Evolution Structurelle dans les Architectures Logicielles à base de Composants

Nassima Sadou

Université de Nantes **École des Mines de Nantes**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2008

N° attribué par la bibliothèque

Evolution Structurelle dans les Architectures Logicielles à base de Composants

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Nassima SADOU-HARIRECHE

*Le 18 décembre 2007 à l'*UFR *Sciences & Techniques, Université de Nantes, devant le jury ci-dessous*

Président	:	Henri BRIAND, Professeur	Université de Nantes
Rapporteurs	:	Tom MENS, Professeur	SEL, Université de Mons-Hainaut
		Dominique RIEU, Professeur	IMAG, Grenoble
Examinateurs	:	Salah SADOU, HDR	VALORIA, Université de Bretagne Sud

Directeur de thèse : Mourad OUSSALAH

Encadrant de thèse : Dalila TAMZALIT

Laboratoire: **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 0366-XXX

EVOLUTION STRUCTURELLE DANS LES ARCHITECTURES LOGICIELLES À BASE DE COMPOSANTS

Nassima SADOU-HARIRECHE

favet neptunus eunti

Université de Nantes

Nassima SADOU-HARIRECHE

Evolution Structurelle dans les Architectures Logicielles à base de Composants

V+VIII+168 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version de l'association de jeunes chercheurs en informatique , Université de Nantes. La classe `these-LINA` est disponible à l'adresse : `http://login.irin.sciences.univ-nantes.fr/`.
Cette classe est conforme aux recommandations du ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche (circulaire n° 05-094 du 29 mars 2005), de l'Université de Nantes, de l'école doctorale « Sciences et Technologies de l'Information et des Matériaux » (ED-STIM), et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

- AFNOR NF Z41-006 (octobre 1983)
  Présentation des thèses et documents assimilés ;
- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Impression : template-Nassima-9.10.tex – 20/03/2008 – 15:59.

Révision pour la classe :

Résumé

Le travail présenté dans cette thèse s'inscrit dans le cadre des architectures logicielles à base de composants. Une architecture logicielle offre une description d'un système à un niveau d'abstraction élevé en terme de composants et d'interactions entre ces composants. La problématique abordée relève de l'évolution structurelle dans les architectures logicielles à base de composants. L'évolution est une nécessité importante dans le monde du logiciel et des systèmes informatiques. Elle permet, dans le cadre des architectures logicielles, d'éviter que celles-ci ne restent figées et soient obsolètes par rapport aux besoins en perpétuels changements. Un autre objectif essentiel est la possibilité de pouvoir élargir les architectures logicielles et d'appliquer le passage à l'échelle, pour prendre en compte de nouveaux besoins ou des fonctionnalités plus complexes. Une architecture doit donc pouvoir être modifiée pour rester utilisable, réutilisable et disponible pour ses utilisateurs, et cela tout au long du cycle de vie du système. Notre contribution à cette problématique se scinde en trois axes :

Le premier axe consiste en la proposition d'un modèle d'évolution dénommé SAEV (Software Architecture EVolution model) permettant l'abstraction, la spécification et la gestion de l'évolution des architectures logicielles. SAEV se veut un modèle générique, uniforme et indépendant de tout langage de description d'architectures logicielles.

Le deuxième axe s'appuie sur deux constats : le premier constat est que les architectures logicielles ne véhiculent pas assez d'informations sur le degré de corrélation entre leurs éléments constitutifs, qui permettraient de déterminer et de propager automatiquement les impacts d'une évolution. Le deuxième constat est que les connecteurs, de par leur position d'intermédiaires entre les éléments architecturaux sont des supports idéaux pour véhiculer les changements entre ces éléments. C'est ainsi que nous proposons d'enrichir les connecteurs par des propriétés sémantiques qui exprimeraient alors la corrélation entre les éléments d'une architecture logicielle qu'ils relient.

Le troisième axe illustre la prise en compte des propriétés sémantiques proposées dans le cadre de l'ADL COSA (Component-Object based Software Architecture). COSA est un ADL hybride qui réifie les concepts communément admis par la majorité des langages de description d'architectures logicielles. Cet axe montre aussi l'application du modèle SAEV sur des architectures logicielles décrites en COSA et en tenant compte des propriétés sémantiques définies.

Mots-clés : Architecture logicielle, ADL, spécification de l'évolution, stratégie d'évolution, règle d'évolution, ECA, propagation de l'évolution, propriétés sémantiques des connecteurs.

Abstract

The work presented in this thesis concerns with the study of component-based software architecture. A software architecture provides a description of a software system at a high level of abstraction in term of components and interactions among these components. The studied problem related to the structural evolution in component-based software architecture. The evolution is a significant issue in the domain of software and information processing systems. It allows architectures to evolve in order to cope with changing requirements. Another essential objective is the possibility of improving the scalability of architectures in order to take into account the new needs with complex functionalities. Therefore, an architecture must accept modifications to remain usable, reusable and available for its users, and this throughout the whole life cycle of the system. Our contribution to these issues is divided into three axes:

The first axis consists of proposing an evolution model called SAEV (Software Architecture EVolution model) allowing the abstraction, the specification and the management of software architecture evolution. SAEV intended to be a generic model, uniform and independent of any architecture description languages.

The second axis is based on two issues: the first issue is that software architecture does not provide enough information concerning the degree of correlation among its components; this correlation would make it possible to determine and to propagate automatically the impacts of an evolution within an architecture. The second issue is that connectors, from their position as intermediaries between architectural elements, are ideal supports to propagate changes within these elements. Thus we propose to enhance connectors with semantic properties that would express the correlation between elements of a software architecture that they connect.

The third axis illustrates the use of semantic properties that are suggested within the framework of the ADL COSA (Component-Object based Software Structures). COSA is a hybrid ADL that supports the commonly admitted concepts by majority architecture description languages. This axis highlights also the application of the model SAEV for software architectures described in COSA taking into account the defined semantic properties.

Keywords: Software architecture, ADL, evolution specification, evolution strategy, evolution rules, ECA, evolution propagation , semantic properties of connectors.

Remerciements

Il me serait impossible de citer nommément toutes les personnes qui m'ont aidé, encouragé et soutenu afin que ce travail puisse voir le jour. Que toutes ces personnes trouvent ici l'expression de ma sincère reconnaissance.

Je tiens tout d'abord à remercier vivement Monsieur Henri BRIAND, professeur à l'école polytechniques de l'université de Nantes, pour m'avoir fait l'honneur d'accepter d'examiner mes travaux et de présider le jury de ma soutenance de thèse. Je tiens aussi à remercier particulièrement Madame Dominique RIEU, professeur à l'université de Grenoble et Monsieur Tom MENS, professeur à l'université de Mons-Hainaut, pour avoir accepté d'être rapporteurs de cette thèse. Je les remercie d'avoir apporter autant de soins pour la lecture de cette thèse ainsi que pour les critiques et suggestions constructives, qu'ils ont apporté à ce travail. Je remercie également Monsieur Salah SADOU, Maître de conférences à l'université de Bretagne sud, pour avoir accepter d'examiner cette thèse.

J'adresse ma grande reconnaissance à Mourad OUSSALAH responsable de l'équipe MODAL du LINA, pour m'avoir accepter au sein de son équipe et d'avoir diriger ce travail de thèse. Tes conseils et tes orientations durant toute cette période de thèse, m'ont toujours redonné confiance, volonté et une grande passion pour la recherche. Mes remerciements vont également à Dalila TAMZALIT, Maître de conférences à l'université de Nantes, pour avoir accepté d'encadrer cette thèse. Merci pour toutes les discussions constructives qu'on a partagé et pour ta patience surtout dans les moments de doutes, où je suis difficile à convaincre.

Je remercie très chaleureusement Salima HAMMA pour son soutien, ses conseils lumineux et surtout pour tout le temps qu'elle m'a toujours consacré malgré ses grandes préoccupations. Un grand merci aussi à tous les membres de l'équipe MODAL avec qui j'ai partagé les grands moments de cette thèse notamment Adel SMEDA et Olivier LE GOAER.

Ma grande gratitude est envers ma famille ici en France et en Algérie. Une grande reconnaissance à mon mari qui m'a soutenu tout au long de ces années de thèse et qui a partagé avec moi les grands moments de doutes où je ne pensais pas aller au bout de ce travail. Merci pour tout le temps que tu as consacré à cette thèse et surtout pour tous tes sacrifices. Mes remerciements les plus profonds pour mon fils LYES qui a su me soutenir avec son grand sourire. Ta naissance au cours de cette thèse ma redonner beaucoup de volonté et ma pousser à aller devant. Désolée pour tout le temps dont je t'ai privé au profit de cette thèse. Ma reconnaissance va aussi envers tous les membres de ma famille et belle-famille pour leur confiance leur encouragement et leur amour. Que chacun d'entre eux trouve ici ma grande gratitude. Je n'oublierai pas de remercier mes amis d'ici et de TIZI-OUZOU avec lesquels j'ai partagé de beaux moments de bonheur. Je remercie tout particulièrement Bourassia pour tout ce qu'on a vécu ensemble.

Enfin, ma grande pensé et gratitude est envers toutes les personnes chères qui ne sont plus là mais qui m'ont jamais quitté réellement, et dont la réussite de ce travail donnerait une énorme joie et fierté. A jamais et pour toujours à mon père.

