!Class def: getCOSAProps():Sequence(UML2!Property)=
  self.ownedAttribute->select(e|e.hasStereotype('COSAProp'));
helper context UML2!Class def:getCOSAConsts():Sequence(UML2!Constraint)=
  self.ownedRule->select(e|e.hasStereotype('COSAConstraints'));
```

```
Helper context UML2!Class def: getCOSAImps():Sequence(UML2!Class) =
  self.clientDependency.target(e|e.hasStereotype('COSAImplementation'));
```

L'élément architectural COSA a été représenté par une classe UML 2.0. Puisque, à une classe UML 2.0 correspond une classe UML 1.4 identique, la classe UML 2.0 «COSAArchitecturalElement», peut être transformée vers une classe UML «CORBAHome».

Les composants COSA sont représentés par des composants UML 2.0. Puisque, à un composants UML 2.0, correspond une classe UML 1.4 (la classe porte le même nom que le composant), le composant UML 2.0 «COSAComponent» peut être transformé vers une classe UML «CORBAHome».

Les connecteurs COSA définissent des abstractions qui englobent les mécanismes de communication, de coordination et de conversion entre les composants (Smeda, Khammaci, Oussalah, 2004a), qui ne sont pas définis explicitement dans la plate-forme CORBA. Donc, nous essayons de trouver les correspondances sémantiques des concepts CORBA. Les connecteurs COSA sont représentés par des classes UML 2.0. Les classes UML 2.0 correspondent aux classes UML 1.4, donc la classe «COSAConnector» est transformée vers une classe «CORBAHome».

Une glu est représentée par une classe d'association UML 2.0. La glu décrit les fonctionnalités attendues représentées par les parties en interaction. Elle n'est pas définie explicitement dans la plate-forme CORBA. Donc, nous essayons de trouver les correspondances sémantiques des concepts CORBA. Puisque les classes communiquent par l'envoi de messages (opération au sens UML), la classe d'association stéréotypée « GluCOSA » est transformée par une opération UML.

Les configurations sont des graphes de composants et de connecteurs. Elles ne sont pas définies explicitement dans la plate-forme CORBA. Donc, nous allons essayer de trouver les correspondances sémantiques des concepts CORBA. Les composants COSA sont représentés par des composants UML 2.0. Les composants UML 2.0 correspondent aux classes UML 1.4, donc le composant «COSAConfiguration» transformé vers une classe «CORBAModule».

Les interfaces de composants et des connecteurs correspondent aux ports UML 2.0. Les ports corresponds aux classes UML (le nom de la classe correspond au nom du port). Cette classe réalise les interfaces fournies du port. Cette classe dépend des interfaces requises par le port. Si, une classe UML, qui représente un composant UML 2.0, doit être reliée à d'autre classes, qui représentant des ports, alors chaque classe qui représentant un composant ou un connecteur (composants et connecteurs «CORBAHome») doit être rattachée à des associations d'agrégation vers les classes des ports (qui correspond à une classe «CORBAComponent»). L'annexe B.1 présente les règles de transformation en ATL (ATLAS group LINA, INRIA Nantes, 2007), selon la description du profil UML 2.0 pour COSA et le profil UML 1.4 pour CORBA.

5.2.3.5 Implémentation

La transformation COSA vers CORBA a été implémentée en Java et ATL sur la plateforme de développement logiciel Eclipse. Le plugin est développé en quatre étapes :

1. le métamodèle COSA (CORBA) avec tous ses tagged-values et ses contraintes OCL a été défini par le profil UML 2.0 (UML 1.4), exporté vers COSA.XML (CORBA.XML).

2. la transformation COSA-CORBA est élaborée. Cette transformation parcourt le modèle UML 2.0 source et recherche les éléments stéréotypés selon le profil COSA. Pour chaque élément identifié du modèle source, la transformation construit les éléments correspondant dans le modèle CORBA cible.

3. Le modèle COSA est défini par un diagramme de composant UML 2.0, complété par tous les stéréotypes et tagged-values nécessaires au modèle de composants puis évalué par son profil.

4. La transformation COSA vers CORBA configurée et exécutée. Le modèle CORBA est évalué par son profil.

Eclipse fournit un langage de définition des transformations des modèles ATL (ATLAS group LINA, INRIA Nantes, 2007). Pour la définition de la transformation COSA-CORBA, nous avons utilisé le langage ATL. La figure 5.8 présente les métamodèles à gauche qui sont COSA et CORBA, les règles de transformation sont à droite.

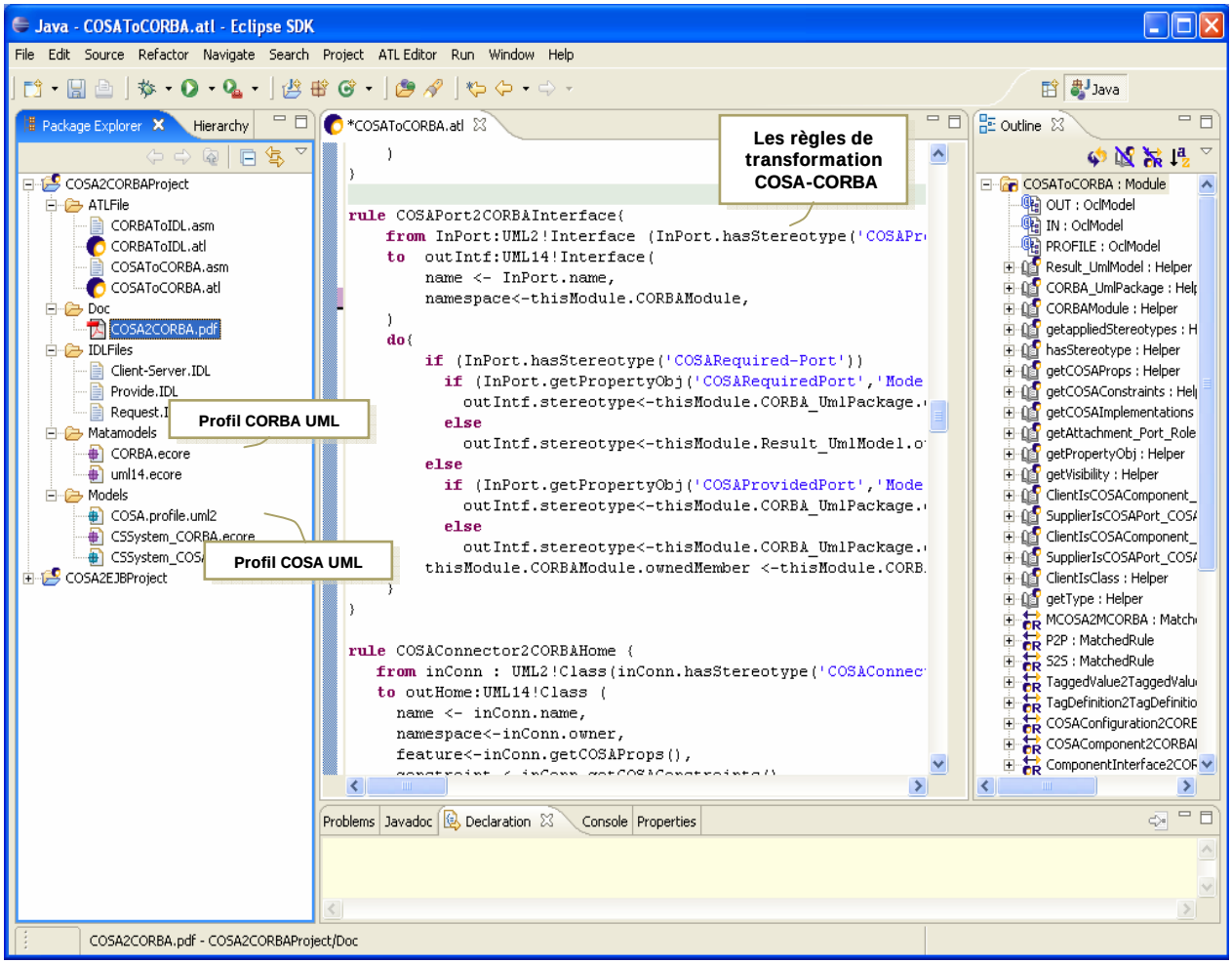


Figure 5.8 *Transformation COSA – CORBA.*

5.3 Etude de cas : Système Client-Serveur

Dans cette section, nous présentons un exemple pour illustrer l'intégration de l'architecture logicielle COSA au sein de MDA de notre stratégie de transformation. Nous avons choisi une application très répandue dans la communauté des architectures logicielles : un exemple de type Client-Serveur. Enfin, nous comparons les transformations COSA – eCore et COSA – CORBA.

5.3.1 Présentation du système Client-Serveur

Le système Client-Serveur est très répandu dans la communauté d'architecture logicielle. Il est traité explicitement par la plupart des ADLs. Il décrit un système avec deux composants (Client et Serveur) qui communiquent par l'intermédiaire d'un connecteur. Le composant Client possède un port requis (request) et un service requis

(Demand-data) et il a deux propriétés non fonctionnelles (data-type et request-rate). Le composant Serveur possède un port fourni (provide) et un service fourni (Send-data). Il possède deux propriétés non fonctionnelles (password-needed et data-type). Le composant Serveur est composé d'une configuration de trois composants (ConnectionManager, SecurityManager et DataBase) et de trois connecteurs (SQLQuery, ClearanceRequest et SecurityQuery). Pour plus de détails sur la configuration interne, on peut se référer à la section 3.2.2.3 du chapitre 3. Le connecteur RPC agit comme un médiateur auquel un client peut passer une requête et un serveur envoyé une réponse. Il possède un rôle fourni (participateur-1) et un rôle requis (participateur-2) et son type de service est *Conversion* (voir figure 5.9). Dans les prochaines sections, nous utilisons les outils de modélisation COSA pour représenter les éléments suivants:

- Configuration client-serveur,
- Composant client,
- Composant serveur,
- Configuration serveur,
- Composant gestion de connexion,
- Composant gestion de sécurité,
- Composant base de données,
- Application client-serveur.

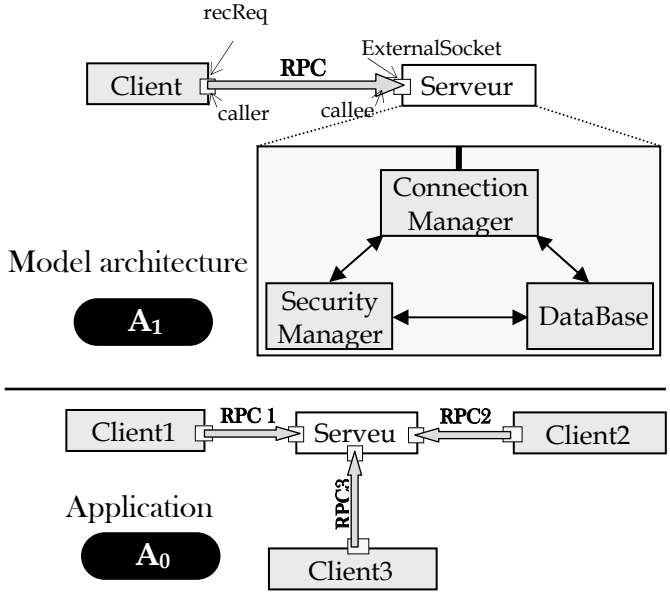


Figure 5.9 *Architecture/application du système Client-Serveur.*

5.3.2 La description du système Client-Serveur avec COSAPlugin

5.3.2.1 Exécution de la transformation COSA vers UML 2.0

Pour examiner la transformation COSA vers UML 2.0 sur le système Client-Serveur, nous avons représenté le PIM en COSA par un modèle de composants UML 2.0 et des contraintes OCL. Ensuite, le profil COSA est appliqué à travers une boîte de dialogue de sélection des profils, illustré dans la figure 5.10. Tous les stéréotypes sont disponibles, affectés, et contribués avec les tagged-values. Le modèle est évalué pour enlever toute violation des contraintes.

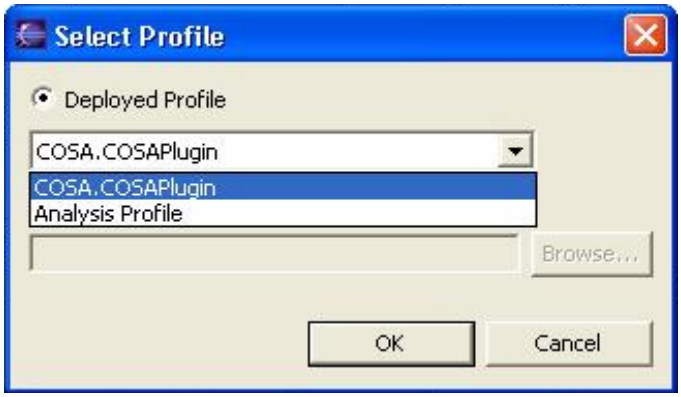


Figure 5.10 *Sélection du profil COSA pour le système Client-Serveur.*

5.3.2.2 Validation de modèle

Le modèle du système Client-Serveur, est validé avec les contraintes sémantiques définies par le profil, vu dansle chapitre précédent. Un ensemble d'instances (ex: arch-1) sont établies en instanciant les classes du modèle et sont évaluées. Le système final projeté est illustré dans la figure 5.11.

5.3.3 La description du système Client-Serveur avec COSABuilder

5.3.3.1 Modélisation des architectures

Pour illustrer comment les concepts et les relations de COSA peuvent être utilisés pour décrire un système, nous nous appuyons sur la figure 5.12 décrivant un système de Client-Serveur. Nous modélisons le système Client/serveur avec les outils de modélisation et d'instanciation COSA pour chaque type d'architecture (modèle et application).

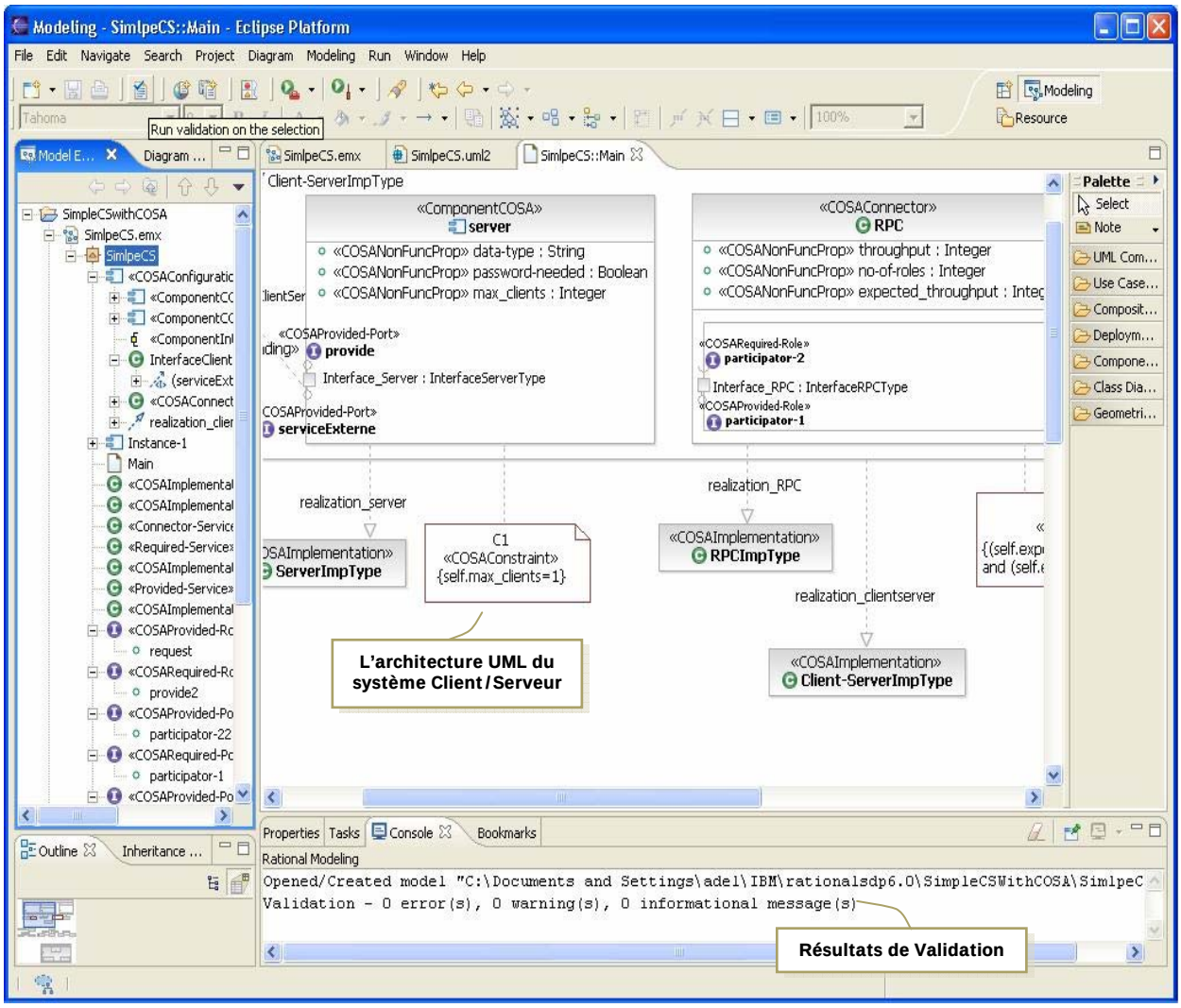


Figure 5.11 *Validation du système Client-Serveur en UML 2.0 avec RSM.*

De nouveaux types de composants et de connecteurs peuvent être facilement définis à partir des éléments existant sur le système client-serveur en utilisant le mécanisme d'héritage (c.-à-d Connecteur d'extension de COSABuilder). La figure 5.13 montre la spécialisation de connecteur RPC, selon deux rôles caller2 et caller3. Le connecteur RPC-1 peut être utilisé pour connecter trois clients avec le serveur via le rôle rcvReq.

5.3.3.2 Exécution de la transformation de l'architecture COSA vers eCore

La transformation COSA2eCore est appliquée au modèle COSA UML (client-server.uml) pour élaborer son correspondant modèle Ecore (client-server.ecore). La figure 5.14, illustre la transformation de COSA-UML vers eCore. Le modèle du système Client-Serveur validé avec les contraintes sémantiques définies par eCore.

Grâce à cette mise en projection de COSA en eCore, nous avons généré automatiquement le modèle de composant eCore de Client-Serveur et évalué de façon automatique toute la partie structurelle de l'application Client-serveur pour l'architecture COSA-eCore en respectant les contraintes sémantiques de cette architecture. Ce résultat est intéressant puisqu'il permet de maintenir un lien fort entre l'architecture abstraite COSA et le modèle d'architecture eCore. Grâce au profil UML 2.0 pour COSA, les cohérences structurelles se trouvent assurées. Le modèle d'architecture COSA est fortement lié au modèle d'architecture eCore qui représente mieux les concepts architecturaux de COSA. Les mécanismes d'extension eCore (concept EAnnotation) du profil COSA couvrent tous les besoins de l'architecte des logiciels.

5.3.3.3 Génération de l'éditeur de l'architecture client-serveur

La figure 5.15, présente un aperçu de l'éditeur graphique de l'architecture client-serveur selon les différentes vues: éditeur d'instance COSA avec sa palette, la palette et la vue Propriétés. Si nous regardons la "palette", nous allons découvrir que tous les éléments du modèle de l'architecture client-serveur, de la même façon que nous avions glisser et déposer l'élément approprié dans la zone qui contient le diagramme d'application client-serveur pour décrire une application complète. Dans la catégorie *ComponentInstance,* on trouve tous les types de composants de l'architecture client-serveur. Dans la catégorie *ConnectorInstance,* on trouve tous les types de connecteurs définis par l'utilisateur de l'architecture client-serveur (par exemple : connecteur RPC), dans la catégorie *ProvidedPortInstance* nous trouvons tous les ports requis de l'architecture client-serveur (exemple : port *externalSocket* du composant serveur), dans la catégorie *RequiredPortInstance*, on trouve tous les ports requis de l'architecture client-serveur (exemple : port recReq du composant client) et il en est de même pour les rôles.

5.3.3.4 Expérimentation des architectures de l'application Client-Serveur

Plusieurs instances d'architecture ont été expérimentées. Une première architecture testée est constituée d'un seul client, un seul serveur et un seul connecteur RPC. Une seconde architecture a trois clients, trois connecteurs RPC et un seul serveur. Chaque connecteur est utilisé pour connecter un client au serveur (voir figure 5.15).

5.3.3.5 Analyse des résultats

Tous comme pour UML 2.0, grâce à la mise en application de l'approche profil UML et MDA, nous avons généré automatiquement le modèle eCore du système client-serveur et son éditeur graphique. Plusieurs instances d'architectures des applications de type client-serveur, ont été expérimentées. Nous constatons que tous les concepts de l'architecture logicielle COSA se retrouvent spécifiés dans la transformation COSA-eCore. Les concepts qui ne sont pas définis explicitement dans eCore (connecteur, rôle, glu, configuration, etc.), sont également spécifiés. Cette transformation permet de conserver les relations sémantiques entre un composant et son type, entre un connecteur et son type et entre une configuration et son type à l'aide des annotations eCore. Soulignons que le résultat final obtenu est fonction de la qualité du langage de transformation des modèles (ATL). Nous nous sommes restreints aux parties structurelles des architectures ; il reste encore à améliorer les modèles eCore pour qu'ils supportent les parties dynamiques.

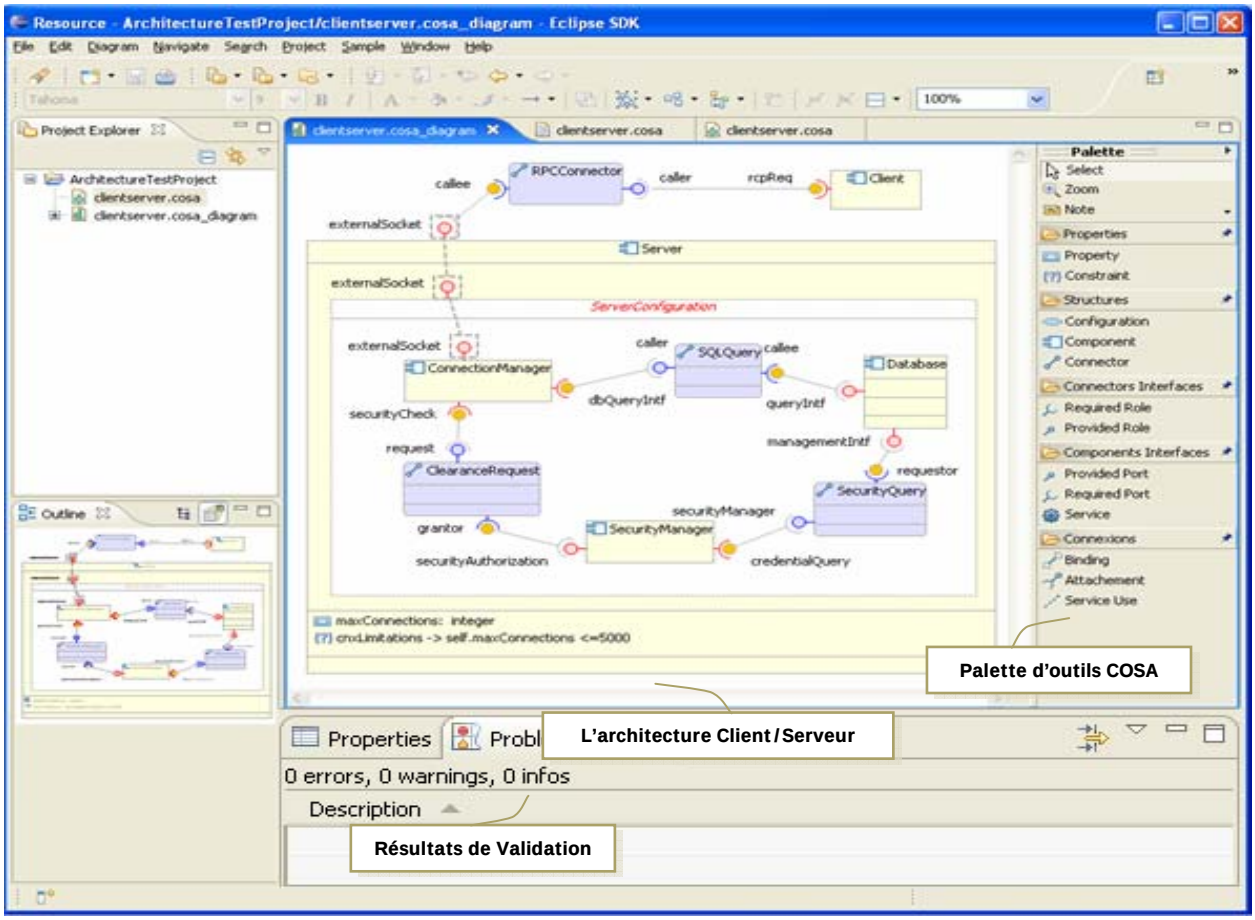


Figure 5.12 *Architecture Client/serveur en utilisant COSABuilder.*

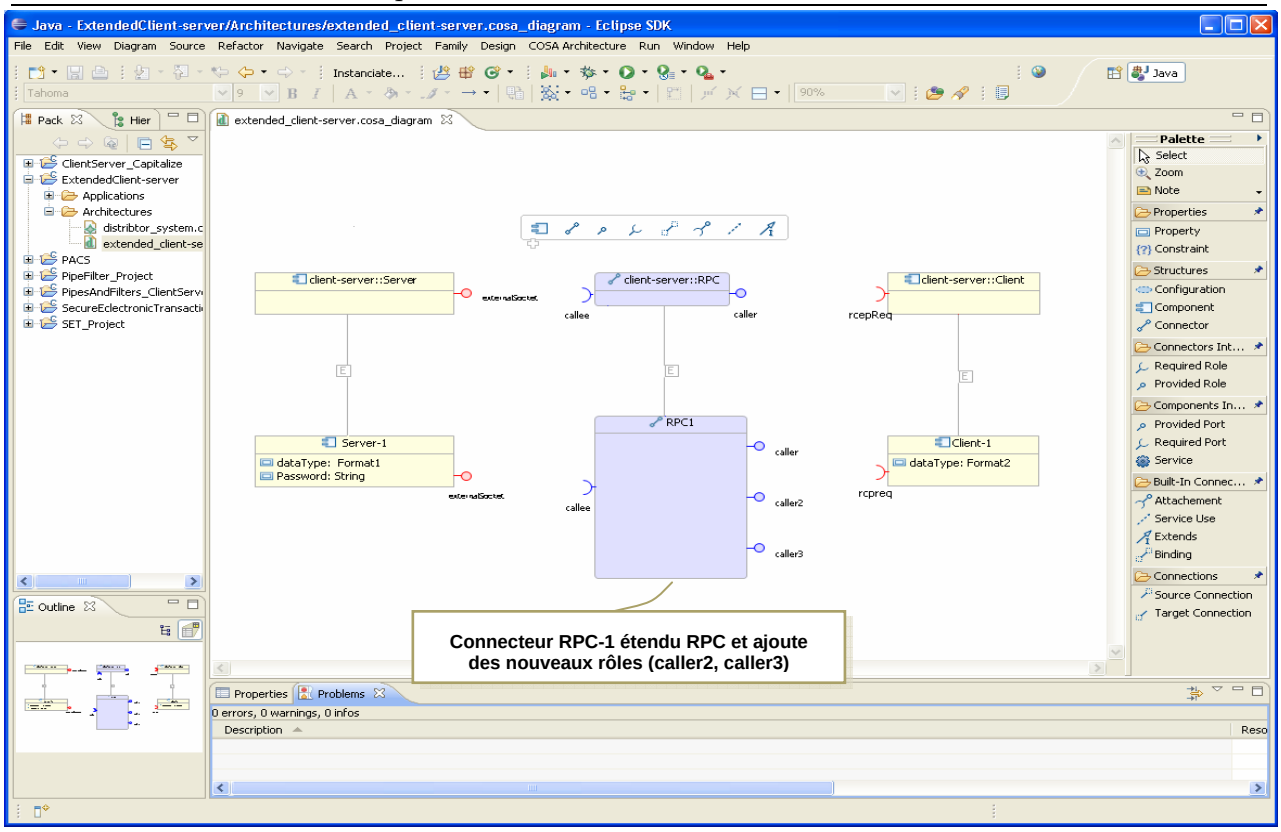


Figure 5.13. *Spécialisation d'un connecteur de l'architecture client / serveur.*

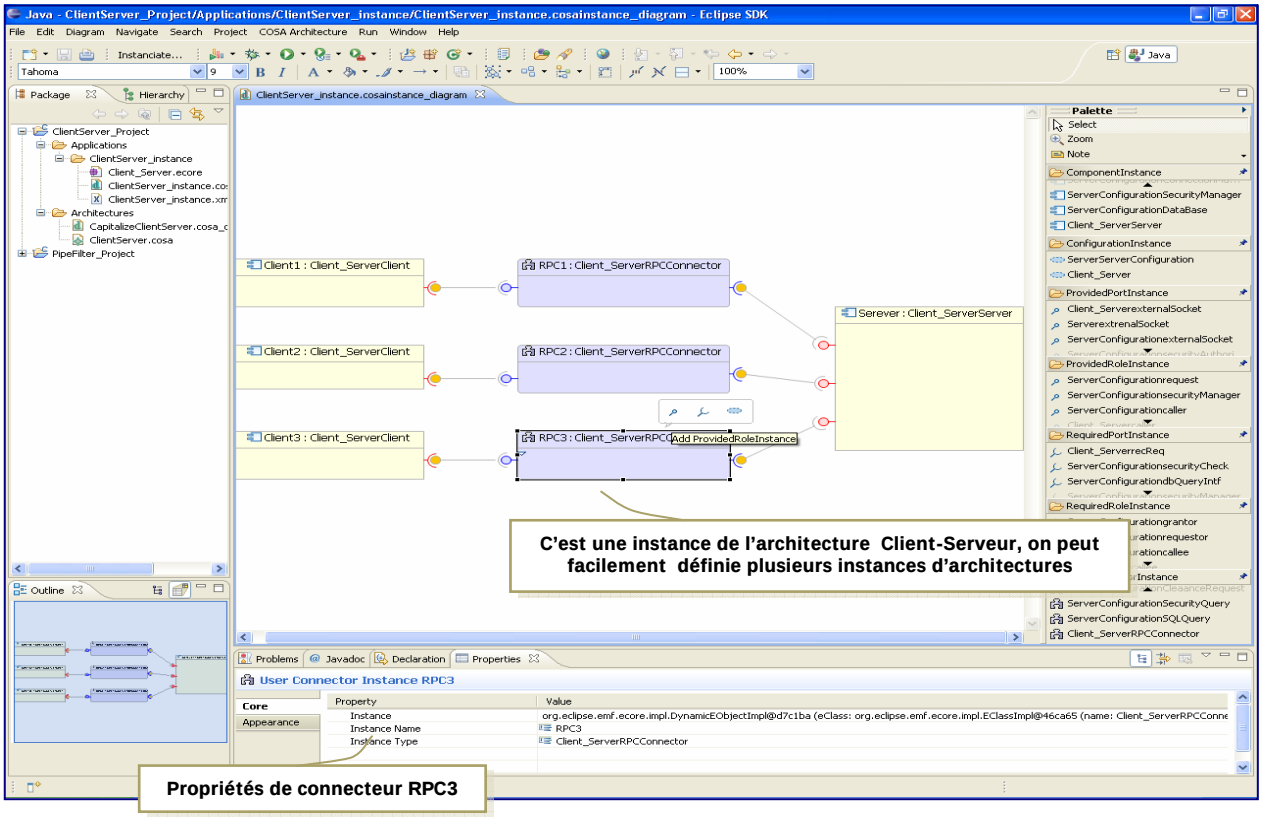


Figure 5.15. *Application Client-Serveur en utilisant le générateur COSAInstantiator.*

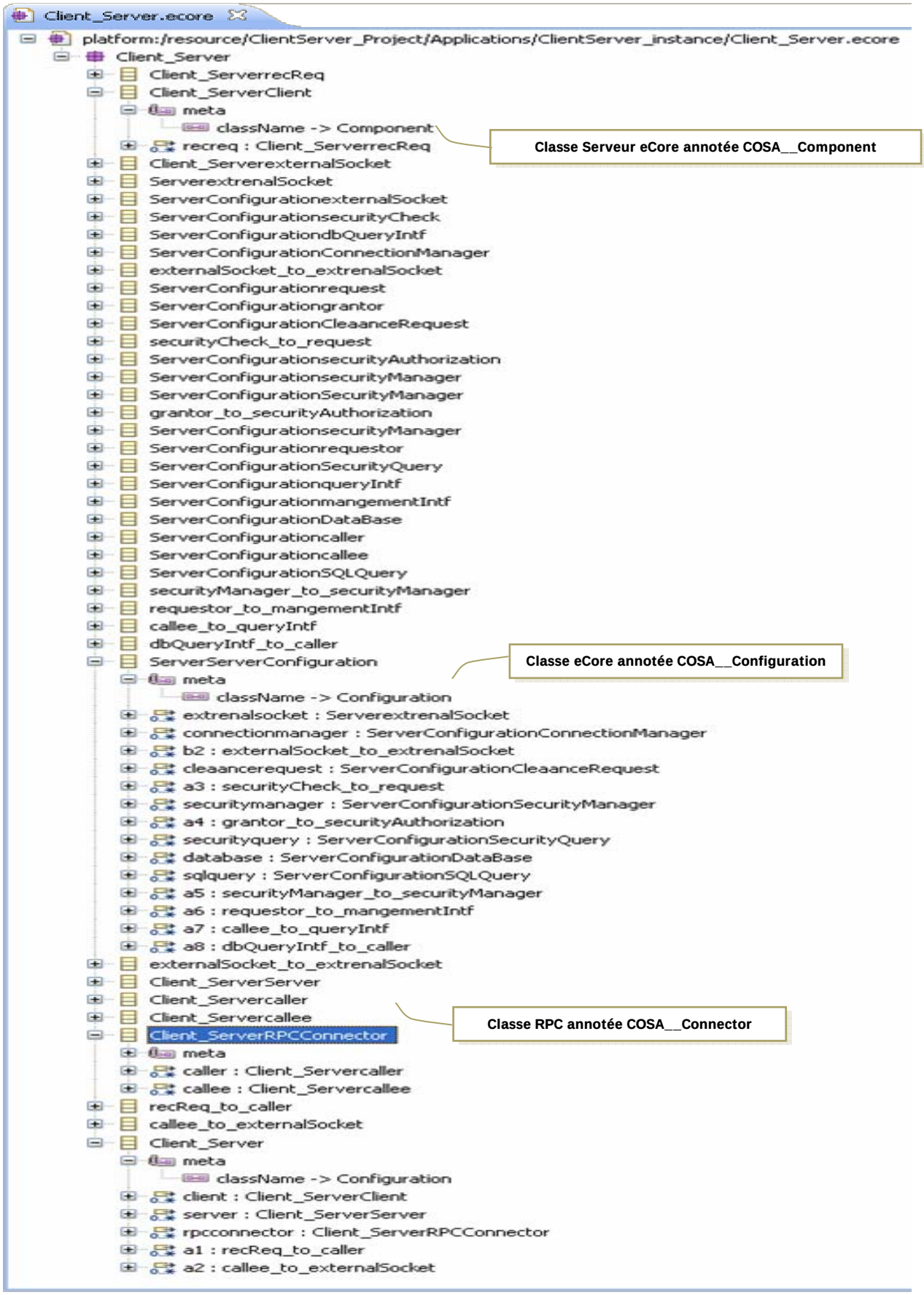


Figure 5.14. *Modèle eCore de l'architecture Client/Serveur.*

5.3.4 Réalisation du système Client-Serveur avec COSA2CORBA Plugin

5.3.4.1 Exécution de la transformation COSA vers UML 2.0

Afin d'examiner cette transformation, nous avons construit un modèle de composants UML 2.0 du système client-serveur, auquel nous avons ajouté tous les stéréotypes et on a défini ses valeurs marquées par l'application de profil UML 2.0 pour COSA. Le modèle est validé par le profil UML 2.0 pour COSA. La configuration COSA vers CORBA permet de spécifier le modèle source et le modèle cible, (exemple : CSSystemwithCOSA.uml2 et CSSsytemwithCORBA.ecore) et les métamodèles (Profils UML COSA et CORBA). La figure 5.16, illustre le résultat de la transformation COSA vers CORBA pour le système client-serveur. Nous constatons que le modèle CORBA contient tous les concepts liés aux connecteurs COSA et à ses configurations. Pour finir, nous avons validé automatiquement ses sémantiques structurelles selon le profil CORBA.

5.3.4.2 Génération de code

À partir de ce modèle, nous avons appliqué la génération de code des interfaces IDL par l'application de la transformation de génération de texte CORBA-IDL. L'annexe B.2 présente un résumé de la transformation CORBA-IDL, selon la description CORBA. Nous avons généré les squelettes de code de l'application Client-Serveur. Ces squelettes sont complétés par les codes dynamiques dans le but de terminer la réalisation de l'application Client-Serveur dans IDL.

5.3.4.3 Analyse des résultats

Grâce à la mise en œuvre de COSA-UML en CORBA, nous avons généré automatiquement les squelettes des interfaces IDL, correspondants aux composants COSA-UML de Client-Serveur. Ce résultat montre que les principes de MDA, sont valables également sur les langages de description d'architecture. Les concepts architecturaux COSA sont fortement liés au profil UML (PIM), et les concepts architecturaux se retrouvent dans la transformation COSA vers CORBA (PIM vers PSM). Nous avons élaboré nous-même, le profil UML 1.4 pour CORBA et le profil UML

2.0 pour COSA ainsi que la transformation COSA-UML vers CORBA. L'approche MDA nous a permis de générer une grande partie du code IDL (Voir Annexe B.3).

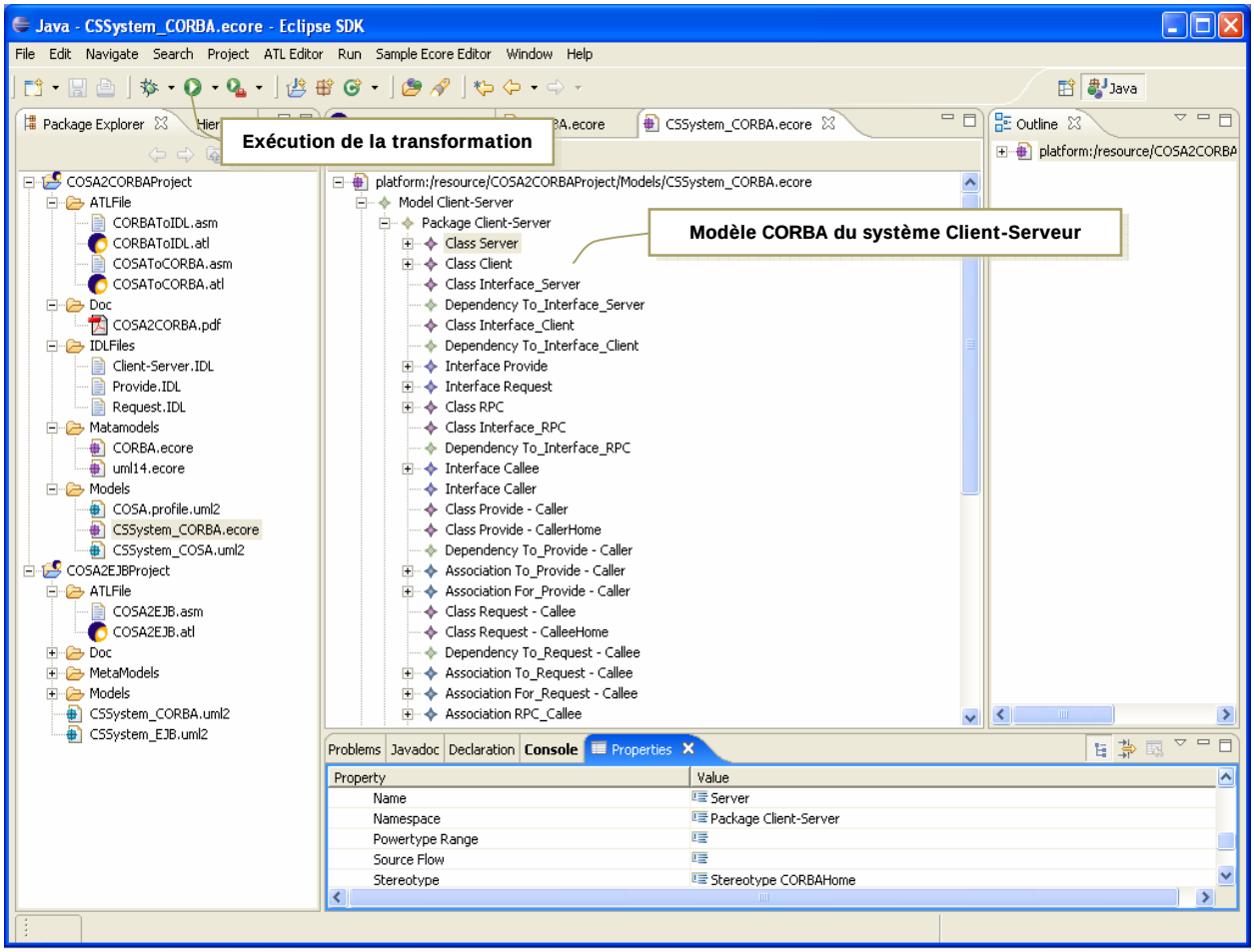


Figure 5.16. Modèle CORBA du système Client-Serveur.

5.3.5 COSA-UML vs. COSA - CORBA

Les transformations COSA-UML, COSA–eCore et COSA-CORBA, montrent qu'il est possible d'automatiser l'intégration de l'architecture logicielle COSA au sein de MDA pour ses parties structurelles. Il reste à noter les caractéristiques de ces transformations :

1. La transformation COSA vers UML est indépendante du langage de programmation, par contre COSA vers CORBA est fortement liée aux implémentations CORBA. Cependant, puisque la transformation COSA vers UML projette une architecture abstraite COSA sur une autre architecture concrète UML, l'architecture, peut être raffinée dans n'importe quel langage d'implémentation. Ce n'est pas le cas de la transformation COSA vers CORBA, qui est une transformation de modèle d'architecture vers un autre modèle d'implémentation qui dépend de CORBA.

2. La transformation COSA vers UML, intéresse plusieurs intervenants (architectes, programmeurs, etc.), par contre COSA vers CORBA n'intéresse que les développeurs CORBA. Pour les différents types d'intervenants (managers, programmeurs, clients, etc.), il est préférable d'avoir un ADL qui n'est associé à aucun langage d'implémentation. Par exemple, les managers et les clients préfèrent les modèles au code concret.

3. COSA vers UML est une transformation de représentation de l'architecture abstraite COSA vers l'architecture concrète UML, par contre COSA vers CORBA est une transformation de génération de code de l'architecture vers l'implémentation. Ceci s'explique d'une part, par le fait que la transformation COSA vers UML est une transformation de représentation et de modélisation, d'autre part, par contre la transformation COSA vers CORBA est une génération de code.

4. Les connecteurs et les composants dans la transformation COSA vers UML sont lisiblement, très séparés par contre dans la transformation COSA vers CORBA, ils sont mélangés. Cependant, on a profité d'UML et de ses capacités d'expressivité des stéréotypes pour représenter les composants COSA et les connecteurs COSA par des classes UML, mais ils restent distincts avec le concept des stéréotypes profil UML. Ce n'est pas le cas, de la transformation COSA vers CORBA, qui utilise le même concept de composants CORBA pour la représentation des composants et des connecteurs COSA.

La table 5.5, résume cette comparaison, suivant les critères de niveau de description, de séparation des aspects, de raffinement, de degré de lisibilité et du type de transformation.

Critères	COSA vers UML	COSA vers CORBA
Niveau de description	Haut niveau	Bas niveau
Séparation des aspects	Aucune séparation	Architecture et implémentation
Raffinement	Indépendant des L.P	CORBA
Lisibilité	Très lisible	Moins lisible
Transformation	Représentation	Génération de code

Table 5.5. *COSA-UML vs. COSA-CORBA.*

5.4 Etude comparative et Synthèse

On compare l'outil COSABuilder développé dans notre travail, avec deux outils utilisés au sein des écoles américaine et européenne, qui sont ACMEStudio (ACMEStudio, 2004) et Fractal ADL (cf. le site de Fractal ADL [Fractal 2004]).

ACMEStudio est bien connu dans la communauté académique et industrielle. COSABuilder et ACMEStudio sont deux outils de modélisation graphiques des architectures qui se basent sur les mêmes concepts d'architectures. A la différence d'ACME, COSA définit explicitement les configurations comme une entité de première classe. Fractal est un outil de composants développé par France Télécom R&D et l'INRIA. L'outil Fractal ADL, offre des moyens de développer et de gérer des systèmes complexes comme les systèmes distribués. Les outils de COSA et Fractal sont deux outils graphiques de description architecturale qui ne se basent pas sur les mêmes concepts d'architectures.

5.4.1 Comparaison avec ACMEStudio

Nous avons demandé à dix développeurs d'utiliser ACMEStudio et à dix autres développeurs, d'utiliser COSABuilder. Chaque groupe a été supervisé par un architecte. A la fin des expériences, soit après environ un mois, un questionnaire a été soumis à tous les développeurs de chaque groupe. L'annexe C.1, résume les réponses du questionnaire. Pour savoir si la différence est significative entre les deux outils, nous avons utilisé des échantillons t-test (t architecte-test).

Après avoir utilisé le logiciel R (R-software, 2008) qui est un logiciel de développement de calcul de statistiques et graphiques, nous avons obtenu les résultats suivants avec 95% de similitude et avec un niveau significatif ($\alpha = 0,05$);

N	20
Moyenne de 1er groupe	7.75
Moyenne de 2ème groupe	9,25
Tscore	-3.048
Degré de liberté	19
P-value	0.005

Table 5.6 *Résultats de comparaison entre COSABuilder et ACMEStudio.*

N: Taille de l'échantillon

Moyenne1 : La moyenne de 1er groupe (ACMEStudio).

Moyenne 2 : La moyenne 2ème groupe (COSABuilder).

Tscore : Paramétre de corrélation entre COSABuilder et ACMEStudio.

P-Value: La significativité de la différence entre COSABuilder et ACMEStudio.

A partir de la table 5.6, $t_{0.975} = \pm 1{,}093$, alors $t_{score} \prec t_{0.975}$ $(-3{,}084 \prec -1{,}093)$, cette différence est significative.

Afin d'évaluer l'efficacité de notre outil, le questionnaire a été conçu pour avoir une réponse à une série de cinq questions principales:

- Est-ce que COSABuilder / COSAInstantiator sont faciles à utiliser?
- Est-ce que COSABuilder/COSAInstantiator contribuent à acquérir des compétences professionnelles ?
- Y a t-il des problèmes face à l'outil COSABuilder ?
- Est-ce que COSABuilder/COSAInstantiator supporte la réutilisabilité/l'usabilité?
- Est-ce qu'on peut utiliser l'outil dans des projets du monde réel?

➢ **Est-ce que COSABuilder est facile à utiliser?**

Le but de cette partie est d'étudier les perceptions des architectes concernant la facilité d'utilisation de notre outil par rapport à ACMEStudio. Cette partie est couverte dans le questionnaire, par les questions suivantes:

Q1 : Est-ce que l'outil à une bonne interface graphique ?

Q5 : Est-ce que l'outil facilite la description des architectures logicielles complexes ?

Q6 : Est-ce que l'outil dispose d'une bonne interopérabilité ?

Q8: Est-ce qu'il est facile de vérifier le modèle d'architecture à tout moment ?

L'annexe C.1 présente les statistiques des architectes pour ces quatre questions. La plupart des architectes (environ 90%) ont préféré COSABuilder/COSAInstantiator et l'ont qualifié aussi facile à utiliser, comparativement à 60% des architectes de l'outil ACMEStudio. Concernant la nature des projets traités, 80% des architectes de l'outil COSABuilder estiment que cet outil a aidé à faire face à des architectures complexes, respectivement à 70% pour ACMEStudio. En outre, tous les architectes ont apprécié l'outil COSA et qualifié aussi à une bonne interopérabilité, respectivement à 60% pour

ACMEStudio. Tous les architectes de l'outil de COSA et ACMEStudio considérer ces outils comme faciles à utiliser.

Le résultat de cette partie montre que 90% des architectes préfèrent COSABuilder/COSAInstantiator qualifiée de facile à utiliser. En comparaison avec ACMEStudio, COSABuilder a une meilleure interface graphique et offre une bonne interopérabilité. L'évaluation a indiqué que les outils COSA sont beaucoup plus efficaces en termes de coût et de difficulté à construire des modèles d'architectures.

- **Est-ce que COSABuilder contribué à acquérir des compétences professionnelles?**

Le but de cette partie est d'évaluer le gain de compétences des architectes à la suite de l'utilisation de cet outil. Cette partie est couverte dans le questionnaire par les questions suivantes:

Q2: Est-ce que l'outil est utile pour acquérir des compétences professionnelles ?

Q3: Est-ce que l'outil est utile d'apprendre des choses ?

Q12: Est-ce que tous les concepts d'architecture sont disponibles dans l'outil ?

Q15: Est-ce que l'outil m'a donné une chance de faire quelque chose d'utile pour ma communauté ?

Q16: Est-ce que l'outil m'a aidé à définir des contraintes sur les éléments d'architecture ?