Sommaire

— Corps du document —

— Pages annexées —

— Annexes —

Introduction

Cadre de la thèse

Cette thèse s'inscrit dans le domaine des architectures logicielles des systèmes à base de composants. Elle adresse plus spécifiquement la problématique de l'évolution dans ces architectures logicielles.

Les systèmes logiciels à base de composants sont des systèmes développés par assemblage de composants réutilisables, préfabriqués et pré-testés. L'intérêt accru ces dix dernières années pour le développement de logiciels à base de composants est motivé principalement par la réduction des coûts et des délais de développement de logiciels. En effet, on prend moins de temps à acheter, et donc à réutiliser, un composant que le concevoir, le coder, le tester, le debogguer et le documenter [65].

Les architectures logicielles ont apporté une réelle contribution dans le développement de tels systèmes logiciels. Leurs principales caractéristiques résident d'une part dans leur pouvoir de gérer les abstractions et les niveaux d'expressivité d'un système, et d'autre part dans leur capacité à prendre en compte la modélisation de la structure et du comportement d'un système. Les architectures logicielles modélisent un système en terme de composants représentant les fonctionnalités de ce système et des connecteurs décrivant les interactions entre ces composants. Actuellement plusieurs langages (dits ADL : Architecture Description Languages) sont proposés pour aider à la spécification, l'analyse et à la vérification des architectures logicielles des systèmes à base de composants.

Bien qu'issus au départ de communautés peu liées, les travaux relatifs aux systèmes à base de composants et aux architectures logicielles sont complémentaires et tendent à se rapprocher. Les premiers, issus du domaine industriel, visent à fournir les briques de base pour la production d'entités réutilisables. Les seconds, issus de la communauté académique, se sont concentrés sur la description de l'assemblage de ces briques et donc sur leurs interactions afin de construire une application de qualité. Il est bien admis qu'actuellement la maîtrise des systèmes logiciels complexes à base de composants est subordonnée à celle de leurs architectures. La bonne conception d'une architecture logicielle peut amener à un produit qui répond aux besoins des utilisateurs et qui peut être modifiée facilement pour ajouter une nouvelle fonctionnalité, alors qu'une architecture inappropriée peut avoir des conséquences désastreuses jusqu'à l'arrêt du projet [30].

Malgré les progrès constants dans le domaine des architectures logicielles des systèmes à base de composants, la problématique de l'évolution logicielle reste peu abordée. La majorité des efforts de recherche dans le domaine se focalise sur la spécification, le développement et le déploiement des architectures logicielles. Peu de travaux, à notre connaissance, se consacrent à l'analyse et à la conception de l'évolution de ces architectures logicielles.

Problématique de l'évolution

Un système logiciel n'est jamais figé, il est toujours amené à évoluer soit pour prendre en compte de nouveaux besoins fonctionnels, techniques ou matériels, soit pour modifier les besoins déjà exprimés,

et cela à n'importe quelle phase de son cycle de développement. L'évolution logicielle est ainsi une nécessité importante dans le monde du logiciel et des systèmes informatiques. Elle permet d'éviter que les systèmes ne restent figés et soient obsolètes par rapport aux besoins en perpétuels changements. En effet, l'évolution logicielle allonge la durée de vie des systèmes et ainsi garantit leur viabilité économique. Un autre objectif essentiel de l'évolution logicielle est la possibilité de pouvoir élargir les systèmes et d'appliquer le passage à l'échelle, pour prendre en compte de nouveaux besoins ou des fonctionnalités plus complexes. Un système doit donc pouvoir être modifié pour rester utilisable, disponible et robuste auprès de ses utilisateurs, et cela tout au long de son cycle de vie.

La problématique de l'évolution peut concerner un système en cours d'exploitation ou en phase de conception. Dans le premier cas, ces systèmes peuvent être revus et reformulés durant leur exploitation et leur maintenance. Leur stabilité et leur entière spécification passent par divers processus de validation [64]. Dans le second cas, ils peuvent bien être instables ou partiellement définis, nécessitant une spécification incrémentale.

Dans la littérature, certains auteurs utilisent le terme *maintenance*, d'autres parlent d'*adaptation* pour désigner l'évolution logicielle. Nous considérons, pour notre part, que le terme *évolution* est un terme plus général qui désigne l'ensemble des changements et des transformations d'un système tout au long de son cycle de vie. Nous considérons les deux autres activités de maintenance et d'adaptation comme étant des cas particuliers de l'évolution. En effet, la maintenance est définie assez souvent comme la totalité des activités requises pour fournir un support, à un coût rentable pour un système logiciel, réalisées de l'étape de pré-livraison à la post-livraison [67, 6]. L'adaptation, quant à elle, a pour objectif d'assurer le bon fonctionnement d'un système confronté à un changement de contexte.

La nécessité de l'évolution logicielle s'accentue encore plus dans le cadre des systèmes logiciels à base de composants où la conception et le développement se basent sur le principe de réutilisation. En effet, en plus des objectifs cités précédemment, dans un système à base de composants, un ou plusieurs de ses composants, une partie de ce système ou le système lui-même peuvent être amenés à être modifiés, adaptés pour pouvoir être réutilisés. La problématique de l'évolution d'un système à base de composants recouvre non seulement l'évolution du code source, mais également l'évolution de l'architecture du système et plus généralement de tout artefact intervenant dans le système logiciel.

Nous abordons l'évolution des systèmes à base de composants au niveau de leur architecture. Ainsi, de la même façon que l'architecture logicielle offre un niveau d'abstraction élevé permettant de raisonner sur les propriétés fonctionnelles et non fonctionnelles d'un système, ce niveau permettrait aussi au concepteur de raisonner sur les différentes évolutions d'un système en terme d'ajout et de suppression de composants, de connecteurs ou sur leur réorganisation sans pour autant se noyer dans le code source du système. De plus, assez souvent pour des raisons de confidentialité, nous pouvons avoir accès à l'architecture d'un système mais moins fréquemment à son code source. On parle alors d'architectures logicielles ouvertes [25].

L'évolution d'une architecture logicielle peut concerner sa structure ou son comportement. Elle peut être réalisée à l'étape de spécification ou de conception du système qu'elle décrit. On parle alors d'*évolution statique*. Elle peut également être réalisés à l'exécution de ce système. On parle alors d'*évolution dynamique*. Dans chacun de ces cas, il faut considérer que tout élément de l'architecture logicielle peut être amené à évoluer. Ainsi, il faut identifier ce qui peut évoluer, comment le faire évoluer, comment garantir la cohérence de l'architecture ayant évolué et ensuite comment répercuter ces évolutions de l'architecture logicielle sur le système qu'elle décrit.

Il est à noter qu'actuellement la majorité des efforts de recherche dans le domaine des Architectures Logicielles se focalisent sur la spécification, le développement et le déploiement des architectures. Peu de travaux, à notre connaissance, se consacrent à l'analyse et à la gestion de l'évolution de ces architectures. Ce manque est d'autant plus crucial durant les phases d'analyse et de conception des architectures logicielles. La plupart des travaux existants se limitent à des techniques offertes par l'ADL lui même, mais qui restent souvent influencées par le langage d'implémentation support et les techniques qu'il propose (tels que le sous typage et l'héritage [55, 52]). Nous sommes convaincus de l'importance d'intégrer l'analyse et la spécification de l'évolution pour toute architecture logicielle, et ce à n'importe quel stade de son cycle de développement et de son cycle de vie. En effet, de par ses spécificités, une architecture logicielle est appelée à être réutilisée, adaptée, maintenue et déployée. Son évolution fait partie intrinsèque de sa vie.

Notre objectif au cours de cette thèse est de contribuer à proposer une solution à ce manque. Nous nous intéressons spécifiquement à la problématique de l'évolution structurelle dans les architectures logicielles. Nous proposons un modèle d'évolution d'architectures logicielles baptisé SAEV (Software Architecture EVolution model) pour adresser cette problématique. Nous voulons SAEV comme un modèle d'évolution indépendant de tout ADL, pouvant s'appliquer à toute architecture logicielle, quel que soit son langage de description. L'apport principal de SAEV est d'offrir une abstraction de l'évolution permettant la spécification de l'évolution explicitement et indépendamment de la spécification de l'architecture elle-même et de son comportement.

Nous soulignons également au travers du modèle d'évolution SAEV, la nécessité de considérer la propagation des impacts de l'évolution d'un élément architectural aux autres éléments auxquels il est relié, tout en sauvegardant la cohérence globale de l'architecture. Cette étape est importante à automatiser autant que possible pour éviter d'introduire des incohérences dans l'architecture ayant évolué et des interactions trop fréquentes avec l'utilisateur. Nous proposons ainsi d'enrichir les descriptions d'architectures logicielles avec plus d'informations sur le degré de corrélation entre ses éléments. Nous avons capitalisé ces informations sous formes de propriétés sémantiques que nous associons au concept de connecteur. En effet, de par leur position d'intermédiaire entre les éléments architecturaux, nous considérons les connecteurs comme les supports idéaux pour véhiculer ces propriétés sémantiques. Le modèle SAEV peut ainsi se baser sur ces propriétés sémantiques pour déterminer et propager automatiquement (autant que possible) les impacts d'une évolution.

Afin d'étayer la prise en compte des propriétés sémantiques proposées, nous présentons une illustration au travers de l'ADL de COSA (Component-Object based Software Architecture) développé au sein de notre équipe. Nous nous baserons aussi sur cet ADL pour montrer une utilisation du modèle d'évolution SAEV.