L'annexe C.1 présente les perceptions des architectes pour ces cinq questions. Les résultats montrent que tous les architectes ont acquis un nombre de compétences professionnelles à la suite de l'utilisation de l'outil COSABuilder et ACMEStudio. En outre, 90% des architectes de l'outil COSABuilder ont acquis des connaissances nouvelles (comme la déférence entre un connecteur défini par l'utilisateur et un connecteur de construction), comparativement à 70% des architectes de l'ACMEStudio. Seulement 10% des architectes qui utilisent COSA ne sont pas d'accord par rapport à 30% qui utilisent ACMEStudio. La plupart des architectes, 90%, montre que la majorité des concepts de modélisation sont disponibles dans COSABuilder comparativement à 80% pour ACMEStudio. Tous les architectes ont montré que les deux outils permettent de vérifier leurs modèles de définir des contraintes de leurs éléments d'architecture et ont donné la définition des contraintes pour les éléments de l'architecture.

Les résultats de cette partie, montrent que les utilisateurs ont acquis des connaissances nouvelles ; la plupart des concepts de modélisation sont disponibles dans COSABuilder comme par exemple les connecteurs de construction, l'attachement, l'Extend, le Binding et l'Use, les connecteurs définis par l'utilisateur et les structures comme des configurations de composants et de connecteurs).

- **Ya-t-il des problèmes, face aux outils de COSA?**

Le but de cette partie est de déterminer les principaux problèmes que les architectes ont pu rencontrés lors de l'utilisation des outils COSA. Cette partie est couverte dans le questionnaire par les questions suivantes:

Q7: La période de temps allouée ne suffit pas pour compléter la conception?

Q11: Est-ce qu'on peut appliquer une méthode de vérification formelle lors de la conception ?

Q14: Est-ce que l'outil m'a donné la chance de travailler sur un projet réel ?

Q17: Avez-vous d'autres recommandations qui pourraient être utiles pour améliorer l'outil ?

L'annexe C.1, présente les perceptions des architectes pour ces cinq questions. Nous pouvons conclure que le problème est la période de temps allouée aux projets. 30% des architectes qui ont utilisé l'outil COSA, estiment que la période n'était pas suffisante, alors que 10% des architectes pensaient que le temps peut être l'un des problèmes, mais sans en être certains, et 60% des architectes ont estimé que le temps alloué est insuffisant pour terminer un projet.

Un autre problème, est la capacité à créer et gérer des architectures professionnelles. 60% des architectes pensent qu'ils peuvent utiliser l'outil COSA sur des projets réels par rapport à 50% accordée aux ACMEStudio. 30% des architectes de l'outil COSA n'étaient pas sûrs. Les outils COSABuilder et ACMEStudio, peuvent facilement appliquer un processus de vérification formelle, lors de la conception (tous les architectes étaient convaincus).

Les résultats de cette partie soulèvent un autre challenge aux concepteurs de génie logiciel : maîtrise des concepts architecturaux et techniques de vérification formelle dans le COSABuilder/COSAInstantiator. ACMEStudio, manque de documentation

réelle par rapport aux outils de COSA. Toutefois, les outils COSA, manquent de génération automatique de parties comportementales de l'implémentation du système.

- **La réutilisation/ l'usabilité de l'outil COSABuilder ?**

Le but de cette partie est de déterminer si l'outil COSABuilder prend en compte la réutilisabilité, la facilité d'utilisation. Cette partie, est couverte dans le questionnaire mis en place:

Q4: l'outil prend-il de la réutilisation ?

Q9: l'outil est 'il facile à utiliser ?

Q13: le modèle d'architecture, peut-il être instancié à tout moment ?

L'annexe C.1 présente les perceptions des architectes pour ces trois questions. Cette partie montre les opinions des architectes sur la facilité d'utilisation et sur la réutilisation de l'outil COSABuilder par rapport à ACMEStudio. La plupart des architectes de l'outil COSA, soit environ 90%, montrent que l'outil ayant une réutilisation forte est bien COSABuilder qui intègre les mécanismes d'instanciation, d'héritage et de composition, comparativement aux 70% des architectes d'ACMEStudio. Pour l'instanciation automatique, tous les architectes qui utilisent l'outil COSABuilder estiment que, les architectures sont instanciables à tout moment, comparativement aux 50% qui utilisent l'outil ACMEStudio. 80% des utilisateurs soulignent la facilité d'utilisation de l'outil COSA par rapport à 50% des utilisateurs ACMEStudio. 40% des architectes de cet outil n'ont pas d'avis arrêté.

Les résultats de cette partie montrent que le degré de réutilisation de l'outil COSABuilder/COSAInstantiator est fort par rapport à l'outil ACMEStudio. Pour décrire différentes architectures, nous pouvons instancier le modèle d'architecture à tout moment.

- **Pouvez-vous utiliser l'outil dans des projets du monde réel?**

Il s'agit de déterminer si l'outil COSABuilder permet de décrire des projets du monde réel. Les réponses aux questions suivantes sont les clés de cette problématique:

Q14: Est-ce que l'outil m'a donné une chance de faire quelque chose d'utile pour ma communauté ?

Q15: Est-ce que l'outil m'a donné la chance de travailler sur un projet réel ?

Pour les services rendus par l'outil COSABuilder dans la communauté, 50% des architectes estiment que l'outil COSABuilder leur donnent une chance d'appliquer leurs connaissances pour servir leur communauté, alors que 40%, sont portés sur ACMEStudio. 40% des architectes, pensent que l'outil de COSA peut leur donner une telle chance, sans en être sûrs. On relève que seuls 10% des architectes ne sont pas d'accord avec les services qu'offrent COSABuilder.

Pour la nature du projet que les architectes ont développé, 60% d'entre eux estiment que l'outil COSABuilder, leur fourni une chance de travailler sur des projets réels. Alors que 50% des architectes pensent plutôt à ACMEStudio. 30% des architectes, ne sont pas sûrs de l'apport de l'outil COSABuilder.

Ces résultats montrent bien que l'outil COSABuilder, apporte une aide aux architectes à réduire les espaces entre l'industrie et la communauté académique dans le domaine de l'architecture logicielle. COSABuilder fournit un moyen facile pour décrire des architectures logicielles complexes avec un éditeur visuel, d'utilisation aisée à la création des diagrammes. Cet outil (COSABuilder) a convaincu les architectes pour son application dans les projets du monde réel, et dans le futur.

Ainsi, on peut conclure que COSABuilder/COSAInstantiator est privilégié : choix fondé par sa facilité d'utilisation, son efficacité et la satisfaction des préférences, qu'il offre. À titre d'observation générale sur les résultats des groupes expérimentaux, on peut relève que:

– 90% des utilisateurs, ont une préférence à COSABuilder/COSAInstantiator : utilisation facile.

– Acquisition de nouvelles connaissances : exemple des connecteurs des utilisateurs et de construction.

– Réutilisation du même modèle d'architecture plusieurs fois avec les mécanismes opérationnels de COSA: héritage, composition, tracibilité, raffinement et instanciation.

– Instanciation des modèles d'architectures à tout instant.

– COSABuilder standard des concepts de modélisation des architectures logicielles : Connecteurs de Construction (Attachement, Extend, Use et Binding), Connecteurs définis par l'utilisateur, Configuration des composants et des connecteurs.

- Déploiement automatique d'une architecture donnée de plusieurs manières.

En résumé, dans la description d'un domaine générique, il est préférable d'utiliser les outils de COSA. Par contre, pour des domaines spécifiques, il est préférable d'utiliser l'outil ACMEStudio. La table 5.7 met en relief, cette comparaison, qui est effectuée selon la disponibilité des concepts architecturaux, la réutilisation, la séparation des aspects, l'instanciation et le raffinement.

Critères	COSABuilder/COSAInstantiator	ACMEStudio
Concepts architecturaux	Standard des ADLs	(-) Configuration, Extend, Use
Réutilisation	Fort	Moyen
Séparation des aspects	Architecture et déploiement sont explicitement séparés.	Architecture et déploiement sont mélangés.
Instanciation	Composants, Connecteurs, Configurations.	Composants, Connecteurs
Raffinement	Langage de transformation ATL	Patrons de transformation

Table 5.7. *COSABuilder/COSAInstantiator vs. ACMEStudio*

5.4.2 Comparaison avec FractalGUI

Notre choix s'est porté sur FractalGUI, système Client-Serveur, implémenté selon l'outil Fractal ADL, décrit dans Fractal ADL [FractalGUI 2005], pour qui nous proposons une étude comparative avec l'outil développé, à savoir COSABuilder. Tout d'abord, pour une comparaison efficace, notons quelques caractéristiques de FractalGUI :

– L'outil FractalGUI, est simple de développement par l'assemblage des composants. Ce qui fait, il offre une bonne séparation entre les aspects architecture et déploiement, dans la description de leurs systèmes. L'assemblage des composants est définis via les fichiers XML, et donc, il est difficile de garantir la réutilisation des composants. En utilisant les outils COSABuilder/COSAInstantiator, on peut déployer automatiquement une architecture donnée de plusieurs manières, sans réécrire le programme de configuration/déploiement. Ils fournissent une bonne séparation de l'application de l'architecture qui rend ces préoccupations réutilisables.

– Les outils des ADLs (Fractal et COSA) fournissent des outils graphiques qui peuvent être utilisés pour visualiser, définir ou modifier graphiquement et interactivement des architectures. A la différence de Fractal, COSABuilder, réunifie les concepts communément admis par la majorité des langages de description

d'architectures logicielles : Connecteurs de construction (*Attachement*, *Extend*, *Use* et *Binding)*, Connecteurs définis par les utilisateurs, Configurations des composants et des connecteurs.

– Le concept de configuration n'existe pas dans l'outil FractalGUI, puisque FractalGUI décrit les architectures comme des composants composites. Dans COSABuilder, la définition des configurations comme des classes instanciables permet la construction de différentes architectures du même système.

– L'outil FractalGUI se limite à une définition implicite des connecteurs en ne considérant que les composants et leurs structures ; ce qui rend l'architecture du système pas très visible. Les interactions entre composants n'ont pas suscité beaucoup d'intention. En effet, dans Fractal les connecteurs sont définit implicitement et leur sémantique enfouie au sein des composants.

– Les outils des ADLs (COSA et Fractal) favorisent les mécanismes d'héritage et de composition. A la différence de COSA, seulement les composants peuvent héritier. L'outil FractalGUI fournit des avantages de réutilisation et tend à offrir une composition hiérarchique.

– Les outils des ADLs Fractal et COSA ne proposent pas explicitement, une manière de définir des styles architecturaux. Récemment, Alti & al. (Alti, Boukerram, Derdour, Roose, 2010), ont proposé une extension du méta-modèle COSA avec la norme ISO-9126 (ISO-IEC, 2001) avec des attributs de qualité du système et des styles architecturaux afin de supporter l'évaluation métaheuristique d'une architecture à base des styles architecturaux.

En résumé, nous nous sommes intéressés à la description des systèmes logiciels par leur modèle, plutôt que d'aller directement vers l'implémentation, avec l'utilisation des outils de COSA, que ceux de FractalGUI.

La table 5.8, récapitule cette comparaison, selon les critères de degré de lisibilité, de séparation des aspects, de séparation des concepts, de traçabilité et d'instanciation.

Critères	COSABuilder/COSAInstantiator	FractalGUI
Lisibilité	Très lisible	Moins lisible
Séparation des concepts	Configurations	-
	Composants, connecteurs sont explicitement séparés.	Composants et connecteurs sont mélangés.
Séparation des aspects	Architecture et déploiement sont explicitement séparés.	Architecture et déploiement sont mélangés.
Raffinement	Projection des modèles	Java
Instanciation	Composants, Connecteurs, Configurations.	Composants.
Génération de code	Outil indépendant du langage d'implémentation	Code Fractal 2.0

Table 5.8. *COSABuilder/COSAInstantiator vs. FractalGUI.*

5.4.3 Synthèse

Dans la majorité des outils (ArchJava, ArchStudio, AcmeStudio, FractalGUI, etc.), l'accent est mis sur la validation de l'architecture de façon correcte. Ce point, concerne la validation de l'ADL COSA sur des cas réels. L'outil COSABuilder vise à permettre aux architectes de décrire graphiquement leurs architectures et de valider automatiquement leurs sémantiques selon l'approche COSA. L'aspect clé, de l'outil COSAInstantiator est la possibilité de générer automatiquement un éditeur graphique pour un modèle d'architecture COSA et de créer des architectures d'applications correctes.

A la différence des autres outils tels que ArchStduio, AcmeStudio, FractalGUI, etc. COSABuilder/COSAInstantiator, définit explicitement les configurations comme une entité de première classe. COSABuilder / COSAInstantiator offre d'autres avantages par rapport à ces outils:

– un environnement de modélisation visuel permettant de modéliser facilement des architectures logicielles complexes,

– un package complet des éléments architecturaux : connecteurs de construction tels que *Attachement*, *Extends*, *Binding* et *Utilisation*, connecteurs définis par l'utilisateur, structures telles que la configuration des composants et des connecteurs,

– une boîte à outils et un modèle standard COSA-eCore permettant de réduire le coût et la difficulté de construire des modèles d'architecture,

– un support formel utilisant OCL, pour non seulement de validation d'un modèle, mais aussi il permet de trouver et de localiser tous les éléments COSA,

– un processus d'instanciation automatique permettant de définir rapidement des applications logicielles,

– une séparation de l'application de l'architecture qui rend ces préoccupations réutilisables,

– une préservation de la traçabilité sémantique de l'architecture COSA dans l'espace de modélisation eCore.

– une description plus adéquate des propriétés définies au niveau méta (concept EClass en eCore) plutôt que d'utiliser un simple attribut à cet effet.

Les architectes qui utilisent COSABuilder peuvent faire face à un certain nombre de problèmes :

– Il n'y a pas de possibilité d'enregistrer, des parties d'une architecture ou des contraintes des modèles.

– Certains modèles du système doivent être clarifiés par l'utilisation des notes. On remarque l'absence de cette information sur le modèle.

5.4.4 Améliorations futures

Le développement des outils COSABuilder/COSAInstantiator, valide le travail réalisé par Khammaci et al. Sur l'ADL COSA (Khammaci, Smeda, Oussalah, 2005). Les outils valident également une partie des idées proposées dans cette thèse autour des composants architecturaux (composants, connecteurs et configurations). Le problème est que le métamodèle eCore, facilite la manipulation d'entités de type produit, mais beaucoup moins celle d'entités de type architectures comme c'est le cas des configurations et de connecteurs, donc, les concepts de base d'architectures logicielles. Les outils COSABuilder/COSAInstantiator ont été testés sur des systèmes réels « Secure Electronic Transaction System » et « Parking Access Control System » (voir Annexe C.2). Cela dit, il reste toujours à valider COSABuilder/COSAInstantiator sur des systèmes complexes et larges avec des partenaires industriels.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'intégration de l'architecture logicielle COSA au sein de MDA. Cette dernière passe par l'utilisation du profil UML. En plus, nous avons présenté trois transformations :

- COSA (PIM) vers UML 2.0 (PSM),
- la transformation COSA profil UML (PSM) vers le métamodèle eCore (PSM) permet une projection de l'architecture abstraite COSA vers une architecture concrète eCore
- la transformation COSA profil UML (PSM) vers CORBA profil UML (PSM). Elle permet une projection de l'architecture COSA modélisée avec UML vers la plate-forme CORBA.

Pour chaque transformation nous avons réalisé un Plug-In, implémenté dans la plateforme Eclipse 3.1. Ensuite, nous avons appliqué, ces transformations au système Client-Serveur. Les résultats obtenus montrent que les concepts COSA sont fortement liés au profil UML (PSM) et se retrouvent spécifiés dans les transformations PSM vers PSM (COSA-eCore) et (COSA-CORBA). A l'heure actuelle, notre plug-in permet à un architecte d'applications, de modéliser correctement des architectures logicielles et de maintenir depuis une architecture, une instanciation correcte des applications. Néanmoins il reste toujours à valider ces outils sur des systèmes industriels complexes et larges.

De cette étude comparative, de outils de COSA avec ceux des ADLs ACME et Fractal, on conclut que l'architecture abstraite COSA est entrée dans une phase de concrétisation et d'implémentation au sein de MDA car les concepts architecturaux COSA se retrouvent au niveau des modèles spécifiques d'eCore et de CORBA. Les outils de modélisation et d'instanciation COSA permettent de fournir la majorité des concepts d'ADLs, basés sur des implémentations de normes, UML, eCore/MOF et de CORBA et de conserver ainsi, la traçabilité sémantique entre l'architecture et l'implémentation. Ces outils sont ouverts et extensibles, a permis d'intégrer au plutôt

les avancées des évolutions technologiques du monde académique et de la recherche, tout en les appliquant à des solutions industrielles. Grâce à l'implication des partenaires industriels, ces outils peuvent être pérennisés dans une nouvelle plateforme de plus grande ampleur.

Conclusion générale et perspectives

Bilan des travaux et apports de la thèse

Les travaux décrits dans cette thèse portent sur la définition d'un profil UML pour l'architecture logicielle et ensuite l'intégration de ce profil dans la démarche MDA. Analysons ce qui a été fait au niveau de ces travaux.

Nous avons exposé les approches de modélisation des architectures logicielles à travers des travaux les plus significatifs dans le domaine. Les modélisations par objets, dite architectures logicielles à base d'objets, et la modélisation par composants, dénommée architecture logicielle à base de composants sont les deux approches qui ont émergé pour décrire les architectures logicielles. Ces deux approches sont très semblables. En fait les deux sont basées sur les mêmes concepts, qui sont l'abstraction et les interactions entre les entités. Cependant, chacune d'elles a ses avantages et également ses défauts. Ainsi, nous pensons qu'une architecture logicielle basée sur les deux approches est très prometteuse pour le futur de l'architecture logicielle et participera à la réduction de la distance sémantique entre la conception et l'implémentation. Les objectifs de cette approche hybride consistent à réduire les coûts de développement, à améliorer la réutilisation des modèles, à faire partager des concepts communs aux utilisateurs de systèmes et enfin à construire des systèmes hétérogènes à base de composants réutilisables sur étagères (Component-Off-The-Shelf). A travers cette étude, nous avons mis en relief, les principaux objectifs de la modélisation des architectures logicielles.

Par ailleurs, les langages de modélisation des architectures logicielles sont relatifs aux approches de modélisation par objets et par composants. Dans ce contexte, nous avons décrit certains langages tout en explicitant les principaux concepts et leurs caractéristiques. Les principaux concepts que doit supporter un langage de description

d'architectures logicielles sont soulignés. Tout en montrant, l'importance de ce métamodèle dans l'approche MDA, nous avons présenté la notion de profil qui donne sa dimension de flexibilité à UML 2.0. Grâce aux profils, il est possible d'appliquer UML 2.0, à l'architecture logicielle, autre que la modélisation d'applications dédiées à la prise en compte des architectures logicielles au sein de MDA.

Un modèle de description d'architecture et d'une démarche d'élaboration d'une architecture logicielle, est étudié. Une approche pour la description d'architecture logicielle baptisée COSA (Component Object based Software Architecture), est présentée. COSA est une approche hybride, basée sur la modélisation par objets et par composants. Pour traiter les interactions entre les composants, COSA défini les connecteurs en tant qu'entités de première classe. Par conséquent, elle encourage la réutilisation non seulement des composants mais également, celle des connecteurs, ainsi que l'amélioration de la réutilisablilité et l'extensibilité des éléments pour soutenir les évolutions dynamiques et statiques des éléments architecturaux. Une solution pour réconcilier les besoins des architectures à base de composants avec les notations orientées objets, rejoint les travaux de Garlan et de son équipe (Garlan 2000a ; Garlan, Cheng, Kompanek, 2001).

L'étude des travaux, dans le domaine des architectures logicielles, défini de façon explicite les concepts et les éléments liés à la description de l'architecture logicielle d'un système. Suite à cette étude, nous avons proposé une architecture à trois niveaux de COSA pour guider l'architecte dans son processus de modélisation. Pour l'élaboration des modèles d'architecture logicielle et leurs inter-relations, cette architecture s'appuie sur trois niveaux d'abstractions:

- niveau méta-architecture,
- niveau architecture,
- niveau application.

Elle présente une spécification complète et structurée de l'architecture logicielle COSA. Elle améliore la réutilisation des architectures logicielles en supportant une hiérarchie pour les trois niveaux conceptuels.

Une proposition d'intégration de l'architecture logicielle COSA au sein de MDA, est établie, ce qui nous a amené à définir une stratégie de transformation directe par l'utilisation du profil UML à l'élaboration des transformations :

- PIM vers PSM, COSA vers UML 2.0

- PSM vers PSM, COSA profil UML vers eCore et COSA profil UML vers CORBA.

L'intérêt de cette solution est de bénéficier des concepts offerts par l'architecture COSA, nécessaires absolument, aux plates-formes d'exécutions objets. Ils augmentent le degré de réutilisabilité des composants au niveau implémentation par l'intégration explicite des connecteurs. Ils réduisent la complexité de contrôle des interactions et des interconnexions entre les composants logiciels. La projection des concepts COSA vers UML est fortement encouragée en raison de la popularité d'UML dans le monde industriel et elle permet de résoudre la coordination des composants distribués dans standard international CORBA. Une solution d'intégration des ADLs au sein de la démarche MDA dont l'intérêt est largement reconnu, dans la communauté scientifique (Marcos, Acuna, Cuesta, 2006; Sánchez et al. 2006) est proposé dans ce travail.

Les réalisations et les expérimentations résultant du profil UML 2.0 pour COSA et l'intégration de ce profil, dans la démarche MDA : PIM vers PSM, COSA vers UML 2.0, PSM vers PSM, COSA profil UML vers eCore et COSA profil UML vers CORBA, validées sur une application de type Client-Serveur, laissent présager des résultats forts intéressants. Une étude comparative entre COSABuilder et ACMEStudio et Fractal, couronne bien ce travail.