Organisation du document

Outre cette introduction, le manuscrit est organisé en six chapitres, suivis d'une conclusion :

Chapitre 1 : Contexte du travail

Dans ce chapitre nous présentons les concepts liés au domaine d'architectures logicielles à base de composants qui représente le contexte global dans lequel s'inscrit notre travail. Nous présentons premièrement quelques définitions de ce qu'est une architecture logicielle, pour ensuite fixer une définition

sur laquelle nous nous appuierons tout au long de ce manuscrit. Nous présentons ensuite les concepts et mécanismes opérationnels proposés par les langages de description d'architectures logicielles à base de composants. Nous illustrons aussi les différents niveaux d'abstraction de description d'une architecture logicielle par un ADL, le niveau Méta, niveau Architectural, niveau Application. Nous terminons ce chapitre en présentant une classification des ADL en première et deuxième générations.

Chapitre 2 : Problématique de l'évolution dans les architectures logicielles : Etat de l'art

Nous abordons dans ce chapitre la problématique spécifique que nous adressons dans cette thèse. Notre premier intérêt se porte sur la description de l'évolution de l'architecture logicielle par rapport à ses propres caractéristiques, que nous dénommons **dimensions de l'évolution architecturale** : l'*objet de l'évolution*, le *type de l'évolution* et le *support de l'évolution*. Nous nous baserons sur ces dimensions pour comparer et analyser les approches d'évolution proposées par les ADL de première et deuxième générations.

De l'analyse de cette comparaison, nous définissons un ensemble de critères qui représentent soit de points positifs en faveur de la prise en compte de l'évolution, identifiés dans les approches d'évolution étudiées ou encore des limites que nous avons recensés au travers de cette analyse. Nous nous appuierons sur ces critères pour nous positionner et construire un modèle d'évolution d'architectures logicielles, présenté dans le chapitre suivant.

Chapitre 3 : SAEV : un modèle d'évolution d'architectures logicielles

Nous présentons dans ce chapitre, le modèle SAEV (Software Architecture EVolution model) que nous proposons pour la description et la gestion de l'évolution d'architectures logicielles. Nous présentons premièrement les objectifs assignés au modèle SAEV au vu des critères établis dans le chapitre précédent, puis nous présentons les concepts proposés par le modèle ainsi que son mécanisme opératoire pour mener une évolution. Nous montrons ensuite l'application uniforme de SAEV sur le niveau Architectural et le niveau Application d'une architecture logicielle. Nous terminons le chapitre en distinguant les objectifs que nous avons atteint, les limites auxquels nous apportons une réponse dans le chapitre 4 et les objectifs qui restent comme perspectives du modèle SAEV.

Chapitre 4 : Propriétés sémantiques des connecteurs

Nous soulignons au travers du modèle SAEV, l'importance de la propagation des impacts de l'évolution d'un élément architectural vers les autres éléments concernés de l'architecture. Dans ce chapitre nous illustrons le rôle que nous associons aux connecteurs pour automatiser autant que possible cette propagation d'impacts. En effet nous considérons les connecteurs de par leur position d'intermédiaires entre les éléments architecturaux comme les supports idéaux pour véhiculer les propagations entre les éléments architecturaux. Nous proposons ainsi, d'enrichir le concept *Connecteur* par des *propriétés sémantiques* qui permettront de véhiculer les informations sur le degré de corrélation entre les éléments d'une architecture logicielle.

Chapitre 5 : Illustration au travers de l'ADL COSA

Dans ce chapitre nous illustrons nos propositions au travers de l'ADL COSA (Component-Object based Software Architecture) proposé, au sein de notre équipe. Nous illustrons premièrement la prise en

compte des propriétés sémantiques dans l'ADL COSA, en considérant les spécificités des connecteurs de cet ADL. Nous montrons aussi comment ces propriétés peuvent être exploitées au niveau méta pour exprimer la corrélation entre les concepts eux mêmes de l'ADL COSA.

Nous présentons ensuite une spécification de l'exemple didactique du Client/serveur [32] en COSA enrichi par les propriétés sémantiques proposées.

Nous nous baserons sur cet exemple pour illustrer l'application du modèle SAEV sur cette architecture Client/Serveur (au niveau architectural) et sur une application construite à partir de cette architecture (au niveau application). Nous illustrons via cette exemple le rôle des propriétés sémantiques proposées pour les connecteurs.

Chapitre 6 : Expérimentations

Nous présentons dans ce chapitre une approche d'implémentation du modèle d'évolution SAEV, basée sur un outil de transformation de graphes. L'idée exploitée dans cette expérimentation est de réutiliser la théorie formelle de transformation de graphes pour la spécification des concepts et du processus de SAEV et de s'appuyer ensuite sur AGG (Attributed Graph Grammar) [83, 86], un outil qui implémente la transformation de graphes pour implémenter le modèle SAEV.

CHAPITRE 1
Contexte du travail

1.1 Introduction

Le travail présenté dans cette thèse s'intègre dans le cadre des architectures logicielles des systèmes à base de composants et plus spécifiquement, il adresse la problématique de l'évolution dans ces architectures logicielles.

Actuellement un grand intérêt est porté au domaine des architectures logicielles. Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement de tels systèmes. L'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels. Elle permet en effet au concepteur de raisonner sur les propriétés (fonctionnelles et non fonctionnelles) d'un système à un haut niveau d'abstraction. L'architecture logicielle joue le rôle de passerelle entre l'expression des besoins du système logiciel et l'étape de codage du logiciel. Pour décrire les architecture logicielles, plusieurs langages de description (ADL : Architecture Description Languages) ont été proposés. Les ADL 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 plates-formes d'exécution.

Le but de ce chapitre est d'apporter les définitions et les notions de base relatives aux architectures logicielles des systèmes à base de composants. Nous mettrons l'accent sur l'aspect description structurelle de ces architectures logicielles ainsi que sur les notions qui nous permettrons d'aborder la problématique de l'évolution dans les architectures logicielles dans le chapitre suivant.

Nous présentons au préalable, dans la section 1.2, une introduction aux systèmes logiciels à base de composants. Pour cerner la notion d'architecture logicielle, nous présentons dans la section 1.3 quelques définitions de cette notion proposées dans la littérature. Nous nous appuyons sur ces définitions pour apporter la définition de l'architecture logicielle que nous retenons et que nous adoptons tout au long de ce manuscrit. Nous abordons dans la section 1.4, les langages de description d'architectures logicielles au travers de leurs principaux concepts descriptifs, de leurs mécanismes opérationnels ainsi qu'au travers de leurs niveaux d'abstraction pour la description des architectures logicielles. Nous terminons cette section en présentant une classification des ADL en ADL de première et de deuxième générations.

1.2 Systèmes logiciels à base de composants : notions et définitions

Après les technologies objets, qui ont profondément modifié l'ingénierie des systèmes logiciels permettant d'améliorer leur analyse, leur conception et leur développement, une nouvelle ère de conception de systèmes dite "*orienté-composant*" est en passe de s'imposer. L'orienté-composant consiste à concevoir et à développer des systèmes par assemblage de composants réutilisables, à l'image par exemple des composants électroniques ou des composants mécaniques [36] : on parle alors de systèmes logiciels à base de composants.

L'approche par composants est fortement inspirée des composants de circuits électroniques. L'expérience ainsi 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 pour construire un système complet sans pour autant dévoiler les détails de leur implémentation.

Les constructeurs d'applications adoptent de plus en plus cette démarche. Il s'agit alors de composer des composants logiciels de natures diverses pour construire une application entière sans pour autant se soucier des détails internes de ces composants. L'intérêt accru des dix dernières années pour le développement à base de composants est motivé principalement par la réduction des coûts et des délais de développement des applications. En effet, résoudre un problème en le découpant en composants puis en les combinant permet de concevoir plus facilement les applications mais aussi d'améliorer leur niveau de réutilisabilité. Une approche à base de composants semble donc indispensable lors des phases du cycle de vie du logiciel.

Le terme composant existe dans le vocabulaire informatique depuis la naissance du génie logiciel. Il a cependant d'abord désigné des fragments de code alors qu'aujourd'hui, il englobe toute unité de réutilisation [9]. La définition de Szyperski [82] donnée lors d'un atelier à ECOOP en 1996 semble être la plus consensuelle et la plus répondue dans la littérature :

" *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties* "

La traduction de cette définition donne : un composant est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite uniquement ses dépendances de contexte. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par de tierces entités.

Cette définition est valable à différents niveaux de compréhension (spécification, conception, implémentation). Elle décrit les caractéristiques d'un composant logiciel ainsi que ses dépendances avec l'environnement extérieur sans toutefois porter des contraintes sur le type d'interaction avec d'autres composants ou le monde extérieur.

Aujourd'hui, bien qu'il existe une diversité considérable dans les propositions de modèles de composants, tous partagent une base conceptuelle similaire, considérée comme un fondement commun de concepts et d'intérêts pour la description des systèmes à base de composants. Les principaux éléments de ce fondement sont [80, 9, 32, 46, 53] :

- Les composants : les composants sont définis comme des unités de composition qui décrivent et/ou assurent des fonctions spécifiques et qui possèdent des interfaces. Ils peuvent être déployés indépendamment et composés avec d'autres composants.
- Les interactions : les interactions expriment les différents échanges entre les composants d'un système. Elles peuvent être décrites explicitement, autrement dit, réifiées au même niveau que les composants, comme elles peuvent être implicites et donc imbriquées dans la description des composants.
- Les interfaces : les interfaces d'un composant sont des points de communication qui lui permettent d'interagir avec son environnement.
- L'architecture : l'architecture logicielle d'un système à base de composants est une spécifica-

tion des composants d'un système et de leurs interactions. Elle permet de fournir un plan précis approprié pour prédire le comportement d'un système avant de le construire et pour guider son développement.

L 'architecture logicielle joue de plus en plus un rôle important dans le développement et la conception des systèmes à base de composants en offrant un niveau d'abstraction élevé pour la structuration de ces systèmes indépendamment, des langages de programmation et de leur plate forme d'exécution.

Nous nous intéressons dans cette thèse aux architectures logicielles de ces systèmes à base de composants, et plus spécifiquement à la problématique de l'évolution dans ces architectures logicielles. Avant d'aborder cette problématique nous tenterons dans la section suivante, de comprendre ce qu'est une architecture logicielle et les moyens proposés pour leurs description et conception.

1.3 Architecture logicielle à base de composants

Les travaux sur les architectures logicielles (Softwares Architectures en anglais) ont connu une effervescence au début des années 90. Ces travaux soulignaient l'importance d'avoir une structure globale à un niveau d'abstraction élevé d'un système avant sa construction. Cette structure globale devait aboutir à des applications plus structurées, plus faciles à comprendre et plus facile à faire évoluer. Bien que tous ces travaux s'accordent sur l'importance de l'architecture logicielle pour le développement des systèmes à base de composants, la définition consensuelle de la notion d'*architecture logicielle* n' a pas pu être établie. Même si de nombreuses définitions ont été proposées, aucune ne s'est vraiment imposée. La page web dédiée à ce sujet du Software Engineering Institute [39] témoigne de la diversification de ces définitions. Dans cette section, nous présentons dans un ordre chronologique, quelques définitions de la notion d'*architecture logicielle* [39, 73] que nous jugeons significatives et complémentaires. Nous présentons ensuite la définition de l'architecture logicielle que nous retenons tout au long de ce manuscrit. Nous terminons cette section en présentant quelques avantages reconnus pour les architectures logicielles.

1.3.1 Architecture logicielle : quelques définitions

Nous apportons dans cette section les définitions suivantes :

- La définition de Perry et Wolf 1992
- La définition de Shaw et Garlan 1995
- La définition de Bass, Clements et Kazman 1998
- La norme AINSI/IEEE std 1471-2000

Pour chacune de ces définitions, nous mettrons notamment l'accent sur les concepts proposés comme éléments descriptifs d'une architecture logicielle.

Définition de Perry et Wolf 1992

Cet article de Perry et Wolf [66] est considéré comme l'article fondateur de l'architecture logicielle en tant que domaine à part entière du génie logiciel. Nous verrons plus loin que les principales fondations énoncées dans cet article, sont reprises huit ans après dans la norme ANSI/IEEE Std 1471-2000 [34] dédiée à l'architecture logicielle et sont toujours considérées comme les concepts de base admis pour une architecture logicielle.

Par analogie à l'architecture des bâtiments, les auteurs proposent les concepts suivants comme étant les aspects importants d'une architecture logicielle :

- Un modèle de l'architecture logicielle ;
- Un style architectural ;
- Des vues architecturales ;

Le *modèle de l'architecture* : est définit comme un triplet (*Elements, Form, Rationale*). Les éléments architecturaux (*Elements*) peuvent être de trois natures : les éléments de données, les éléments de transformation de données et les éléments de connexion entre les éléments précédents. La forme architecturale (*Form*) correspond à la fois à des propriétés et à des relations. Les propriétés définissent des contraintes sur les éléments pris individuellement. Les relations définissent des contraintes sur la disposition des éléments entre eux. Les critères de choix (*Rationale*) capturent les motivations de l'architecte pour le choix d'une architecture plutôt qu'une autre (temps, performances, etc.).

Le *style architectural* dénote une représentation abstraite regroupant des caractéristiques et des décisions de conception communes pour des architectures similaires.

Les *vues architecturales* permettent de représenter différents aspects d'une même architecture. Ces vues sont représentées de manière séparée mais sont interdépendantes.

Nous retiendrons ainsi de cette définition qu'une architecture peut être décrite par un modèle architectural. Ce modèle architectural est structuré en un ensemble d'éléments reliés entre eux. Des propriétés permettent de contraindre les liens entre ces éléments. De plus, une architecture logicielle peut être décrite au travers de plusieurs vues, chacune considérant un aspect particulier de cette dernière.

Définition de Garlan et Perry 1995

La définition donnée par les auteurs est issue d'un groupe de discussion sur les architectures logicielles du SEI (Software Engineering Institute). L'architecture logicielle est définie comme :

" *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time . . .* " [33].

L'architecture logicielle est définie comme la structure d'un système ou d'un programme. Cette structure est composée d'un ensemble d'éléments désignés par composants, des relations entre ces éléments et des règles qui gouvernent la conception et l'évolution de ces éléments et de ces relations.

Deux autres principes que nous jugeons importants sont évoqués dans cet article :

- Formalisation de l'architecture logicielle : l'architecture logicielle tente de fournir un cadre formel, des notations et des outils pour l'étude de l'organisation des structures de haut niveau

d'un système, au delà des structures de données et de l'algorithmique.

– Distinction des composants de leurs interactions : il est nécessaire de dissocier dans une description architecturale, l'implémentation des composants des interactions entre ces composants. Ce nouveau modèle de l'architecture se focalise sur l'interaction en identifiant des entités de connexions appelées connecteurs et en décrivant de manière précise les protocoles de communications entre les composants.

Cette définition souligne que l'architecture logicielle se positionne bien au-delà du code source, qu'elle définit la structure à un haut niveau d'abstraction d'un système, que des notations et un cadre formels s'imposent pour la description de ces structures et que les interactions entre les composants doivent être distinguées et considérées au même niveau que les composants.

Définition de Bass, Clements et Kazman 1998

" *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them . . .* " [10].

Cette définition évoque qu'un système peut comprendre plusieurs structures et qu'aucune d'entre elles ne peut être considérée comme définissant l'architecture de ce système. De la même manière qu'il existe plusieurs structures, il existe plusieurs types de composants (modules, processus), plusieurs types d'interactions entre les composants (subdivision, synchronisation) et différents contextes (développement, exécution).

Un autre point important évoqué par ces auteurs est la distinction entre l'architecture concrète d'un système et la description architecturale de cette dernière. L'architecture concrète d'un système correspond à l'architecture définie dans le code source. Elle correspond à la réalité. Une description architecturale est une abstraction du code source qui tente de décrire au mieux cette architecture concrète. Elle sert pour la conception, la compréhension et les analyses architecturales.

Les auteurs ont souligné aussi le fait que l'architecture logicielle ne se limite pas à la structure du système et que le comportement des composants fait parti de la description de l'architecture. Le comportement d'un composant décrit comment les autres composants interagissent avec lui.

Nous retiendrons ainsi, de cette définition, qu'il existe une distinction entre l'architecture concrète d'un système et sa description architecturale, l'existence de plusieurs structures pour un même système et la considération des comportements des composants comme faisant partie d'une description architecturale.

La norme AINSI/IEEE std 1471-2000

A notre connaissance, l'IEEE std 1471-2000 est la première norme formelle pour les descriptions architecturales de systèmes logiciels. Nous nous attardons sur sa présentation. Cette norme a été développée par le groupe de travail d'architecture d'IEEE (Institute of Electrical and Electronics Engineers) en collaboration avec des représentants du monde industriel et des organisations académiques, et a été sujette aux critiques intensives de plus d'une centaine de chercheurs du domaine, avant d'être publiée. Il s'agit des recommandations pratiques de l'IEEE pour les descriptions architecturales des systèmes logiciels. La figure suivante montre le modèle conceptuel de description architecturale tel qu'il a été défini

dans l'IEEE std 1471-2000.

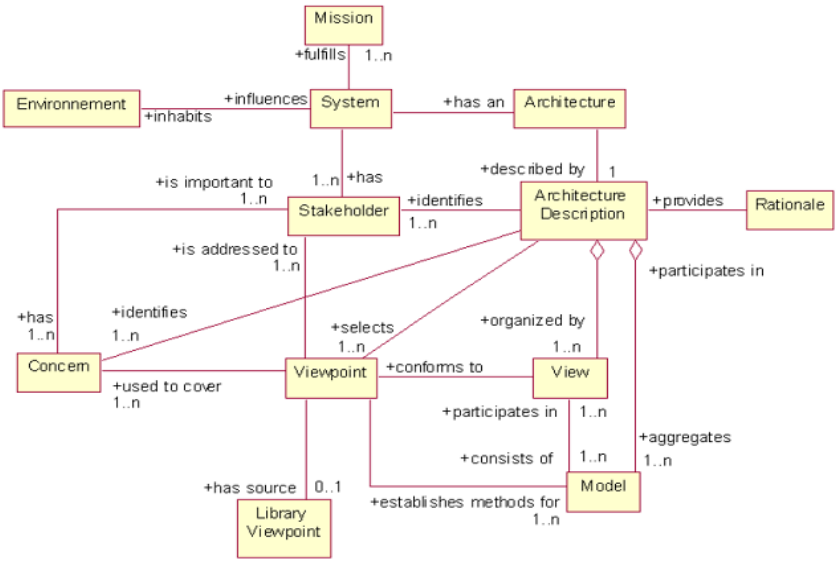

Figure 1.1 – Modèle conceptuel pour une description architecturale : Extrait de [34]

Dans ces recommandations, un système (*System*) est défini comme une collection de *composants* organisés afin d'accomplir une fonction ou un ensemble spécifique de fonctions. Le terme système englobe des applications individuelles, des sous-systèmes, des familles de produits, etc. A partir de cette définition, il s'en suit que n'importe quoi peut être un système pourvu qu'il satisfasse certains buts (accomplir une ou plusieurs fonctions).