Intérêts du travail

Ces intérêts se résument, comme suit :

1- Définition d'un profil UML 2.0 pour l'architecture logicielle COSA. L'intérêt primordial d'un profil UML 2.0 de COSA est d'exprimer les concepts architecturaux en UML 2.0 et donc de définir de manière formelle les concepts de l'architecture logicielle COSA. En d'autre terme, l'utilisation de stéréotypes, de règles et de valeurs marquées permet précisément de mieux capturer les concepts de l'architecture COSA.

2- L'exploitation des capacités de profil UML (modèle et métamodèle) basée sur un ensemble des stéréotypes dotés des contraintes OCL dans la projection de l'architecture logicielle COSA vers UML et vers les plates-formes d'exécution, permet de mieux clarifier la sémantique de ce modèle.

3- L'intégration du profil UML 2.0 pour COSA au sein de la démarche MDA. Les profils UML existants sont dédiés à un type d'application (système distribué, temps réel, etc...) alors que nôtre profil est indépendant et spécialisé dans une activité particulière de modélisation des langages de description d'architectures. L'intégration de ce profil au sein de MDA, confère aux modèles d'architectures objets, un niveau d'abstraction haut et un degré de réutilisation équivalent à celui de l'architecture COSA.

4- Une stratégie de transformation directe par l'utilisation du profil UML à l'élaboration des transformations PIM vers PSM, COSA vers UML 2.0 et PSM vers PSM, COSA profil UML vers CORBA. Notre objectif est d'automatiser le processus de dérivation des plateformes d'implémentation depuis les concepts d'architecture logicielle.

Conclusion

Notre travail, est orienté en deux axes : la définition et la vérification formelle des concepts de l'architecture logicielle et l'intégration des concepts d'architecture logicielle au sein de MDA

Le premier axe concerne la définition formelle des concepts de l'architecture logicielle COSA. En effet, certains concepts tels que les connecteurs, les configurations, la glu, les rôles, les associations d'attachements Binding, Use sont spécifiés formellement. L'exploitation des capacités de profil UML (modèle et métamodèle) pour la définition formelle de ces concepts permet de mieux clarifier la sémantique de ce modèle. Cela prouve la cohérence structurelle des architectures des systèmes logiciels. Le système de type Client-Serveur, nous permet de mieux se rendre compte, des aspects d'architectures logicielles dans les systèmes complexes.

Le deuxième axe concerne l'intégration des concepts d'architecture logicielle au sein de MDA : nous avons montré grâce aux transformations COSA vers UML 2.0, COSA-UML

vers eCore et COSA-UML vers CORBA, qu'il est possible d'automatiser partiellement la prise en compte des architectures logicielles pour leurs aspects structuraux. Il reste encore la prise en compte de leurs aspects dynamiques. Il nous semble très intéressant de générer un sous-ensemble de la partie dynamique. Cependant, pour prendre en compte des aspects dynamiques, il faut nécessairement que l'architecture logicielle COSA gagne en complétude, pour ses aspects dynamiques.

Perspectives

Le travail de thèse a permis la mise en place d'un cadre conceptuel pour la modélisation des architectures logicielles. Il s'agit d'une définition d'un profil UML pour l'architecture logicielle (particulièrement COSA) et ensuite l'intégration de ce profil dans la démarche MDA. Cette thèse peut se prolonger vers plusieurs perspectives de recherche qu'on peut résumer autour de cinq points.

- **Styles Architecturaux**

Ce premier point concerne l'amélioration de l'approche COSA en définissant des contraintes et des règles spéciales pour supporter les styles architecturaux. Comme nous l'avons vu, dans le chapitre 2, les styles architecturaux définissent une famille de systèmes en terme de patterns d'organisation structurelle. Ils déterminent l'ensemble du vocabulaire désignant le type des entités de base (composants et connecteurs) tels que pipe-filtre, client-serveur, événement, processus, tableau noir, etc. Si nous intégrons des styles architecturaux à COSA, les architectures hétérogènes seront plus faciles à définir. Nous pouvons aussi définir de nouvelles architectures en composant différents styles et de des nouvelles métriques de qualité architecturale tels que la complexité structurelle, le couplage et la cohésion.

- **Mécanisme de raffinement**

Il est fortement souhaitable de fournir un mécanisme de raffinement pour l'approche COSA. Ce point concernant la définition des patrons des connecteurs architecturaux pour le développement des architectures à base d'UML. L'utilisation du langage OCL permet la vérification formelle des connecteurs architecturaux et l'application

progressive des patrons des connecteurs permet de pallier au problème de traçabilité entre l'architecture et son système exécutable.

- **Description formelle des modèles**

Il est important de formaliser les concepts du modèle COSA et du méta-modèle MADL (Meta Architecture Description Language) (Smeda, Oussalah, Khammaci, 2005b). Un premier effort avec une esquisse d'utilisation OCL a été réalisé dans le chapitre 5. Toutefois cet effort doit être poursuivi pour la description formelle de nos deux modèles, comme avec, la méthode B (Abrial, 1996 ; Abrial, 1999) ou CSP (Hoare, 1995) tel qu'il est utilisé dans Wright (Allen, Garlan, 1997). Une telle technique permet de mieux préciser les concepts proposés dans COSA et MADL et d'enlever les ambiguïtés dans la définition de leurs concepts. En outre, l'utilisation du la méthode B dans la projection de COSA vers UML permet de mieux clarifier la sémantique ou les variantes sémantiques de ces modèles. Ce travail fait l'objet actuellement une suite de notre thèse.

- **Intégration sémantique des ADLs au sein de MDA par une ontologie**

L'idée ici serait de généraliser notre approche d'intégration des concepts d'architectures logicielles pour n'importe quel ADL (COSA, ACME, Fractal, etc.) et pour n'importe quelle plateforme d'exécution (CORBA, EJB, .NET). Si nous utilisons une ontologie architecturale, les architectures hétérogènes seront plus faciles à intégrer.

- **Description contextuelle des modèles**

Ce point concerne l'extension des concepts du COSA pour prendre en compte la qualité d'une architecture logicielle et le contexte (ressources, préférences de l'utilisateur et les contraintes).

Bibliographie

Abrial J.R., (1996). The B-Book: Assigning Programs to Meanings, Cambridge University Press.

Abrial J.R., (1999). Introducing Dynamic Constraints in B, In D. Bert (Ed.), B'99: Recent Advances in the Developement and Use of the B Method, (LNCS: Springer-Verlag, 1998, pp 83-128).

ADL Toolkit., (2004). http://www-2.cs.cmu.edu/~acme/adltk/tools.html

Accord., (2002). Etat de l'Art sur les Langages de Description s'Architecture (ADLs), Rapport technique, INRIA, France.

Aldrich J., Chambers C., Notkin D., (2001). Component-Oriented Programming in ArchJava, Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components, Tampa Bay, Florida.

Aldrich J., Chambers C., Notkin D., (2002). ArchJava: Connecting Software Architecture to Implementation, Proceedings of the 24th International Conference on Software Engineering (ICSE'02), Orlando (USA).

Aldrich J., Sazawal V., Chambers C., Notkin D., (2003). Language Support for Connector Abstractions, Proceedings of the 2003 European Conference on Object-Oriented Programming (ECOOP'03), Darmstadt (Germany).

Allen R., Garlan G., (1994). Formalizing Architectural Connection, Proceedings of the Sixteenth International Conference on Software Engineering (ICSE'94), Sorrento (Italy).

Allen R., Garlan G., (1997). A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3, pp 213-249.

Allen R., Douence R., Garlan D., (1998). Specifying and Analyzing Dynamic Software Architecture, Proceedings of the Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal.

Alloui I., Oquendo F., (2004). Describing Software-intensive Process Architectures using a UML-based ADL. In Proceedings of the Sixth International Conference on Enterprise Information Systems (ICEIS'04), Porto (Portugal), pp 201-208.

Alti A., Khammaci T., (2005). Transformation des concepts des composants architecturaux en UML 2.0. 1er Congrès International en Informatique appliquée (CIIA'05), pp 14 -21, B.B.A Algérie, ISBN: 9947-0-1042-2, 19 -21 novembre.

Alti A., Khammaci T., Smeda A., (2006). Building UML Profile for COSA Software Architecture. 2nd IEEE Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'06), ISBN: 0-7803-9521-2, Damascus (Syria), Vol. 2, pp 1041 -1042, 24-28 April.

Alti A., Khammaci T., Smeda A., (2007a). Representing and Formally Modelling COSA Software Architecture with UML 2.0 Profile, International Review on Computers and Software, ISSN: 1828-6003, Vol. 2, No. 1, pp 30-37.

Alti A., Khammaci T., Smeda A., (2007b). Towards an Automatic Transformation From COSA Software Architecture To CORBA platform, First International Conference on Digital Communications and Computer Applications (DCCA'07), ISBN: 0-7803-9521-2, Just (Jordan), pp 980 – 989, 19 -21 Mars.

Alti A., Khammaci T., Smeda A., Bennouar D., (2007). Integrating Software Architecture Concepts into the MDA Platform. The Second International Conference on Software and Data Technologie (ICSOFT'07), INSTICC Press, ISBN: 978-989-8111-05-0, Barcelona (Espagne), INSTICC Press, Vol. 1, pp 30-37, 19-49 July.

Alti A., Smeda A., (2007). Intégration de l'architecture logicielle COSA au sein de MDA : Retour d'expériences, Actes de 6éme atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, held in conjunction with INFORSID 2007, Perros-Guirec, France, pp 144-156, 22 -25 Mai.

Alti A., (2008). An Automatic Transformation from COSA Software Architecture to EJB Platform. 3rd IEEE Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'08), ISBN: 978-1-4244-1751-3, Damascus (Syria), Vol. 3, pp 1 -6, 7-11 April.

Alti A., Smeda A.,(2008). Integration of Architectural Design and Implementation Decisions into the MDA Framewrok. The Third International Conference on Software and Data

Technologie (ICSOFT'08), ISBN: 978-989-8111-52-4, INSTICC Press, Porto (Portugal), pp 366-371, July 5-8 2008.

Alti A., Djoudi M., (2009). Patrons des Connecteurs Architecturaux pour le Développement des Architectures à base d'UML, 1er Conférence Internationale des Technologies de l'Information et de la Communication (CITIC'09), Sétif (Algérie), pp 58, 4 -5 Mai.

Alti A., Boukerram A., Smeda A., Maillard S., Oussalah M., (2010). COSABuilder and COSAInstantiator: An Extensible Tool for Architectural Description, International Journal of Software Engineering and Knowledge Engineering, ISSN : 0218-1940, Vol. 20, No. 3, pp 423-455.

Alti A., Boukerram A., Roose P,. (2010). Context-Aware Quality Model Driven Approach: A New Approach for Quality Control in Pervasive Computing Environments. The 4th European Conference on Software Architecture (ECSA 2010), LNCS 6285 Springer, ISBN 978-3-642-15113-2, Copenhagen (Denmark), pp 441-448, 23 -26 August.

Atlas group LINA, INRIA Nantes, (2007) ATL: Atlas Transformation Language, ATL User Manual version 7.0.

Amirat A., Oussalah M., (2009). Towards an UML Profile for the Description of Software Architecture, Proceedings of the First International Conference on Applied Informatics (ICAI'09), Bou Arréridj (Algeria), pp , 15-17, November 2009

ArchJava., (2004). Disponible sur http://archjava.fluid.cs.cmu.edu.

Baker S., CORBA Distributed Objects Using Orbit, Addison Wesley, 1997.

Balter R., Bellissard F., Boyer F., Riveill M., Vion-Dury J., (1998). Architecturing and Configuring Distributed Applications with Olan, Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, UK, pp 241-256.

Barbier F., Cauvet G., Oussalah M., Rieu D., Souveyet C., (2004). Concepts Clés et Techniques de Réutilisation dans l'Ingénierie des Systèmes d'Information. Ingénierie des Composants dans les Systèmes D'informations, M. Oussalah et D. Rieu (réd.), Hermes Science Publications.

Bartlett D., (2004). OMG Interface Definition Language, Defining the Capabilities of a Distributed Service, Rapport technique, http://www-128.ibm.com/devloperworks/webservices/library/co-corbajct3.html.

Bass L., Clements P., R. Kazman, (1998). Software Architecture in Practice. Addison-Wesly Publishing, Reading, Massachusetts.

Bézivin J., (2005). Model Driven Engineering: An Emerging Technical Space, in *Proc. Generative and Transformational Techniques in Software Eng.*, Braga, Portugal, pp. 36-64, July 4 -8

Booch G., (1994). Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company Inc., Redwood City, California.

Booch G., Rumbaugh J., Jacobson I., (1998). The Unified Modeling Language User Guide. Addison-Wesly Publishing, Reading, Massachusetts.

Brinkkemper S., Hang S., Bulthuis A. and van den Gor G., (1995). Object-Oriented Analysis and Design Methods: A Comparative Review. Disponible sur http://elex.amu.edu.pl/languages/oodoc/oo.html.

Bruneton E., (2004). Developing with Fractal, the ObjectWeb Consortium, disponible sur http://fractal.objectweb.org/tutoriel/2004

Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., Stafford J., (2002). Documenting Software Architectures: Views and Beyond. Boston, MA: Addison-Wesley.

Clements P., (2003). Documenting Software Architectures: Views and Beyond. Addison-Wesly Publishing, Reading, Massachusetts.

Cheng S., Garlan D., (2001). Mapping architectural Concepts to UML-RT, Proceedings of the Parallel and Distributed Processing Techniques and Applications, USA.

Ciancarini P., Mascolo C., (1996).Analyzing and Refining an Architectural Style.

Coupaye D., Bruneton E. Stefani J., (2002). The Fractal Composition Framework, Proposed Final Draft of Interface Specification version 0.9, the ObjectWeb Consortium**.**

Cox B.J., (1986). Object-Oriented-Programming-an Evolutionary Approach, Addison-Wesly ISBN: 0-201-10393-1.

DeRemer F., Kron H., (1976). Programming-in-the-Large Versus Programming-in-the-Small, IEEE Transactions on Software Engineering SE-2, Vol. 2, pp 321-327.

Eclipse,(2006).http://www.openarchitectureware.org/pub/documentation/4.3.1/html/conte/ts/uml2ecore_reference.html

Egyed A., Krutchen P., (1999). Rose/Architect: a tool to visualize architecture. Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences (HICSS'99).

FractalGUI., (2005).http://fractal.ow2.org/fractalgui/index.html

Frankel D., (2003). Model Driven Architecture, Applying MDA to Enterprise Computing, Wiley, Indianapolis, USA.

Gammma E., Helm R., Johnson R., Vlissides J., (1995). Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesly Publishing, Reading, Tortora (réd.), World Scientific Publishing Company, Singapore, pp 1-39.

Garlan D., (1995). What is Style? , Proceedings of the Dagstuhl Workshop on Software Architecture, Saarbruecken (Germany).

Garlan D., (2000a). Software Architecture and Object-Oriented Systems. Proceedings of the IPSJ Object-Oriented Symposium 2000, Tokyo, Japan.

Garlan D., (2000b). Software Architecture: A Roadmap, In ICSA'2000, 22nd International Conference on Software Engineering, pp 91-101.

Garlan D., Shaw M., (1993). An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora (réd.), World Scientific Publishing Company, Singapore, pp 1-39.

Garlan D., Allen R., Ockerbloom J., (1994). Exploiting Style in Architectural Design Environments, Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, pp 175 – 188.

Garlan D., Perry D., (1995). Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, Vol. 21, No. 4.

Garlan D., Monroe R., Wile D., (1997). Acme: An Architecture Description Interchange Language, Proceedings of CASCON'97, Toronto, Canada.

Garlan D., Monroe R., Wile D., (2000). Acme: Architectural Description of Component-Based Systems, Leavens Gary and Sitaraman Murali, Foundations of Component-Based Systems, Cambridge University Press, pp.47-68.

Garlan D., Monroe R., Wile D., (2000). Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, Leavens Gary et Sitaraman Murali (réd.) Cambridge University Press, pp 47-68.

Garlan D., Chang S., Kompanek J., (2001). Reconciling the Needs of Architectural Description with Object-Modeling Notations. Science of Computer Programming Journal 44 (Elsevier Press, 2001), pp 23-49.

Garlan D., Cheng S.W., Kompanek A., (2002). Reconciling the Needs of Architectural Description with Object-Modeling Notations. Science of Computer Programming Journal, Special UML, Edition 44, Elsevier Press, pp 23-49.

GMF, (2006). Introduction to the Eclipse Graphical Modeling Framework, Eclipse, IBM, and Borland, Editors. EclipseCon 2006, available at http://bdn1.borland.com/devcon05/article/1,2006,33309,00.html.

Goulao M., Abreu F.B., (2003). Bridging the gap between ACME and UML 2.0 for CBS, In Proceedings Workshop of Specification and Verification of Component-Based Systems, Helsinki, Finland.

Graiet M., Bhiri M.T., Dammak F., Giraudin J.P., (2006). Adaptation d'UML 2.0 à l'ADL Wright. 1re Conférence Francophone sur les Architectures Logicielles, ISSN: 7462-1577, pp 83- 96.

Grim R., (1997). Professional DCOM Programming, First edition, Peer Information Inc., ISBN: 186100060X.

Hasselbring W., (2002). Component-Based Software Engineering, Handbook of Software Engineering and Knowledge Engineering, 2 ed. S.K. Chang (World Scientific Publishing Company, Singapore), pp 289-49.

Hoare C., (1995). Communicating Sequential Processes, Prentice Hall International, Englewood Cliffs, N.J, ISBN: 0131532718.

Hofmeister C., Nord R.L., Soni D., (1999). Describing Software Architecture with UML. Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, IEEE Computer Society Press, pp 145-160.

ISO-IEC, (2001). ISO/IEC 9126-1 in Software Engineering – Product quality – Part 1: Quality model.

Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J.R., (2004). Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, School of Computer Science, Carnegie Mellon University.

Jacobson I., Chirterson M., Jonsson P., et Overgaard G., (1992). Object-Oriented Software Engineering, Addison-Wesly Publishing, Reading, Massachusetts.

Kazman R., O'Brien L., Verhoef C.,(2001). Architecture Reconstruction Guidelines, Technical report.

Khammaci T., Oussalah M., Smeda A., (2003). Les ADLs : Une Voie Prometteuse Pour les Architectures Logicielles, Agents, Logiciels, Coopération, Apprentissage & Activité Humaine 2003 (ALCAA'03), Bayonne, France.

Khammaci T., Smeda A., Oussalah M., (2004). ALBACO : Une Architecture Logicielle à Base de Composants et d'Objets pour la Description de Systèmes Complexes. Proceedings of the 8th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI 2004), Sousse, Tunisie.

Khammaci T., Smeda A. Oussalah M., (2005). Coexistence of Software Architecture and Object Oriented Modeling, Handbook of Software Engineering and Knowledge Engineering Volume 3: Recent Advances, S.K. Chang (réd.), World Scientific Publications Co., Singapore, ISBN : 98125627737, pp 119-150.

Krutchen P.B., (1995). The 4+1 View Model of architecture. IEEE Transaction on Software Engineering, Vol. 12, No. 6, pp 42-50.

Kruchtan P.B., Selic B., Kozaczynski W., (2001). Describing Software Architecture with UML. Proceedings of the 23rd International Conference on Software Engineering, pp 715-716.

Luckham D., Augustin L., Kenny J., Vera J., Bryan D. et Mann W. (1995). Specification and Analysis of System Architecture using Rapide, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp 336-355.

Maillard S., Smeda A., Oussalah M., (2007). COSA: An Architectural Description Meta-Model. Proceedings of the Second International Conference on Software and Data Technologies, Volume SE, Barcelona (Spain), INSTICC Press 2007, ISBN 978-989-8111-06-7, pp 445-448, July 22-25.

Magee J., Dulay N., Eisenbach S., and Karmer J., (1996). Specifying Distributed Software Architectures. Proceedings of the Fifth European Software Engineering Conference, Barcelona (Spain), September 1995.

Magee J., and Karmer J., (1996). Dynamic Structure in Software Architecture. Proceedings of ACM SIGSOFT'96: Fourth Symposium Foundation of Software Engineering, San Francisco, CA, pp 3-14.

Mcllroy D., (1968). Mass-produced Software Components.1st International Conference on Software Engineering, Garmish Pattenkirschen, Germany.

Magee J., Dulay N., Eisenbach S., Karmer J., (1995). Specifying Distributed Software Architectures, Proceedings of the Fifth European Software Engineering Conference, Barcelona (Spain).

Magee J., Karmer J., (1996). Dynamic Structure in Software Architecture, Proceedings of ACM SIGSOFT'96 : Fourth Symposium Foundation of Software Engineering, San Francisco, CA, pp 3-14,.

Manset D., Verjus H., McClatchey R., Oquendo F. (2006), A Formal Architecture-Centric, Model-Driven Approach for the Automatic Generation of Grid Applications, Proceedings of the 8th International Conference on Enterprise Information Systems, ICEIS, Paphos, May.

Marcos E., Acuňa C.J., Cuesta C.E., (2006). Integrating Software Architecture into a MDA Framework. In EWSA'2006, 3th European Workshop on Software Architecture. Nantes, France, pp128 -143.

Martin J., (1996). Principles of Object-Oriented Analysis and Design, Prentice-Hall, Englewood Cliffs, New Jersy, CA, pp 3-14.

Medvidovic N., Oreizy P., Robbins J., Taylor R., (1996). Using Object-Oriented Typing to Support Architectural Design in the C2 Style. Proceedings of ACM/IGSOFT'96: Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA.

Medvidovic N., Taylor R., Whitehead E., (1996). Formal Modeling of Software Architecture at Multiple Levels of Abstraction, Proceedings of the 1996 California Software Symposium, Los Angles, CA, pp 28-40.

Medvidovic N., Rosenblum D., Taylor R., (1999). A Language and Environment for Architecture-Based Software Development and Evolution, Proceedings of 21st International Conference on Software Engineering, Los Anglos, CA, pp 44-53.

Medvidovic N., Taylor R., (2000). A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, No. 1, pp 2-57.

Medvidovic N., Rosenblum D.S., Robbins J.E., Redmiles D.F., (2002). Modeling Software Architecture in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol.11, No. 1, pp 2-57.

Mehta N., Medvidovic N., Phadke S., (2000). Twords a Taxonomy of Software Connectors, 22nd International Conference on Software Engineering, Limerick (Ireland).

Model Driven Architecture., (2002). http://www.omg.org/mda

Murata T., (1989). Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, Vol.77, No. 4, pp 541-580.

Object Management Group., (1997). UML Semantics and Appendices v1.1, disponible sur http://www.omg.org/docs/ad/97-09-07.pdf .

Object Management Group., (2001). Unified Modeling Language Specification V.1.4., http://www.omg.org/docs/formal/01-09-67.pdf.

Object Management Group., (2002). CORBA Components: An Adopted Specification". http://www.omg.org/docs/formal/02-06-66.pdf.

Object Management Group., (2004a). UML 2.0 Superstructure Specification: Revised Final Adopted Specification. http://www.omg.org/docs/ptc /04-10-02.pdf .

Object Management Group., (2004b). UML Profile for CORBA Components Specification. http://www.omg.org/docs/ptc /04-03-04.pdf

Object Management Group., (2005). UML OCL 2.0 Specification: Revised Final Adopted Specification. http://www.omg.org/docs/ptc /05-06-06.pdf.

Object Management Group., (2008). Catalog of UML Profile Specifications. http://www.omg.org/technology/documents/profile_catalog.htm .

Oquendo F., Cîmpan S., Balasubramaniam D., Kirby G., Morrison R., (2002). The ArchWare ADL: Definition of the Textual Concrete Syntax, Technical Report D1.2b, ArchWare Project IST-2001-32360.

Oquendo F., (2004). Π-ADL: an Architecture Description Language Based on the Higher Order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes, ISSN: 0163-5948, Vol. 29. No. 3, pp 15-28.

Oquendo F., (2006). Formally Modelling Software Architecture with the UML 2.0 Profile For π-ADL, ACM SIGSOFT Engineering Notes, ISSN: 0163-5948, Vol. 31, No. 1, pp 1-13.

Oussalah C., Smeda A., Khammaci T., (2004). An Explicit Definition of Connectors for Component, Proceedings 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2004), pp 44-51, Brno (Czech Republic).

Oussalah C., Khammaci T., Smeda A., (2005). Les composants : définitions et concepts de base, dans Ingénierie des composants, Editions Vuibert, ISBN 2-7117-4836-7.

Perry D., (1997). Software Architecture and its Relevance to Software Engineering, Invited talk, Proceedings of the Second International Conference on Coordination Models and Languages, Berlin (Germany).

Perry D., Wolf A., (1992). Foundations for Study of Software Architecture, ACM/SIGSOFT Software Engineering Notes, Vol. 17, pp 40-52.

Pilone D., Pitman N., (2005). UML 2.0 in a Nutshell, O'Reilly & Associates, Portland, Oregon, ISBN: 0596007957.

Plasil F., Balek M., Janecek R., (1998). SOFA/DUCP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, IEEE CS Press, Annapolis, Maryland, USA, 1998.

Plasil F., Besta M., Visnovsky S., (1999). Bounding Component Behavior via Protocols, Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99), Santa Barbara, USA.

Ratcliffe O., (2005). Approche et Environnement Fondés sur les Styles Architecturaux pour le Développement de Logiciels Propres à des Domaines Spécifiques, Thèse de Doctorat, Université de Savoie, Annecy.

Rodrigues M.N., Lucena L., Batista T., (2004). From Acme to CORBA: Bridging the Gap. In EWSA'2004, the 1st European Workshop on Software Architecture, pp. 103-114.

Roman E., Amber S., Jewell T., Marinescu F., (2004). Mastering Entreprise JavaBeans, 3rd Edition, John Willey et Sons, Canada, ISBN 0764576828.

Roh S., Kim K., Jeon T., (2004).Architecture Modeling Language based on UML 2.0, Proceedings of the 11 th Asia-Pacific Software Engineering Conference (APSEC'04).

Rosenblum D., Medvedovic N., (1997). Domain of Concern in Software Architectures and Architecture Description Languages, Proceedings of the USENIX Conference on Domain Specific Languages, CA.

R-software., (2008). The R Project for Statistical Computing, http://www.r-project.org

Rubin W., Brain M., (1998). Understanding DCOM, Prentice Hall Englewood, Cliffs, New Jersy, ISBN: 0130959669.

Rumbaugh J., Blaha M., Permerlani W., Eddy F. et Lorensen W., (1991). Object-Oriented Modeling and Design, Practice-Hall, Englewood Cliffs, New Jersey, ISBN: 0136298419.

Sadou N., Oussalah M., Tamzalit D., (2005). Software Architecture Evolution: Description and Management Process, Proceedings of The 2005 International Conference on Software Engineering Research and Practice (SERP'05), Las Vegas, Nevada.

Sadou N., Tamzalit D., Oussalah M., (2005). A Unified Approach for Software Architecture Evolution at Different Abstraction Levels, Proceedings of International Workshop on Principals of Software Evolution, Lisbon, Portugal.

Sánchez P., Magno J., Fuentes L., Moreira A., Araújo J., (2006). Towards MDD Transformation from AORE into AOA. In EWSA'2006, Proceedings of the 3th European Workshop on Software Architecture. France, pp 159 -174.

Selic B., (1999). Turing Clockwise: Using UML in the Real-Time Domain. Communication of the ACM, Vol. 42, No. 10, pp 46-54.

Selic B., Rumbaugh J., (2000). Using UML for Modeling Complex Real-Time Systems, Object Time white paper, June 2000, http://www.objecttime.com

Shaw M., DeLine R., Klein G., Ross G., Young D., Zelesnik G., (1995). Abstractions for Software Architecture and Tools to Support Them. IEEE Transaction on Software Engineering, Vol. 21, No. 4.

Shaw M., DeLine R., Zelesnik G., (1996). Abstractions and Implementations for Architectural Connections. Proceedings of the 3rd International Conference on Configurable Distributed Systems, Annapolis, Maryland.

Shaw M., Garlan D., (1996). Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Upper Saddle River, N.J., ISBN: 0131829572.

Smeda A., Khammaci T., Oussalah M., (2003). Software Connectors Reuse in Component-Based Systems, Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI'03), Las Vegas, Nevada.

Smeda A., Khammaci T., Oussalah M., (2004a). A Multi-Paradigm Approach to Describe Complex Software System, Journal of WSEAS Transactions on Computers, ISSN: 1109-2750, Vol. 3, No. 4, pp 936-941.

Smeda A., Khammaci T., Oussalah M., (2004b). Operational Mechanisms for Component-Object Based Software Architecture. First IEEE Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'06), Damascus (Syria).

Smeda A., Oussalah M., Khammaci T., (2005a). Mapping ADLs into UML 2.0 Using a Meta ADL, Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), IEEE CS. Press, ISSN: 1109-2750, Pittsburg (Pannsylvania), Vol. 3, No. 4, pp 936-941.

Smeda A., Oussalah M., Khammaci T., (2005b). MADL: Meta Architecture Description Language, Proceedings of SERA 2005, IEE CS Press, Pleasant, Michigan, USA.

Smeda A., Khammaci T., Oussalah M., (2005c). Les Modèles de Composants Académiques, Ingénierie des Composants : Concepts, Techniques et Outils, M. Oussalah (réd). Vuibert Informatique, Paris, ISBN 2711748367.

Smeda A., Alti A., Oussalah M., Boukerram A., (2009). COSAStudio: A Software Architecture Modeling Tool, World Congress on Science, Engineering And Technology (WCSET'09), ISSN: 1307 - 6892, No. 49, pp 263-266, 28 -30 Januray 2009.

Soni D., Nord R.L., Hofmeister C., (1995). Software Architecture in Industrial Applications, Proceedings of the 17th International Conference on Software Engineering (ICSE'17), Seattle, WA, USA (1995) pp 196-207.

Soucé J., Duchien L., (2002). Etat de l'Art sur les Langages de Description d'Architecture (ADLs), CNAM-CEDRIC, USTL-LIFL, RNTL ACCORD.

Taylor R., (1996). A Component-and Message- Based Architectural Style for GUI Software, IEEE Transactions on Software Engineering, Vol.22, No. 6.

UML Toolkit., (2005). http://www.netfective.com/?cID=produits0

Van Ommering R., Van Der Linden Karmer F., J., Magee J., (2000). The Koala Component Model for Consumer Electronics Software, IEEE Computer, p. 78 -85.

Vestal S., (1996). MetaH Programmer's Manual 1.09, Rapport Technique, Honeywell Technology Center, Minneapolis, MN.

Villa T., Kam T., Brayton R., Sagiovanni-Vincentelli A., Sangiovanni-Vincentelli A.,(1997). Synthesis of Finite State Machines : Logic Optimization, Kluwer Academic Publishers, Philip Drive/ Norwell, MA, ISBN:0792398920.

Warmer J., Kleppe A., (1998). The Object Constraint Language: Precise Modeling with UML, Addison-Wesly Publishing, Reading, Massachusetts.

Liste des publications personnelles

Chapitres dans des ouvrages

Alti A., Boukerram A., Roose P., (2010). Context-Aware Quality Model Driven Approach: A New Approach for Quality Control in Pervasive Computing Environments. In Muhammad Babar and Ian Gorton, editors, Software Architecture, Volume 6285 of Lecture Notes in Computer Science Springer, ISBN 978-3-642-15113-2, Copenhagen (Denmark), pp 441-448, August 23 – 26, 2010. *indexé par ISI-JCR (Thomson Reuters).*

Publications dans des revues internationales

Alti A., Khammaci T., Smeda A., (2007). Representing and Formally Modelling COSA Software Architecture with UML 2.0 Profile, International Review on Computers and Software, ISSN: 1828-6003, Vol. 2, No. 1, pp 30-37, January 2007.

Alti A., Khammaci T., Smeda A., (2007). Using B Formal Method to Define Software Architecture Behavioral Concepts, International Review on Computers and Software, ISSN: 1828-6003, Vol. 2, No. 5, pp 510-519, September 2007.

Alti A., Khammaci T., Smeda A., (2007). Integrating Software Architecture Concepts into the MDA Platform with UML Profile, Journal of Computer Science, ISSN: 1549-3636, Vol. 3, No. 10, pp 793-802, October 2007.

Makhlouf D., Roose P., Dalmau M., Ghoualmi-Zine N., **Alti A.,** (2010). Vers une architecture d'adaptation automatique des applications reparties basées composants - Revue des Nouvelles Technologies de l'Information, Conférence francophone sur les Architectures Logicielles (CAL 2010), ISSN : 174 – 1667, pp. 1-14, March 2010.

Alti A., Boukerram A., Smeda A., Maillard S., Oussalah M., (2010). COSABuilder and COSAInstantiator: An Extensible Tool for Architectural Description, International

Journal of Software Engineering and Knowledge Engineering, ISSN: 0218-1940, Vol. 20, No. 3, pp 423-455, May 2010. *ISI-JCR Facteur d'impact (2008):0.447*.

Makhlouf D., Roose P., Dalmau M., Ghoualmi Z., **Alti A**., (2010). UML-Profile for Multimedia Software Architectures - International Journal of Multimedia Intelligence and Security (IJMIS) Inderscience Publishers, Vol. 1, No. 3, pp. 209-231, 2010. *indexé par ISI (Thomson Reuters)*.

Makhlouf D., Roose P., Dalmau M., Ghoualmi-Zine N., **Alti A.**, (2010). MMSA: Metamodel Multimedia Software Architecture - Advances In Multimedia, Hindawi Ed. ISSN : 1687-5680, DOI 10-1155, e-ISSN: 1687-5699, pp. 1-17, July 2010. indexé par ISI (Thomson Reuters).

Makhlouf D., Roose P., Dalmau M., Ghoualmi-Zine N., **Alti A.**, (2010). Une approche pour les architectures logicielles à composants multimédia – An International Journal on Information - Interaction – Intelligence, ISSN 0249-6399, Vol. 10 N. 2, 2010.

Communications dans des conférences internationales

Alti A., Khammaci T., Smeda A., (2006). Building UML Profile for COSA Software Architecture. 2nd IEEE Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'06), ISBN: 0-7803-9521-2, Damascus (Syria), Vol. 2, pp 1041 -1042, April 24-28, 2006.

Alti A., Khammaci T., Smeda A., (2007). Towards an Automatic Transformation From COSA Software Architecture To CORBA platform, First International Conference on Digital Communications and Computer Applications (DCCA'07), ISBN: 0-7803-9521-2, Just (Jordan), pp 980 – 989, 19 -21 Mars, 2007.

Alti A., Khammaci T., Smeda A., Bennouar D., (2007). Integrating Software Architecture Concepts into the MDA Platform. The Second International Conference on Software and Data Technologie (ICSOFT'07), ISBN: 978-989-8111-05-0, Barcelona (Espagne), INSTICC Press, Vol. 1, pp 30-37, July 19-49, 2007.

Alti A., (2008). An Automatic Transformation from COSA Software Architecture to EJB Platform. 3rd IEEE Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'08), ISBN: 978-1-4244-1751-3, Damascus (Syria), Vol. 3, pp 1 -6, April 7-11 2008.

Alti A., Djoudi M., (2008). Formal Specification and Verification of Software Architecture Behavioral Concepts using UML 2.0 Profile and B. The 10th Maghrebian Conference on Information Technologies (MCSEAI'08), Oran (Algeria), pp 1 -6, April 28-30 2008.

Alti A., Smeda A., Djoudi M., (2008). An Automatic Transformation from COSA Software Architecture to CORBA Platform. The 8th International Conference on New Technologies in Distributed Systems (NOTERE'08), ACM 978-1-59593 – 937 -1, Lyon (France), Vol. 1, pp 30-37, June 23-27 2008.

Alti A., Smeda A.,(2008). Integration of Architectural Design and Implementation Decisions into the MDA Framewrok. The Third International Conference on Software and Data Technologie (ICSOFT'08), ISBN: 978-989-8111-52-4, INSTICC Press, Porto (Portugal), pp 366-371, July 5-8 2008.

Alti A., Smeda A., (2008). A Meta-ontology for Architecture Description Languages and MDA Platforms. International Conferences on Innovation in Software Engineering (ISE'08), IEEE Computer Society, ISBN 978-0-7695-3514-2, Vienna (Austria), pp 118-123, 10–12 December 2008.

Alti A., Djoudi M., (2009). Patrons des Connecteurs Architecturaux pour le Développement des Architectures à base d'UML, 1er Conférence Internationale des Technologies de l'Information et de la Communication (CITIC'09), Sétif (Algérie), pp 58, 4 -5 Mai 2009.

Smeda A., **Alti A.**, Boukerram A., (2009). A Software Architecture Modeling Tool. The 8th WSEAS International Conference on TELEcommunications and INFOrmatics (TELE-INFO '09), ISSN: 1790 – 5117, Istanbul (Turkey), pp 68 – 72, 30 Mai - 1 June 2009.

Alti A., Smeda A.,(2009). Architectural Styles Quality Evaluation and Selection. The 4th International Conference on Software and Data Technologie (ICSOFT'09), INSTICC Press 2009, ISBN: 978-989-674-009-2, Sofia (Bulgaria), INSTICC Press, pp 74-82, July 26-29 2009.

Makhlouf D., Roose P., Dalmau M., Ghoualmi-Zine N., **Alti A.**, (2010). Vers une architecture d'adaptation automatique des applications reparties basées composants - Conférence sur les Architectures Logicielles (CAL 2010) , pp. 1-14, RNTI L-5 - ISSN : 174 - 1667 - ISBN : 978-2-85128-930-5, Pau, France, 9 – 12 March 2010.

Alti A., Boukerram A., Makhlouf D., Roose P., (2010). Context-aware Quality Model Driven Approach. The 10th International Conference on New Technologies in Distributed

Systems (NOTERE 2010), IEEE CFP1090J -PRT, ACM Tunisia Chapter, ISBN:978-1-4244-7066-2, Tozeur (Tunisia), pp 197-204, May 31 – June 2, 2010.

Alti A., Boukerram A., (2010). QualiStyle: A Tool for Automatic Quality Evaluation and Selection of Architectural Styles. The 10th International Conference on New Technologies in Distributed Systems (NOTERE 2010), IEEE CFP1090J -PRT, ACM Tunisia Chapter, ISBN: 978-1-4244-7066-2, Tozeur (Tunisia), pp 243-249, May 31 – June 2, 2010.

Makhlouf D., Roose P., Dalmau M., Ghoualmi-Zine N., **Alti A.**, (2010). An adaptation approach for component-based software architecture, The 34th Annual IEEE Computer Software and Application Conference (COMPSAC 2010), ISBN: 0730-315-10, pp. 179-187, Seoul (South Corea), July 19 – 23 2010.

Alti A., Boukerram A., (2010). Semantic Mapping of ADLs into MDA Platforms Using a Meta-Ontology. The First International Conference on Machine and Web Intelligence (IEEE ICMWI 2010), IEEE CFP1023L6-CDR, ISBN: 978-1-4244-8610-6, Algiers (Algeria), pp 405-412, October 3-5, 2010.

Alti A., Boukerram A., (2010). CxQWS: ConteXt-aware Quality semantic Web Service. The First International Conference on Machine and Web Intelligence (IEEE ICMWI 2010), IEEE CFP1023L6-CDR, ISBN: 978-1-4244-8610-6, Algiers (Algeria), pp 42-49, October 3-5.

Communications dans des workshops internationaux (avec comité de lecture)

Alti A., Smeda A., (2006). Un profil UML 2.0 pour l'architecture logicielle COSA, 5éme atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, held in conjunction with INFORSID 2006, Hammamet (Tunisie), pp 62-73, 1-3 Juin 2006.

Alti A., Smeda A., (2007). Intégration de l'architecture logicielle COSA au sein de MDA : Retour d'expériences, 5éme atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, held in conjunction with INFORSID 2007, Perros-Guirec (France), pp 62-73, 22 Mai 2007.

Communications dans des congrès internationaux (avec comité de lecture)

Alti A., Khammaci T., (2005). Transformation des concepts des composants architecturaux en UML 2.0. 1er Congrès International en Informatique appliquée (CIIA'05), ISBN: 9947-0-1042-2, pp 14 -21, B.B.A (Algérie), 19 -21 novembre 2005.

Smeda A., **Alti A.**, Oussalah M., Boukerram A., (2009). COSAStudio: A Software Architecture Modeling Tool, World Congress on Science, Engineering And Technology (WCSET'09), ISSN: 1307 - 6892, No. 49, pp 263-266, 28 -30 Januray 2009.

Liste des outils COSA

Profil UML 2.0 pour l'architecture logicielle COSA, COSAPlug-In

L'outil COSAPlug-In écrit en java a été implémenté dans la plate forme de développement logiciel d'IBM RSM pour Eclipse. COSAPlug-In permet d'instruire les architectes et les designers aux solutions orientées architecture selon l'approche COSA, et utilisé avec succès à la création des modèles d'architectures logicielles des systèmes complexe. Un Plug-In qui fournit un profil UML 2.0 pour COSA contient un ensemble de stéréotypes avec toutes ses valeurs marquées et contraintes OCL 2.0 nécessaire à la validation pratique de la projection COSA vers UML 2.0. COSAPlug-In offre les fonctionnalités suivantes:

- Performance de vérification des modèles d'architecture UML profilé COSA.
- Simplicité de création et rapidité de modification d'une architecture logicielle par la création du modèle UML 2.0 (diagramme de composants) et ensuite ajouté les contraintes OCL nécessaires au modèle. Après que, le modèle évalué par le profil.
- Validation sémantique et vérification de cohérence structurelle des modèles d'architecture UML 2.0 profilé COSA contre les violations des contraintes sémantique définit par le profile.
- Capacité aux architectes de construire facilement des modèles d'architecture UML 2.0 avec l'approche COSA.
- Labilité aux concepteurs de réutilisé et partagé les même concepts architecturaux durant tous le cycle de vie du système.

Transformation de COSA vers CORBA, COSA2CORBAPlug-In

L'outil COSA2CORBAPlug-In écrit en java et ATL (Atlas Transformation Language) a été implémenté dans la plate forme de développement logiciel Eclipse. Un outil de

transformation des architectures COSA vers CORBA qui offre une simplicité de transformation des modèles d'architectures UML 2.0 profilé COSA vers la plate-forme standard CORBA. Plus précisément, l'automatisation de la production des modèles UML des applications CORBA et la génération des interfaces CORBA, à partir des modèles COSA UML dans un contexte d'ingénierie des modèles. On applique deux transformations successives :

- COSA2CORBA qui permet la transformation de l'architecture COSA modélisée avec UML 2.0 vers l'architecture CORBA modélisée avec UML 1.4 et CORBA2ID qui automatise la génération des interfaces portable depuis le modèle CORBA UML.

Transformation de COSA vers EJB 2.0, COSA2EJBPlug-In

Un outil écrit en java et ATL (Atlas Transformation Language) a été implémenté dans la plate forme de développement logiciel Eclipse permet une transformation des modèles d'architectures UML 2.0 profilé COSA vers la plateforme EJB 2.0.

COSABuilder et COSAInstantiator

L'outil COSABuilder vise à permettre aux architectes de décrire graphiquement leurs architectures et de valider automatiquement ses sémantiques selon l'approche COSA. L'aspect clé de l'outil COSAInstantiator est la possibilité de générer automatiquement un éditeur graphique pour un modèle d'architecture COSA et de créer des architectures d'applications correctes.

Glossaire

ACME	Langage d'échange d'architectures.
ADL	*Architecture Description Language* : langage de description d'architecture. Les ADLs permettent de décrire les architectures logicielles indépendamment de leurs implémentations. Les ADLs décrit un système à un niveau haut d'abstraction en termes des composants et des connecteurs et des liens entre eux.
Architecture abstraite	Correspond à un ensemble des concepts de description architectural d'un système logiciel. COSA est un exemple des architectures abstraites.
Architecture concrète	Correspond à un ensemble des concepts des plates-formes objets et définie l'architecture dans le code source. UML est un exemple des architectures concrètes. Le raffinement architectural est le passage d'une architecture abstraite vers une architecture plus concrète.
CBSD	*Component-Based Software Development :* approche de développement de logiciel basé sur l'assemblage des composants préfabriqués, permet aux développeurs de faire abstraction des détails d'implémentation et de faciliter la manipulation et la réutilisation des composants.
CCM	*CORBA Component Model* : modèle de composants logiciels pour les applications industriels de type 3-tiers développé par l'OMG dans le cadre de la norme CORBA.
CIM	Modèle d'exigences.
Composant	Entité modulaire d'une application, packagé et documentée de façon à être réutilisable. Un composant peut avoir une ou plusieurs interfaces offertes/requises.
Configuration	Représente un graphe de composants et de connecteurs et définit la façon dont ils reliés entre eux.
Connecteur	Entité architectural qui encapsule les mécanismes d'interactions entre les composants et les règles qui régissent ces interactions. Les connecteurs peuvent être implicites, des ensembles énumérés ou dont la sémantique est définie par l'utilisateur. Un connecteur est caractérisé par des interfaces, des contraintes et des propriétés.
COSA	Component-Object based Software Architecture : approche hybride composant-objet pour la description d'architecture logicielle.

CORBA *Common Object Request Broker Architecture :* est une plate-forme d'exécution standard et international qui décrit des architectures et des spécifications pour le traitement d'objets répartis, portable.

COTS *Components Off-The-Shelf :* composants sur étagère qui sont des briques élémentaires standard que l'on peut acquérir. Ils peuvent être développés par l'entreprise qui les utilise ou avoir été développés par une autre.

EJB *Entreprise JavaBeans* : modèle de composants logiciels pour les applications industrielles de type 3-tiers développé par SUN dans le cadre de l'environnement Java édition entreprise (J2EE).

IDL *Interface Definition Language* : langage de définition des interfaces d'objets et de composants logiciels, indépendant des codes spécifique d'implémentation.

Interface définit des services fournis et requis par un type de composant, qui lui permettent d'interagir avec son environnement, y compris avec d'autres types de composants.

Métamodèle Définition précise de tous les concepts de modélisation. Métamodèle UML associe à chacun de ses concepts des entités syntaxiques.

MDA *Model Driven Approach:* démarche de développement proposée par l'OMG. Elle se base sur la transformation automatique des modèles vers des applications complètement déployable et exécutable. MDA supporte le développement de systèmes informatiques en assurant l'interopérabilité et la portabilité et définit plusieurs niveaux des modèles : CIM, PIM, PSM, code.

.NET Plateforme logicielle de Microsoft pour le développement d'applications architecturé autour d'une machine virtuelle.

OMG *Object Management Group:* consortium oeuvrant pour la standardisation des modèles de conception et d'implantation orientés objets.

PIM Modèle d'application indépendant des plates-formes d'exécution.

Profil UML Spécialisation d'UML à un domaine particuler. Un profil est composé de stéréotypes, de valeurs marquées et de contraintes. Le profil CORBA est un support graphique des composants CORBA en UML.

PSM Modèle d'application spécifique à une plate-forme d'exécution.

Transformation Automatisation des productions des modèles. Une transformation entre des modèles d'exigences (CIM), des modèles indépendants des plates-formes (PIM) et des modèles spécifiques (PSM).

Wright Langage d'échange d'architectures réalisé à l'université de Carnegie Mellon.

Annexes

Annexe A : Code ATL de la transformation COSA - eCore