Un système est hébergé dans un environnement (*Environnement*) qui l'influence et qui peut être influencé par ce système [10]. L'environnement, parfois appelé contexte, détermine les conditions et les circonstances du développement opérationnel, politique et autres influences sur le dit système.

Un système doit remplir un certain nombre de missions *Mission* de son environnement. Par mission les auteurs entendent une fonctionnalité du système qui est destinée à un ou plusieurs *acteurs* et qui doit répondre à un ensemble d'objectifs

Un acteur peut être tout intervenant (*Stakeholder*) dans le système (utilisateur final, développeur, architecte, concepteur, etc). Chacun de ces acteurs a des tâches et donc des intérêts différents par rapport au système et à son architecture. Aussi, dans la description de l'architecture, tous ces intérêts doivent être retrouvés.

Un système a une architecture qui peut être représentée par une description architecturale (*Architecture Description*). La description architecturale est définie comme une collection de produits pour documenter une architecture. Cette norme indique que pour une architecture correspond une seule description architecturale, hors nous sommes en accord avec Sanlaville [73] et [10] que pour une architecture, plusieurs descriptions peuvent être établies. La description architecturale peut être décomposée en une ou plusieurs vues (*View*). Chaque vue couvre un ou plusieurs centres d'intérêts des acteurs.

Une vue est définie comme une représentation d'un système entier dans la perspective de satisfaire un ensemble relatif d'intérêts des acteurs. Une vue est créée selon des règles et des conventions définies dans un point de vue (*Viewpoint*). Un point de vue peut être considéré comme le méta modèle de plusieurs vues. Il est défini comme une spécification des conventions pour construire et utiliser une vue.

En plus des informations décrites dans les vues, une description architecturale peut contenir d'autres informations telle que la vue d'ensemble d'un système et/ou son support de raisonnement. Cette information n'est pas forcément décrite selon une définition du point de vue, mais peut être trouvée dans des recommandations de documentation organisationnelle.

Une description architecturale sélectionne un ou plusieurs points de vue. Ce choix dépend bien entendu des centres d'intérêts des acteurs. Un point de vue peut être défini avec la description architecturale, mais il peut également être défini ailleurs et être seulement utilisé dans la description architecturale. De tels points de vue définis à l'extérieur sont appelés des points de vue de bibliothèque (*Library Viewpoint*).

Enfin, une vue peut se composer d'un ou plusieurs modèles (*Model*) et un modèle peut participer à une ou plusieurs vues. Chaque modèle est défini selon les méthodes établies dans la définition correspondante du point de vue. La description architecturale agrège les modèles qui sont organisés en vues.

Les auteurs de cette norme ont regroupé les concepts nécessaires pour une description architecturale. Ils ne se sont pas focalisés sur les briques de base de cette description architecturale, qu'ils énoncent comme juste un ensemble de produits, ceci est sans doute pour garder une définition d'une architecture logicielle qui soit largement acceptée. Ils n'affirment pas que tous ces concepts doivent être nécessairement considérés pour parler d'architecture logicielle. Par exemple la notion de vues architecturales, de besoins particuliers, ne sont pas forcement abordé dans toutes les descriptions architecturales.

Nous avons présenté jusqu'ici quelques définitions qui ont tenté d'apporter une réponse à ce qu'est une architecture logicielle. Nous les avons présenté dans un ordre chronologique pour montrer l'évolution de ces définitions. Nous avons souligné pour chaque définition les points importants qui ont apporté des contributions sur ce qu'est une architecture logicielle. La norme AINSI/IEEE std 1471-2000 a capitalisé ces définitions en présentant un modèle conceptuel d'une description architecturale. Ces différentes définitions nous ont permis de cerner la notion d'architecture logicielle. Dans la section suivante nous présentons la définition que nous retenons pour l'architecture logicielle.

1.3.2 Architecture Logicielle : définition adoptée

Notre but dans cette section n'est pas d'apporter une nouvelle définition de la notion d'architecture logicielle, mais de présenter la définition de l'architecture logicielle que nous retenons et sur laquelle nous nous baserons tout au long de ce manuscrit.

Nous considérons l'architecture logicielle comme la *structure* d'un système à un *haut niveau d'abstraction* exposant son organisation comme une collection de *composants* qui assurent les fonctions de calcul et des *connecteurs* qui relient ces composants et coordonnent leurs interactions pour satisfaire les fonctionnalités du système et les contraintes globales d'intégrités (invariants structurels, coordination, etc). Cette structure considère comme brique de base le concept de composant, nous parlerons alors d'*architecture logicielle à base de composants*. Une architecture peut être décrite par une ou plusieurs descriptions architecturales. Une description architecturale peut se focaliser sur l'aspect structurel et/ou sur l'aspect comportemental d'un système.

Dans ce travail nous nous intéressons à la description structurelle des architectures logicielles. Nous n'abordons pas l'aspect comportemental d'une architecture logicielle, ni les notions de vues et de points de vues.

1.3.3 Avantages des architectures logicielles

La description de l'architecture logicielle s'impose de plus en plus comme une étape indispensable du développement des systèmes à base de composants en permettant au concepteur de raisonner sur les propriétés fonctionnelles et non fonctionnelles (fiabilité, sécurité, ...) du système à un haut niveau d'abstraction. Il est bien admis aujourd'hui qu'une bonne architecture peut amener à un produit qui répond aux besoins des utilisateurs et qui peut être modifié facilement et qu'une mauvaise architecture peut avoir des conséquences désastreuses sur le système. D'autres avantages ont été reconnus pour les architectures logicielles à base de composants tels que :

- La dissociation des interactions entre les composants de leur implémentation, ainsi que la définition explicite de ces interactions dans la plupart des descriptions architecturales ;
- La considération des interfaces comme des entités de première classe et leur description explicite ;
- La description d'une architecture globale d'un système peut être spécifiée avant de compléter la construction de ses composants (implémentation) ;
- La possibilité de définir des représentations hiérarchiques très riches, notamment via la notion de la composition ;

Au-delà de leur apport de structuration, les architectures logicielles constituent un guide pour l'évolution des systèmes logiciels à base de composants. En effet, aborder la problématique de l'évolution au niveau de l'architecture permet au concepteur de raisonner à un haut niveau d'abstraction sur l'ajout, la suppression, la modification des composants et/ou des connecteurs ou la réorganisation de l'architecture, sans pour autant se noyer dans les détails de l'implémentation de ses éléments. Ce dernier point fera l'objet des chapitres à venir.

Nous avons présenté jusque là quelques définitions proposées dans la littérature de la notion d'architecture logicielle. Nous avons présenté ensuite la définition que nous retenons pour cette notion. Nous avons souligné quelques avantages reconnus pour les architectures logicielles. Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécifications architecturales. Les langages de description d'architectures constituent une bonne réponse. Nous les abordons dans la section suivante.

1.4 ADL : Langages de description d'architecture logicielle à base de composants

Un ADL (Architecture Description Language) est un langage offrant des formalismes pour modéliser une architecture logicielle à base de composants d'un système. Un ADL fournit aussi bien une syntaxe concrète qu'un cadre conceptuel pour caractériser des architectures logicielles [31]. Le cadre conceptuel reflète les caractéristiques du domaine pour lequel l'ADL est prévu. Il englobe aussi leur sémantique (par exemple, les CSP [37], les machines à états [87]).

Les langages de description d'architectures trouvent leurs racines dans les langages d'interconnexion

de modules (Module Interconnection Languages) des années 70 [23]. Ils sont aujourd'hui dans une phase de maturité. Parmi les représentants actuels de ces langages nous citons : Rapide [47], UniCon [75], ACME [31, 32], Wright [4, 3], C2SADEL [53], Darwin [49, 48] , Fractal [19], COSA [76, 42] , SafArchie [7] [8], UML2.0 [35], xADL2.0 [21].

Le but de cette section est de présenter les concepts communément admis par la majorité de ces ADL pour la description structurelle d'une architecture logicielle. Nous présentons, en plus de ces concepts, les mécanismes opérationnels de ces ADL qui interviennent dans la description structurelle d'une architecture logicielle. Nous illustrons ensuite les différents niveaux d'abstraction de description d'une architecture logicielle. Nous terminons cette section en présentant une classification de ces ADL sur laquelle nous nous baserons dans le chapitre suivant pour illustrer les approches d'évolution d'architectures logicielles dans les ADL.

1.4.1 Concepts de base des ADL

Les concepts de base d'une description architecturale sur lesquels s'accordent l'ensemble des travaux portant sur les ADL reposent sur les concepts de : *Composant*, *Connecteur*, *Configuration* et *Interface*. Un ADL doit fournir ainsi les notations et les sémantiques nécessaires pour la spécification de ces concepts de base. Le diagramme de classes UML [35] suivant, décrit ces concepts ainsi que les liens qui existent entre eux :

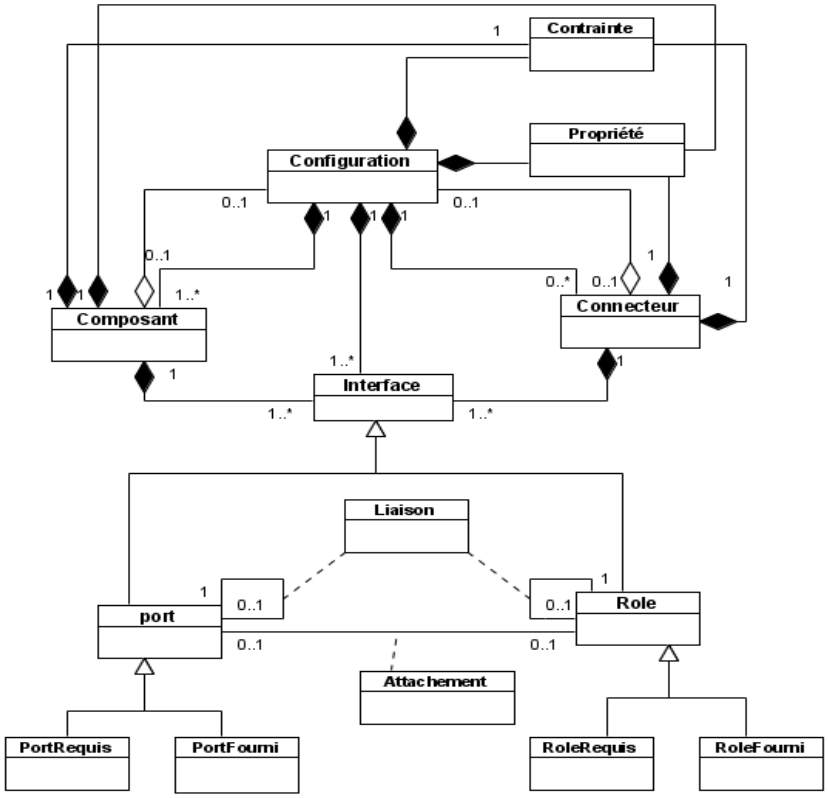

Figure 1.2 – Les concepts de base des langages de description d'architectures logicielles

Les paragraphes suivants apportent les définitions communément admises pour ces concepts illustrés dans ce diagramme de classes.

1.4.1.1 Composant

Un composant représente une unité de calcul ou de stockage de données à laquelle est associée une unité d'implémentation. Il peut être aussi petit qu'une procédure simple ou aussi grand qu'une application. Il est l'élément de base, et doit être composable, auto descriptif, configurable, réutilisable et autonome [9]. Un serveur, une base de données, une fonction mathématique sont des exemples de composants. Tous les ADL existants modélisent les composants sous une forme ou une autre. Un composant est décrit par une interface (cf. sous section 1.4.1.4), un type, des contraintes et des propriétés non fonctionnelles :

a. Type d'un composant : le type est au paradigme composant ce qu'est une classe au paradigme objet. Il permet l'encapsulation de fonctionnalités et des éléments internes en vu d'être réutilisable. Un type de composant peut être instancié plusieurs fois dans une même architecture ou peut être réutilisé dans d'autres architectures [55]. Les instances d'un type ont les mêmes structures et comportement que celui du type. La modélisation explicite des types facilite également la compréhension et l'analyse d'une architecture.

b. Contrainte d'un composant : la contrainte d'un composant précise des restrictions qui doivent être vérifiées afin de respecter les utilisations prévues d'un composant et d'établir les dépendances parmi ses éléments internes. Les contraintes d'un composant peuvent être définies soit dans un langage de contrainte séparé, soit spécifiées directement en utilisant les notations de l'ADL hôte.

c. Propriété non fonctionnelle d'un composant : la propriété non fonctionnelle d'un composant représente toute contrainte particulière liée à l'environnement d'utilisation du composant. Les propriétés non fonctionnelles sont nécessaires pour permettre la simulation du comportement des composants, leur analyse, leur traçabilté depuis leur conception jusqu'à leur implémentation et l'aide dans la gestion de projet (par exemple, en définissant des conditions de performance rigoureuses, le développement d'un composant peut devoir être assigné au meilleur ingénieur). Peu d'ADL offrent une structure d'accueil de propriétés non fonctionnelles malgré tous leurs avantages potentiels.

1.4.1.2 Connecteur

Un connecteur est un bloc de construction utilisé pour exprimer les interactions entre composants ainsi que les règles qui gouvernent cette interaction. Les connecteurs sont des entités architecturales qui relient des ensembles de composants et agissent en tant que médiateurs entre eux. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure et l'émission d'évènements. 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 [31]. Contrairement aux composants, les connecteurs peuvent ne pas correspondre à des unités de compilation. Cependant, les spécifications de connecteurs dans un ADL peuvent également contenir des règles pour implémenter un type spécifique d'un connecteur.

De manière générale, comme le souligne [63, 56], 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 , 2) les ensembles énumérés de connecteurs prédéfinis, comme ceux par exemple

d'UniCon [75] et 3) les connecteurs dont les sémantiques sont définies par les utilisateurs, comme ceux par exemple de Wright.

- *Premier groupe* : ce premier groupe d'ADL ne réifie pas le concept de connecteur comme entité de première classe. Les interactions entre les composants sont spécifiées en termes de liaisons directes (bindings en anglais) entre leurs interfaces. Ces liaisons sont modélisées en ligne et ne peuvent en aucun cas être renommées ou réutilisées. Ceci est le cas des interactions de Darwin [49] et de Rapide [47].
- *Deuxième groupe* : les ADL de ce groupe proposent des types de connecteurs *énumérés*. Généralement ces types de connecteurs ne peuvent pas être instanciés ni être modifiés. Le concepteur ne peut pas utiliser d'autres connecteurs en dehors de ceux définis par l'ADL lui même. Par exemple, un connecteur dans UniCon est spécifié par un protocole. Un protocole est défini par un type de connecteurs, un ensemble de propriétés et des rôles typés qui servent comme points d'interaction avec les composants. Les types de connecteurs sont : *FileIO*, *Pipe*, *Procedure Call* , *RPC*, *Scheduler*, *RT*, *Data Access* et *PL Bundler*.
- *Troisième groupe* : les ADL de ce groupe considèrent les connecteurs comme entités de première classe au même titre que les composants. Par exemple, dans le langage Wright [5], les connecteurs sont définis comme un ensemble de rôles et une glu. Chaque rôle définit le comportement d'un participant dans l'interaction. La glu définit comment les rôles interagissent entre eux.

Définir explicitement les connecteurs permet de dissocier la description des interactions de la description des composants eux mêmes, ce qui permet de faciliter la compréhension du système et d'augmenter la réutilisation des composants. Les connecteurs sont de plus en plus considérés explicitement par les ADL. Un connecteur explicite est décrit par une interface (cf. sous section 1.4.1.4), un type, des contraintes et des propriétés non fonctionnelles :

a. Type d'un connecteur : la communication au niveau architecture peut faire appel à des protocoles complexes. Pour abstraire ces protocoles et les rendre réutilisables, les ADL doivent modéliser les connecteurs comme des types. Évidement, seuls les ADL qui considèrent les connecteurs comme entités de première classe distinguent les types de connecteurs de leurs instances, tels que les ADL Wright [4] [3] et ACME [32].

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

c. Propriété non fonctionnelle d'un connecteur : les propriétés non fonctionnelle d'un connecteur ne sont pas forcément obtenues des spécifications de sa sémantique. Elles représentent en général les informations additionnelles nécessaires pour l'implémentation correcte d'un connecteur. Modéliser les propriétés non fonctionnelles des connecteurs permet la simulation de leur comportement, leur analyse, et la sélection de connecteurs appropriés et leurs correspondances [55].

1.4.1.3 Configuration

Une configuration représente un graphe de composants et de connecteurs et définit la façon dont ils sont reliés entre eux. Cette notion 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. Le rôle clé d'une configuration est de faciliter la communication entre les différents intervenants dans le développement d'un système. En effet, en faisant abstraction des détails des composants et des connecteurs, les configurations offrent une vision du système à un haut niveau d'abstraction qui peut être potentiellement comprise par des personnes avec différents niveaux d'expertise et de connaissance techniques [76]. Une configuration peut être aussi décrite par une interface (cf. sous section 1.4.1.4), un type, des contraintes et des propriétés non fonctionnelles.

a. Type de configuration : pour permettre la construction de différentes architectures d'un système, une configuration doit aussi être définie comme une classe instançiable. Ainsi, on peut déployer une architecture donnée de plusieurs manières, sans réécrire le programme de configuration/déploiement [42]. Quelques ADL uniquement permettent la définition des types de configuration tel que COSA [76] et SafArchie [7].

b. Contraintes de la configuration : une contrainte de la configuration décrit les dépendances entre les composants et les connecteurs de cette configuration. Ces contraintes sont aussi importantes que celles spécifiées au niveau des composants et des connecteurs. Cependant, la plupart des ADL se sont focalisés plus sur des contraintes locales (au niveau des composants et connecteurs) que sur les contraintes au niveau des configurations.

c. Propriété non fonctionnelle d'une configuration : certaines propriétés non fonctionnelles ne sont reliées ni aux connecteurs ni aux composants et doivent être exprimées au niveau d'une configuration. Les propriétés non fonctionnelles au niveau des configurations sont nécessaires pour choisir par exemple les composants et les connecteurs appropriés, pour réaliser des analyses, pour appliquer et faire respecter des contraintes, etc .