```
-- ================================================================================
-- transformation rules begin
-- ================================================================================
-- This rule generates a eCoreClass from COSAConfiguration
rule MainConfiguration {
   from IN: COSA!Configuration(IN.parent.oclIsUndefined())
   to OUTpkg: MOF!EPackage(
      name <- IN.name,
      nsURI <- IN.getMetaData('nsURI'),
      nsPrefix <- IN.getMetaData('nsPrefix'),
   ),
   OUTdetails : MOF!EStringToStringMapEntry (
      key <- 'className'
      value<-'MainConfiguration'
   ),
   OUTannotation : MOF!EAnnotation (
      source <- 'meta',
      details<- OUTdetails
   ),
   OUTclass : MOF!EClass(
      name <- IN.name,
      eAnnotations<- Sequence {OUTannotation,IN.constraints},
      eStructuralFeatures <- IN.properties,
      eClassifiers <- IN.elements
   )
   do {
       thisModule.Result_ecorelModel <- OUTpkg;
       OUTpkg.eClassifiers<- OUTclass;
       }
}
-- This rule generates a eCoreClass from COSAConnector
rule  COSAUserConnector2MOFClass {
   from IN : COSA!UserConnector
   to  OUTclass : MOF!EClass(
       name <- IN.parent.name+IN.name,
       eAnnotations<- Sequence {OUTannotation,IN.constraints},
       eStructuralFeatures <- IN.properties
      )       ,
      OUTref : MOF!EReference(
       name <- IN.name,
       eType<-IN,
       lowerBound<-0,
       upperBound<- -1 ,
       containment<-true
      ),
      OUTdetails : MOF!EStringToStringMapEntry (
       key <- 'className'
       value<-'Connector'
      ),
      OUTannotation : MOF!EAnnotation (
       source <- 'meta',
```