1.4.1.4 Interface

L'interface est la seule partie visible des trois concepts précédents. Elle fournit un ensemble de points d'interaction et de *services* (messages, opérations, variables). Ces services peuvent être des services fournis par un composant, un connecteur ou une configuration comme ils peuvent être aussi des *besoins* nécessaires pour que ces derniers puissent rendre leurs services. Les points d'interaction d'un composant et d'une configuration sont appelés *ports*. Deux types de ports peuvent être distingués :

- les *ports services* (fournis), qui exportent les services des composants et des configurations.
- les *ports besoins* (requis), qui importent les besoins des composants et des configurations.

Les points d'interaction d'un connecteur quant à eux sont appelés *rôles* et décrivent les rôles des participants à l'interaction. On distingue également les *rôles requis* et les *rôles fournis*.

Les interactions entre les interfaces sont définies par des *attachements* ou des *bindings*.

- Attachement (liaison) : représente la connexion entre un port d'un composant et un rôle d'un connecteur.
- Binding (correspondance) : dans une description architecturale un composant et une configuration (respectivement un connecteur) peuvent être composés hiérarchiquement d'un ou plusieurs sous-composants (respectivement sous-connecteurs). Les connexions entre les ports (respectivement les rôles) d'une configuration ou d'un composant composite (respectivement d'un connecteur composite) et les ports (respectivement les rôles) de leurs sous composants (respectivement sous connecteurs) sont appelées correspondances(bindings).

Les attachements et les binding sont spécifies au moment de la spécification de la configuration.

Après avoir présenté les concepts des ADL pour la description d'une architecture logicielle, nous présentons dans la section suivante les mécanismes opérationnels des ADL qui interviennent dans la description d'une architecture logicielle.

1.4.2 Mécanismes opérationnels des ADL

Les ADL fournissent des outils de spécification et de développement pour pouvoir prendre en compte des systèmes à grande échelle susceptibles d'évoluer. Aussi, pour augmenter la réutilisabilité et la compréhensibilité des architectures, des mécanismes spéciaux doivent être pris en compte par les ADL. En plus de leurs apports pour la structuration des spécifications des architectures logicielles, nous verrons plus loin dans le chapitre 2, que ces mécanismes sont indispensables pour améliorer l'évolutivité et le passage à l'échelle de ces architectures logicielles. Les paragraphes suivants présentent les mécanismes que nous avons identifié au travers des ADL étudiés [69, 77].

1.4.2.1 Instanciation

Elle constitue le mécanisme de base le plus répandu dans les ADL. Il est identique à celui de l'approche objet. Chaque élément architectural est représenté par un type. On peut ainsi assimiler l'instance d'un élément architectural à un objet, qui doit respecter la structure et le comportement donnés par son type. Ainsi dans une spécification d'architecture, un type peut être instancié plusieurs fois et chaque instance correspond à une implémentation différente de l'élément architectural. Par exemple, si un ensemble de propriétés est défini pour un type donné alors chaque instance de ce modèle (type) doit avoir les mêmes propriétés.

1.4.2.2 Héritage et sous-typage

L'héritage et le sous-typage permettent la réutilisation des éléments architecturaux, mais présentent quelques différences. La relation d'héritage fournit une classification de types (composant/connecteur/configuration) par une réutilisation de ceux-ci, alors que le sous-typage se base sur la réutilisation des constituants de ces types.

L'*héritage* est une représentation hiérarchique de types, tel qu'un sous-type hérite d'un ou plusieurs types. Typiquement, le sous-type augmente ou redéfinit la structure et le comportement existants de son type. C'est un outil puissant pour l'évolution en permettant à un type hérité d'être modifié en ajoutant, en supprimant, ou en changeant sa structure. Le lien d'héritage est peu utilisé dans les ADL.

Le *sous-typage* est le mécanisme le plus souvent utilisé par les ADL. Il peut être défini par le fait que les valeurs d'un sous-type sont utilisables dans le contexte prévu pour le super-type et sans erreurs [15]. ACME propose par exemple le sous-typage structurel des composants et les connecteurs grâce à la clause *extends*. Les travaux sur C2 dans [52, 53] dénombrent plusieurs déclinaisons du mécanisme de sous-typage : de *nom*, d'*interface*, de *comportement*, d'*implémentation*. Le sous-typage *hétérogène* est alors la combinaison de ces sous-typages par les clauses *and*, *or* et *not*. D'autres ADL comme UniCon, définissent les composants par énumération donc ils n'adoptent pas les mécanismes d'héritage et de sous-typage.

1.4.2.3 Composition

Les architectures sont nécessaires pour décrire des systèmes logiciels à différents niveaux de détails, où les comportements complexes sont soit explicitement représentés soit encapsulés dans des composants et des connecteurs. La composition est ainsi la capacité à pouvoir construire des éléments architecturaux à partir d'éléments déjà existants, en les encapsulant dans une plus grande structure. On parle alors de *composants-composites* ou *connecteurs-composites*. Un ADL doit pouvoir prendre en compte le fait qu'une architecture entière devienne un composant simple dans une plus grande architecture. Par conséquent, la prise en compte de la composition ou de la composition hiérarchique est cruciale. Ce mécanisme s'appuie sur les interfaces, qui jouent le rôle important de connexion, entre composants internes eux-mêmes, et entre la super-structure et ses constituants qui la composent.

1.4.2.4 Raffinement et traçabilité

Le raffinement architectural est le passage d'une architecture abstraite vers une architecture plus concrète. Cette dernière contient plus d'informations qui sont consistantes avec les informations de l'architecture abstraite [11]. L'argument le plus souvent évoqué pour créer et utiliser les ADL est qu'ils sont nécessaires pour établir le pont entre les diagrammes informels de haut niveau de type "boîtes-et-lignes" et les langages de programmation qui sont considérés comme des langages de bas niveau. Comme les modèles architecturaux peuvent être définis à différents niveaux d'abstractions ou de raffinement, les ADL fournissent aux développeurs et aux concepteurs des outils expressifs et sémantiquement riches pour les spécifier. Les ADL doivent également permettre une traçabilité des changements à travers ces niveaux de raffinement. Notons que le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les ADL actuels.

Après avoir présenté les concepts et les mécanismes opérationnels communément admis par la majorité des ADL, nous montrons dans la section suivante qu'une architecture logicielle peut être décrite par un ADL au travers de plusieurs niveaux d'abstraction.

1.4.3 Niveaux d'abstraction d'une description d'architecture logicielle

La spécification d'une architecture logicielle par un ADL peut passer par plusieurs niveaux d'abstraction. Le contenu de chaque niveau dépend des concepts réifiés par cet ADL. En effet, nous constatons que dans tous les ADL le concept *composant* est réifié et considéré comme entité de première classe. La majorité des ADL distinguent le *composant type* du *composant* qui représente l'instance de ce composant type. Par conséquent, la majorité des ADL distinguent deux niveaux de description pour les composants, le *niveau type* et le *niveau instance*. Cette distinction n'est pas considérée pour tous les autres concepts. Nous distinguons ainsi :

- Les ADL qui réifient le concept de *connecteur* et non le concept de *configuration* tel que : Wright [4, 3].
- Les ADL qui réifient les concepts de *connecteur* et d'*interface* et non le concept de *configuration*, tel que xADL2.0 [21].
- Les ADL qui réifient les concepts d'*interface* et de *configuration* et non le concept de *connecteur*, tel que SafArchie [7, 8] ;

- Les ADL qui réifient tous les concepts *composant*, *connecteur*, *configuration* et *interface*, tel que COSA [42, 76].

  Ainsi de manière générale, nous considérons [70], [71] que la description d'une architecture logicielle peut être établie au travers de trois niveaux d'abstraction que nous dénommons : le **niveau Méta**, le **niveau Architectural** et le **niveau Application** illustrés par la figure 1.3 :