```
       details<- OUTdetails
      )
      do{
       Result_ecorelModel.eClassifiers<- OUTclass;
       if (IN.parent.name =Result_class.name)
         Result_class.eStructuralFeatures<-OUTref;
        }
}
-- This rule generates a eCoreClass from COSABuiltInConnector
rule  COSABuiltInConnector2MOFClass COSAUserConnector2MOFClass {
   from IN : COSA!BuiltInConnector
   to  OUTclass : MOF!EClass(
    name <- IN.source.name+'_to_'+IN.target.name,
    eAnnotations<-OUTannotation
   ),
   OUTref: MOF!EReference(
    name <- IN.name,
    eType<-IN,
    LowerBound<-0,
    upperBound<- -1,
    containment<-true
   ),
   OUTdetails: MOF!EStringToStringMapEntry (
    key <- 'className',
    value <-if (IN.oclIsKindOf(Binding))then
               'Binding'
            else
             if (IN.oclIsKindOf(Attachment)) then
                 'Attachment'
               else
                 'Use'
               endif
             endif
            endif
   ),
   OUTannotation  : MOF!EAnnotation (
       source <- 'meta',
       details<- OUTdetails
   )
   do{
    let ch_par : String = ''
    if (not IN.parent.parent.oclIsUndefined())
      ch_par=IN.parent.parent.name+IN.parent.name
    else
      ch_par=IN.parent.name;
    Result_ecorelModel.eClassifiers<-OUTclass;
    Result_parent.eStructuralFeatures<-OUTref;
     }
}
-- ================================================================================
-- transformation rules end
-- ================================================================================
```

Annexe B : Un Extrait du Code ATL des transformations COSA - CORBA et CORBA – IDL

B.1. La transformation COSA vers CORBA

```
/*-----------------   définit ici les differentes helper's ---------------------*/

================================================================================
-- model-specific helpers begin
-- ============================================================================


--Global varialbe to store the UML model (root)
helper def: Result_UmlModel: UML14!Model = '';

--Global varialbe to store the CORBA UML Module
helper def: CORBAModule: UML14!Package = '';


-- This helper returns the applied UML2!Stereotype
helper context UML2!NamedElement def:hasStereotype(name:String): Boolean =
  self.getappliedStereotypes-> select (e| e.name =name)->notEmpty();

-- This helper returns the UML2!Property of UML2!NamedElement
helper context UML2!NamedElement def: getCOSAProps():Sequence(UML2!Property) =
  self.ownedAttribute->select(e|e.hasStereotype('COSANonFuncProp'));

-- This helper returns the UML2!Constraint of UML2!NamedElement
helper context UML2!NamedElement def: getCOSAConstraints():
Sequence(UML2!Constraint)=self.ownedRule->select(e|e.hasStereotype('COSAConstraint'

-- This helper returns the UML2!Dependency of UML2!NamedElement
helper context UML2!NamedElement def: getCOSAImplementations():
  Sequence(UML2!Dependency) =self.clientDependency->collect(cd|cd.supplier);



-- This helper returns the UML2!Dependency of UML2!Usage
helper context UML2!Usage def: SupplierIsCOSAPort_COSARole(): Boolean =
       if (self.supplier->forAll(s|s.hasStereotype('COSARequired-Port') or
                                  s.hasStereotype('COSARequired-Role'))) then
              true
       else
              false
       endif;

-- ============================================================================
-- model-specific helpers end
-- ============================================================================

-- ============================================================================
-- transformation rules begin
-- ============================================================================

-- This rule generates a CORBAModule from a COSAConfiguration
rule COSAConfiguration2CORBAHome {
   from inConfig : UML2!Component(inConfig.hasStereotype('COSAConfiguration'))
   to outHome:UML14!Package (
              name <- inConfig.name,
              namespace<-thisModule.Result_UmlModel,
              stereotype <-ApplyStereotype ('CORBAModule'),
```

```
    )
    do {thisModule.CORBAModule<-inConfig;}
}

-- This rule generates a CORBAHome from a ComponentCOSA
rule COSAComponent2CORBAHome {
    from inComp: UML2!Component(inComp.hasStereotype('ComponentCOSA'))
    to outHome:UML14!Class (
              name <- inComp.name,
              namespace<-inComp.owner,
              feature<-inComp.getCOSAProps(),
              constraint <-inComp.getCOSAConstraints(),
              clientDependency <-inComp.clientDependency,
              stereotype <- ApplyStereotype ('CORBAHome')
    )
}

-- This rule generates a CORBAComponent and CORBAManages from a ComponentInterface
rule ComponentInterface2CORBAComponent{
    from inItf : UML2!Port(inItf.hasStereotype('ComponentInterface'))
    to outComp:UML14!Class (
              name<-inItf.name,
              namespace<-thisModule.CORBAModule,
              stereotype <- ApplyStereotype ('CORBAComponent')
        ),

    outManages:UML14!Dependency (
              name<-'To_'+inItf.name,
              namespace<-thisModule.CORBAModule,
              supplier<-inItf,
              client<-inItf.owner,
              stereotype <- ApplyStereotype (‘CORBAManages’)
        )
}

-- This rule generates a CORBAInterface from a COSAPort if Mode = synchronous
--                      CORBAEventPort from a COSAPort if Mode = asynchronous
rule COSAPort2CORBAInterface{
       from InPort:UML2!Interface (InPort.hasStereotype('COSAPort'))
       to     outIntf:UML14!Interface(
              name <- InPort.name,
              namespace<-thisModule.CORBAModule,
       )
       do{
              if (InPort.hasStereotype('COSARequired-Port'))
                 if (InPort.getPropertyObj('COSARequiredPort','Mode')=#synchronous)
                    outIntf.stereotype<- ApplyStereotype('CORBAInterface');
                 else
                     outIntf.stereotype<-ApplyStereotype ('CORBAEventPort');
              else
                 if (InPort.getPropertyObj('COSAProvidedPort','Mode')=#synchronous)
                    outIntf.stereotype<- ApplyStereotype('CORBAInterface');
                 else
                    outIntf.stereotype<-ApplyStereotype ('CORBAEventPort');

              thisModule.CORBAModule.ownedMember ->including(outIntf);
       }
}
-- This rule generates a CORBAHome from a COSAConnector
rule COSAConnector2CORBAHome {
    from inConn : UML2!Class(inConn.hasStereotype('COSAConnector'))
    to outHome:UML14!Class (
              name <- inConn.name,
              namespace<-inConn.owner,
              feature<-inConn.getCOSAProps(),
              constraint <-inConn.getCOSAConstraints(),
              clientDependency <-inConn.clientDependency,
              stereotype <- ApplyStereotype ('CORBAHome')
    )
```

```
}

-- This rule generates a CORBAComponent and CORBAManages from a COSAConnector
rule ConnectorInterface2CORBAComponent{
   from inItf : UML2!Port(inItf.hasStereotype('ConnectorInterface'))
   to outComp:UML14!Class (
            name<-inItf.name,
            namespace<-thisModule.CORBAModule,
            stereotype <- ApplyStereotype ('CORBAComponent')
      ),

   outManages:UML14!Dependency (
            name<-'To_'+inItf.name,
            namespace<-thisModule.CORBAModule,
            supplier<-inItf,
            client<-inItf.owner,
            stereotype <- ApplyStereotype ('CORBAManages')
      )
}

-- This rule generates a Dependency between CORBAComponent and CORBAHome
-- CORBAManages if CORBAInterface or CORBAConsumes if CORBAEventPort of Attachment
rule COSAAttachment2CORBAComponent{
      from InAss:UML2!Association (InAss.hasStereotype('Attachment'))
      using {
         connectionr: UML2!Interface=InAss.getAtt('ProvidedPort','ProvidedRole');
         connectiond: UML2!Interface=InAss.getAtt('RequiredPort','RequiredRole');
      }
      to     outComponent:UML14!Class(
            name <- InAss.name,
            namespace<-thisModule.CORBAModule,
            stereotype <- ApplyStereotype ('CORBAComponent'),
      ),

      outHome:UML14!Class (
            name<-InAss.name+'Home',
            namespace<-thisModule.CORBAModule,
            stereotype <- ApplyStereotype ('CORBAHome'),
      ),

      outManages:UML14!Dependency (
            name<-'To_'+InAss.name,
            namespace<-thisModule.CORBAModule,
            supplier<-outComponent,
            client<-outHome,
            stereotype <-ApplyStereotype ('CORBAManages')
      ),

      outUsesConsumes:UML14!Association (
            name<-'To_'+InAss.name,
            namespace<-thisModule.CORBAModule,
            stereotype <-if thisModule.Result_UmlModel.hasCORBAInterface() then
                              thisModule.ApplyStereotype('CORBAManages')

                        else
                              thisModule. ApplyStereotype('CORBAConsumes')
                        endif
      ),
    outProvidesPublishe:UML14!Association (
            name<-'For_'+InAss.name,
            namespace<-thisModule.CORBAModule,
            stereotype<- if thisModule.Result_UmlModel.hasCORBAInterface() then
                              thisModule.ApplyStereotype('CORBAProvides')
                        else
                              thisModule.ApplyStereotype('CORBAPublishs')
                        endif
      )
}
```

```
-- Generate IDL-Attribute from COSANonFuncProp
rule COSAProperty2CORBAAttribute{
       from prop:UML2!Property (prop.hasStereotype('COSANonFuncProp'))
       to     attr:UML14!Attribute(
              name <- prop.name,
              type <- prop.getType(),
              owner<- prop.owner,
              initialValue <-prop.defaultValue,
              ownerScope<-#sk_classifier,
              stereotype <- ApplyStereotype('IDL-Attribute')
       )
}

-- ============================================================================
-- transformation rules end
-- ============================================================================
```

B.2. La transformation CORBA vers IDL

```
-- file CORBAToIDL.atl

query CORBAtoIDL =
       -- generation form the first instance of CORBA Module
       UML!Package.allInstances()->collect(e |
              if not e.IsCORBAModule() then true
              else e.generateCORBAModule().writeTo('c:/'+e.name+'.IDL')

              endif);

--/*---------------   définit ici les differentes helper's -------------------*/
-- ============================================================================
-- Stereotypes helpers begin
--
-- ============================================================================
-- This helper returns all UML classes stereotyped CORBAModule
helper context UML!ModelElement def: IsCORBAModule() : Boolean =
        self.oclIsKindOf(UML!Package) and
           self.stereotype->select (e| e.name ='CORBAModule')->notEmpty();

-- This helper returns all UML classes stereotyped CORBAHome
helper context UML!ModelElement def: IsCORBAHome() : Boolean =
        self.oclIsKindOf(UML!Class) and
           self.stereotype->select (e| e.name ='CORBAHome')->notEmpty();

-- This helper returns all UML classes stereotyped CORBAComponent
helper context UML!ModelElement def: IsCORBAComponent() : Boolean =
       self.oclIsKindOf(UML!Class) and
           self.stereotype->select (e| e.name ='CORBAComponent')->notEmpty();

-- This helper returns all UML dependencies stereotyped CORBAManages
helper context UML!ModelElement def: IsIDLAttribute() : Boolean =
        self.oclIsKindOf(UML!Attribute) and
           self.stereotype->select (e| e.name ='IDL-Attribute')->notEmpty();
-- This helper returns all UML operations stereotyped IDL-Operation
helper context UML!ModelElement def: IsIDLOperation() : Boolean =
        self.oclIsKindOf(UML!Operation) and
           self.stereotype->select (e| e.name ='IDL-Operation')->notEmpty();

-- This helper returns all UML interfaces stereotyped CORBAInterface
helper context UML!ModelElement def: IsCORBAInterface() : Boolean =
        self.oclIsKindOf(UML!Interface) and
           self.stereotype->select(e| e.name ='CORBAInterface')->notEmpty();

-- ============================================================================
-- Stereotypes helpers end
```

```
-- ================================================================================

/*------ définit differentes helper's de génération du code IDL  ------------*/

-- ================================================================================
-- code generation helpers begin
-- ================================================================================

-- Code generation of CORBA Module and its CORBA Interfaces
helper context UML!Package def: generateCORBAModule():String =
       '//File: '+self.name+'.idl\n'
       +'module '+self.name +' {\n'
       +'//   défintion des interfaces \n'
       +self.ownedElement->select(intf|intf.IsCORBAInterface())
          ->iterate(intf;accMG:String=''|accMG+intf.generateCORBAInterface()+'\n')
             +self.ownedElement->select(mg|mg.IsCORBAManages())
       ->iterate(mg;accMG:String=''|accMG+mg.generateCORBAManages()+'\n')+'};';

-- Code generation of CORBA Interface and its IDL-operations
helper context UML!Interface def: generateCORBAInterface() : String =
       ('interface ' + self.name +' {\n' +
       self.ownedElement->select (f | f.IsIDLOperation())
         ->iterate (e; acc: String = '' | acc + e.generateIDLOperation())
            +'}; \n').writeTo ('c:/'+self.name+'.IDL')
-- Code generation of CORBA manages
helper context UML!Dependency def: generateCORBAManages():String =
    self.supplier->select(cp|cp.IsCORBAComponent())
             ->iterate(cp;accCP:String=''|accCP+cp.generateCORBAComponent())
       +'  home '+self.client->select(h|h.IsCORBAHome())
             ->iterate(h;accH:String=''|accH+h.name.toString()+' ')
       +'manages '+self.supplier->select(cp|cp.IsCORBAComponent())
             ->iterate(cp;accCP:String=''|accCP+cp.name.toString()+' ')+' {\n'
       +self.client->select(h|h.IsCORBAHome())
             ->iterate(h;accH:String=''|accH+h.generateCORBAHome()+'  };\n\n');

-- Code generation of CORBA components
helper context UML!Class def: generateCORBAComponent():String =
       '//  description du composant '+self.name+'\n'+
       '  component '+self.name +' {\n  };\n';

-- Code generation of IDL-Attribute
helper context UML!Feature def: generateIDLAttribute():String =
       '      attribute '+self.name.toString() +';\n';

-- Code generation of IDL-Attribute
helper context UML!Class def : generateCORBAHome():String =
     self.feature->select(f|f.IsIDLAttribute())
             ->iterate(f;accF:String=''|accF+f.generateIDLAttribute());

-- Code generation of CORBA operations
helper context UML!Operation def : generateIDLOperation() : String =
    '  void '+self.name.toString()+ '()\n';

-- ================================================================================
-- code generation helpers end
--
================================================================================
```

Annexe C : Evaluation des outils COSA et Etudes de Cas

C.1. Evaluation des outils de modélisation et d'instanciation

Dear Developer,

This is a questionnaire to get your opinion concerning the evaluation of our COSA Tools compared to ACMEStudio. This questionnaire is part of an academic study for research purposes, and doesn't intended for any commercial purposes. The purpose is to determine to what extend our tools are useful for software architecture. Your opinion is very important in this study, so please try to answer the questions truly. We thank you for your help, and appreciate your cooperation in supporting the scientific research.

No.	*Question*	*Oui*	*Peut être*	*Non*
1	*Est-ce que l'outil à une bonne interface graphique?*			
2	*Est-ce que l'outil est utile pour acquérir des compétences professionnelles ?*			
3	*Est-ce que l'outil est utile d'apprendre des choses ?*			
4	*L'outil prend-il de la réutilisation ?*			
5	*Est-ce que l'outil facilite la description des architectures logicielles complexes ?*			
6	*Est-ce que l'outil dispose d'une bonne interopérabilité ?*			
7	*La période de temps allouée ne suffit pas pour compléter la conception?*			
8	*Est-ce qu'il est facile de vérifier le modèle d'architecture à tout moment ?*			
9	*L'outil est 'il facile à utiliser ?*			
10	*Est-ce que l'outil est flexible ?*			
11	*Est-ce que on peut appliquer une méthode de vérification formelle lors de la conception ?*			
12	*Est-ce que tous les concepts des architectures sont disponibles dans l'outil ?*			
13	*Le modèle d'architecture peut être instancié à tout moment ?*			
14	*Est-ce que l'outil m'a donné la chance de travailler sur un projet réel ?*			
15	*Est-ce que l'outil m'a donné une chance de faire quelque chose d'utile pour ma communauté*			
16	*Est-ce que l'outil ma aidé à définir des contraintes sur les éléments d'architecture ?*			
17	*Avez-vous d'autres recommandations qui pourraient être utiles pour améliorer l'outil ?*			

Résumé des réponses autour COSABuilder/COSAInstantiator

No.	*Oui*	*Peut être*	*Non*
1	*9*	*1*	*-*
2	*10*	*-*	*-*
3	*9*	*-*	*1*
4	*9*	*-*	*1*
5	*10*	*-*	*-*
6	*10*	*-*	*-*
7	*7*	*1*	*2*
8	*10*	*-*	*-*
9	*10*	*-*	*-*
10	*9*	*-*	*1*
11	*10*	*-*	*-*
12	*9*	*1*	*-*
13	*10*	*-*	*-*
14	*7*	*2*	*1*
15	*9*	*2*	*1*
16	*10*	*-*	*-*
17	*- Les outils COSABuilder manque de la génération automatique des parties importantes d'implémentation du système et manque de maturité.* *- L'instanciation automatique est le point fort des outils de COSA.*		

Résumé des réponses autour ACMEStudio

No.	*Oui*	*Peut être*	*Non*
1	*7*	*1*	*2*
2	*10*	*-*	*-*
3	*7*	*-*	*3*
4	*9*	*1*	*-*
5	*7*	*-*	*3*
6	*6*	*3*	*1*
7	*6*	*1*	*3*
8	*10*	*-*	*-*
9	*9*	*-*	*1*
10	*7*	*2*	*1*
11	*10*	*-*	*-*
12	*8*	*2*	*-*
13	*5*	*1*	*4*
14	*6*	*3*	*1*
15	*7*	*1*	*2*
16	*10*	*-*	*-*
17	*- ACMEStudio manque de la documentation et de la maturité.* *- L'adaptation et la réutilisation sont les principaux avantages d'ACMEStudio*		

C.2. Etudes de cas

C.2.1. Etudes de cas 1 : Système de contrôle d'accès au parking

Nous avons essayé dans la première étude de cas de décrire un système de contrôle d'accès au parking en utilisant les outils COSABuilder/COSAInstantiator. Ce système contrôle la réservation d'un ensemble des places disponibles dans le parking. Un seul utilisateur ayant une identité connus et des droits d'accès corrects seront autorisés à entrer dans le parking. Un contrôleur contrôle l'accès au parking. En plus, un contrôleur ayant une console ce qui permet à

un superviseur d'interagir avec le système (via le port *ExternalAccess*). Ce superviseur permet à d'autres voitures de réserver dans le parking. Le contrôleur affiche le nombre de places disponibles via l'écran LED.

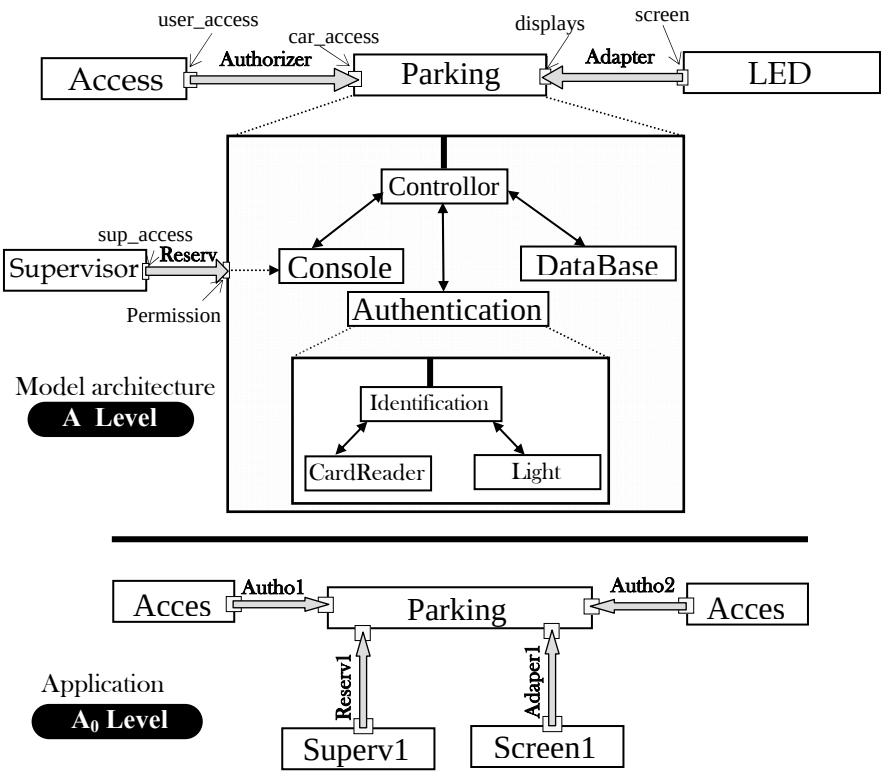


Figure C.1. ***L'architecture et l'application du système de contrôle d'accès au Parking.***

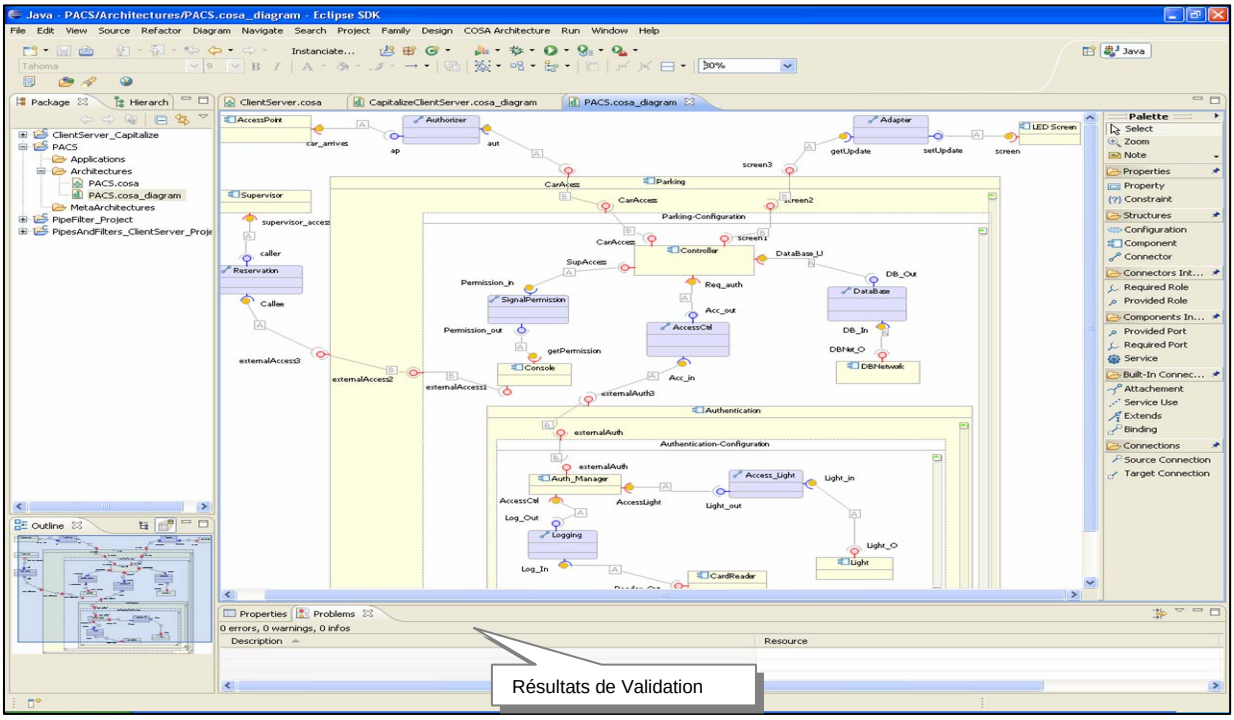


Figure C.2. ***L'architecture du système SCAP sous COSABuilder.***

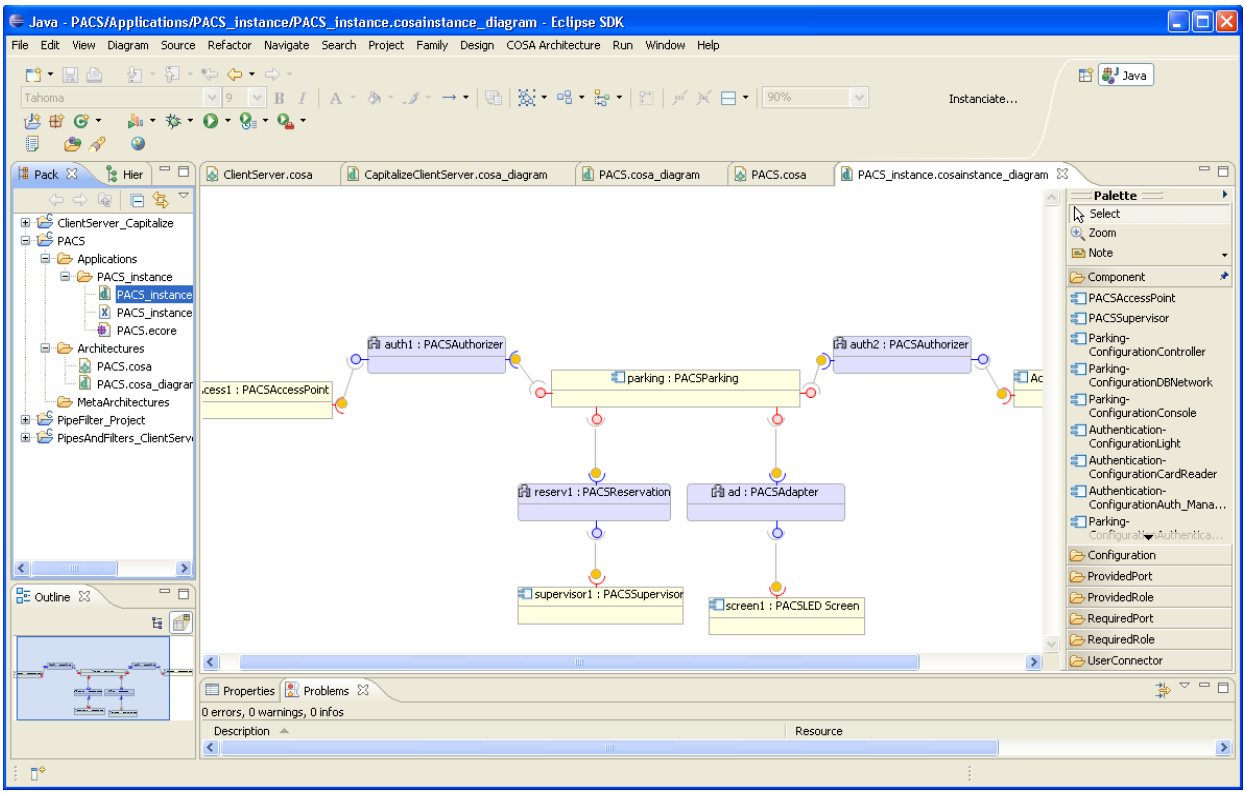


Figure C.3. ***L'application SCAP obenue par le générateur des instances COSA eCore.***

C.2.2. Étude de cas 2: Protocole de sécurisation des transactions électroniques

Secure Electronic Transaction (SET) est un protocole de sécurisation des transactions électroniques est mis au point par Visa et MasterCard, et s'appuyant sur le standard SSL.

Le titulaire de la carte bancaire *Visa ou MasterCard* (client), utilise son PC pour acheter des articles sous l'Internet. Le protocole assure une transaction électronique sécurisée pour l'acheteur. Les coordonnées bancaires des clients sont transmises en toute sécurité, les données sont cryptées par le protocole de sécurité SSL (Sécurité Socket Layer). SET est basé sur l'utilisation d'une signature électronique au niveau de l'acheteur et une transaction mettant en jeu non seulement l'acheteur et le vendeur, mais aussi leurs banques respectives. Un composant « *Client* » a besoin de sept ports de communications, trois ports avec le composant « *Autorisation de certificat* » (Certificate Authority) et les autres avec le composant « *Acheteur* » (Marchant).

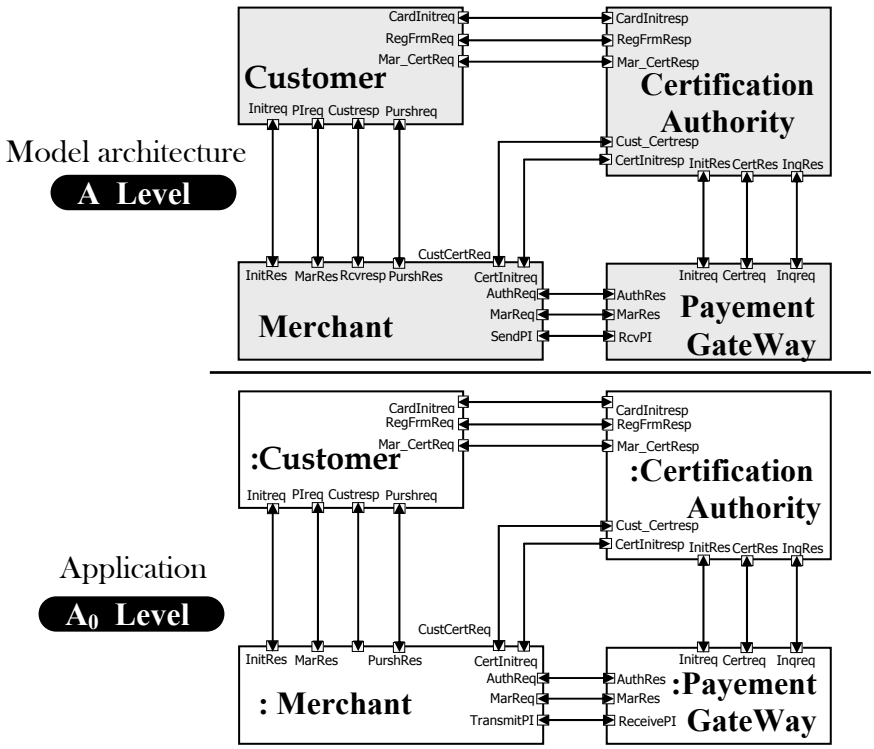


Figure C.4. *L'architecture et l'application du système SET*

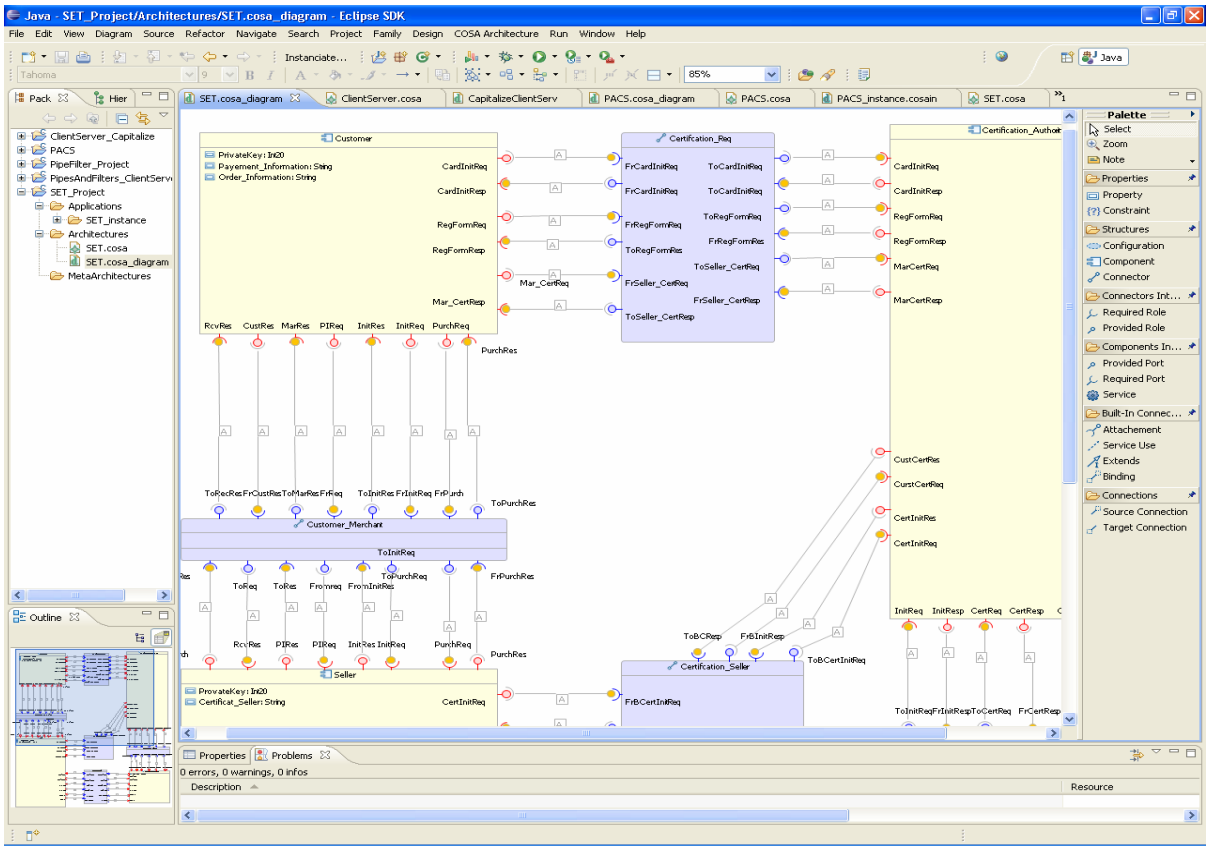


Figure C.5. *La description de l'architecture SET sous COSABuilder.*