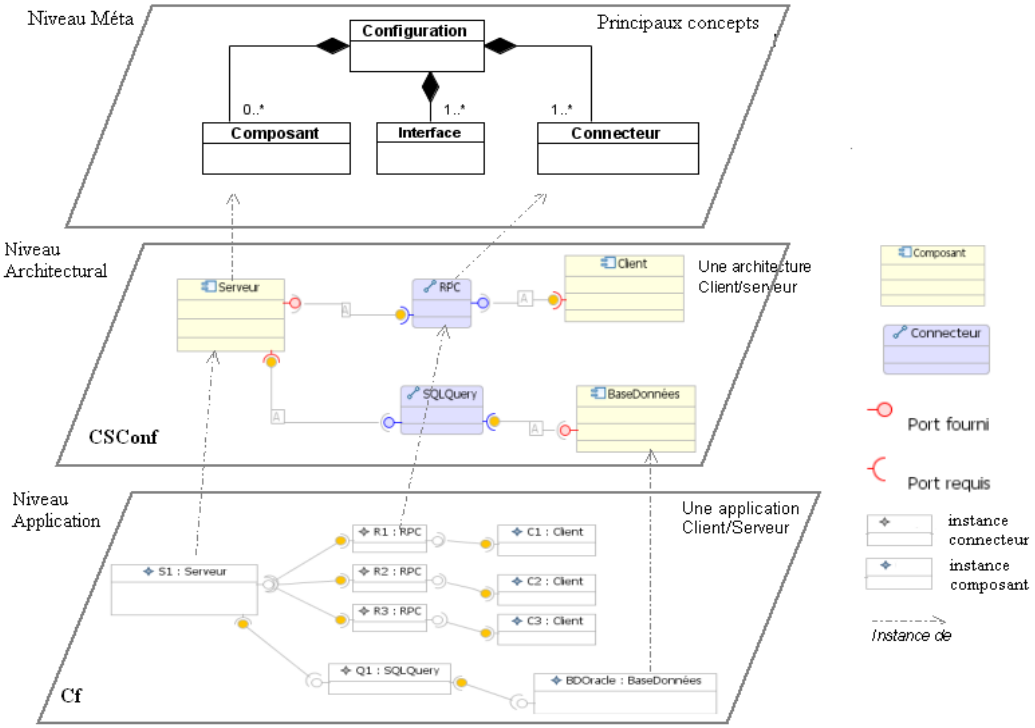

Figure 1.3 – Niveaux d'abstraction de description d'architectures logicielles

1.4.3.1 Niveau Méta

Il représente le niveau de définition de tous les concepts architecturaux réifiés qu'un ADL peut proposer pour la description d'une architecture logicielle. Chaque ADL peut être décrit au niveau Méta au travers des concepts qu'il propose et des différents liens entre ces concepts. Ce niveau permet ainsi de définir les caractéristiques des concepts de chaque ADL qui doivent être respectés à la spécification d'une architecture en utilisant cet ADL. Les concepts de *configuration*, *composant*, *connecteur*, *interface* sont des exemples de concept du niveau Méta. D'autres travaux tel que [80, 76] utilisent ce niveau pour construire une méta modélisation de description d'architecture logicielle qui permet entre autres l'échange d'architectures entre les différents ADL.

Un extrait du méta modèle de l'ADL COSA est illustré dans la figure 1.3, via le diagramme de classes d'UML [35].

1.4.3.2 Niveau Architectural

Il représente le niveau de description d'architectures logicielles en utilisant un ADL donné, lui même défini auparavant au niveau Méta. A ce niveau plusieurs types de composants, de connecteurs, de configurations peuvent être définis. Ces types doivent être conformes aux concepts de l'ADL définie au niveau Méta.

Le niveau Architectural offre plusieurs avantages, tels que la réutilisation des différents types définis dans d'autres architectures logicielles, la délimitation d'un cadre pour l'architecture, c'est à dire offrir un ensemble de contraintes que devront respecter les futures architectures d'applications et de s'appuyer sur ces contraintes pour permettre un premier niveau de contrôle de cohérence entre les types d'éléments architecturaux.

La figure 1.3 précédente présente au niveau Architectural une architecture de type *Client-serveur* composée d'une *Configuration* type `CSConf` ; de trois composants types : `client`, `serveur` et `BaseDonnées` et de deux connecteurs type `RPC` et `SQLQuery`. La spécification présentée est décrite en utilisant l'ADL COSA [79, 76].

1.4.3.3 Niveau Application

Il représente le niveau de description des applications construites conformément à leurs architectures décrites au niveau *Architectural*. Chaque élément architectural du niveau Application est une instance d'un élément type du niveau Architectural. Par exemple à partir de l'architecture type *Client-serveur* de la figure 1.3, on peut construire l'application suivante composée de la configuration `Cf` : Instance de `CSConf` ; de trois composants `C1`, `C2`, `C3` : instances du composant `Client` ; du composant `BDoracle` : instance du composant `Basedonnées` ; `S1` : instance du composant `serveur` ; de trois connecteurs `R1`,`R2`,`R3` : instances du connecteur `RPC` ; `Q1` : instance du composant `SQLQuery`.

Toute description d'une architecture logicielle peut ainsi être positionnée sur ces trois niveaux d'abstraction. Ceci permettrai entre autres :

- l'aide à la compréhension de la description de l'architecture logicielle ;
- l'identification des concepts réifiés de ceux qui ne le sont pas ;
- la distinction des liens entre les niveaux d'abstraction ;

Nous verrons plus loin dans le chapitre 2, que cette distinction est importante à considérer lorsque la problématique de l'évolution est abordée. En effet, il est nécessaire de préciser sur quel niveau d'abstraction une évolution est opérée ainsi que sur quels autres niveaux, peut-elle avoir des impacts.

Nous avons présenté les concepts et mécanismes proposés par les ADL pour la description d'une architecture logicielle ainsi que les niveaux d'abstraction de cette description. Nous reviendrons sur la présentation d'exemples d'ADL dans le chapitre suivant. Pour compléter la caractérisation des ADL, nous présentons dans le section suivante une classification des ADL que nous adopterons dans le chapitre suivant, pour aborder la problématiques de l'évolution architecturale dans les ADL.

1.4.4 Classification des ADL adoptée

Plusieurs critères ont été proposés dans la littérature pour classifier et comparer les ADL [1, 65, 55], tels que :

- L'aspect du système auquel ils s'intéressent : structurel et/ou comportemental ;
- Leurs formalismes de base : formels ou semi formels ;
- Leur origine : écoles américaines ou écoles européennes.

Pour classer les ADL, nous avons opté pour une classification basée sur le critère de *spécificité* ou non d'un ADL à un domaine particulier. Un domaine d'un ADL peut être le type de système auquel il est destiné (dynamique, concurrent, réparti, etc) ou un domaine d'activité (aéronautique, système de familles de produits) ou toute autre caractéristique limitant l'utilisation d'un ADL.

En se basant sur ce critère, cette classification distingue ainsi les ADL spécifiques à un domaine appelés alors ADL de *première génération*, des ADL qui peuvent adresser plusieurs domaines, désignés par les ADL de *deuxième génération* [24, 21].

1.4.4.1 ADL de première génération

Les ADL de première génération adressent des problématiques de domaines spécifiques. Ils se focalisent généralement sur des propriétés particulières d'un système et proposent souvent une syntaxe figée et propre à ces propriétés. Parmi ces ADL, nous citons : Darwin [49, 48] destiné aux systèmes distribués et qui se base sur π-calculus pour la spécification du comportement d'un système, Wright [4, 3] pour la spécification des systèmes parallèles communicants et qui se base sur CSP pour la spécification du comportement et C2SADEL [53] est destiné aussi aux systèmes distribués et dont la spécification est établie spécifiquement dans le style C2 [54].

1.4.4.2 ADL de deuxième génération

Les constats et le recul pris par rapport aux ADL vers la fin des années 1990, ont permis de souligner qu'un ensemble commun de concepts, " *une ontologie*", peut être définie. Cette ontologie regroupera l'ensemble des concepts partagés par la majorité des ADL. Il sera alors possible de définir un ADL générique qui combinera l'ensemble de ces concepts et ne posera aucune contrainte sur les propriétés du système à développer. Ce type d'ADL peut, ainsi servir par conséquent comme une passerelle entre les différents ADL existants. Les ADL les plus représentatifs de cette catégorie sont ACME [33, 32] et xADL2.0 [21, 20].

ACME a pour but principal de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADL. Comme le langage ACME se focalise sur l'aspect structurel des architectures logicielles, il offre un mécanisme d'annotation (annotation mechanism) pour favoriser l'intégration d'autres informations spécifiques ne concernant pas la structure et apportées par certains ADL (telle que la spécification du comportement par exemple).

xADL2.0 est défini comme un langage de description flexible et extensible de schémas XML (eXtensibls Markup Lanuage) [12]. Ses auteurs ont largement tirer profit d'XML pour apporter cette souplesse et bénéficier des outils supports d'XML. Le langage offre une base composée des concepts communément admis par les ADL et permet ensuite au concepteur de définir d'autres concepts qu'il juge adaptés pour son système.

Les ADL de deuxième génération tentent d'offrir une base qui regroupe l'ensemble des concepts communément admis par les ADL, tout en offrant les moyens à l'utilisateur de définir d'autres concepts

ou d'intégrer des spécifications établies avec d'autres ADL.

Nous avons consacré cette section à la présentation des langages de description des architectures logicielles au travers de leurs concepts descriptifs, leurs mécanismes opérationnels. Nous avons illustré ensuite les différents niveaux d'abstraction de description d'une architecture logicielle. Nous avons terminé la section en présentant une classification des ADL, en ADL de première et de deuxième générations.

1.5 Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'intègre cette thèse : le domaine des architectures logicielles à base de composants.

Pour cerner la notion d'architecture logicielle nous avons présenté dans la section 1.3, les définitions significatives proposées dans la littérature pour cette notion. Nous avons présenté plus en détail la norme AINSI Std 1471-2000) qui constitue, à notre connaissance, la première norme pour expliciter ce qu'est une architecture logicielle. Nous avons présenté ensuite la définition que nous adoptons tout au long de ce travail et du manuscrit.

Nous avons abordé dans la section 1.4, les langages de description des architectures logicielles (ADL). Nous avons illustré en premier les concepts communément admis par ces ADL : composant, connecteur, configuration et interface, ainsi que leurs descriptions. Nous avons abordé ensuite les mécanismes opérationnels offerts par les ADL : l'instantiation, l'héritage et sous-typage, la traçabilité et le raffinement. Nous avons défini les différents niveaux d'abstraction de description d'une architecture logicielle, le niveau Méta, le niveau Architectural et le niveau Application. Nous avons terminé cette section en présentant une classification de ces ADL en ADL de première génération et ADL de deuxième génération, selon la spécificité ou non d'un ADL à un domaine particulier.

Nous avons voulu illustrer dans ce chapitre toutes les définitions et notions que nous jugeons nécessaires pour aborder la problématique de l'évolution dans les architectures logicielles. Cette problématique fait l'objet du chapitre suivant.

CHAPITRE 2

Problématique de l'évolution dans les architectures logicielles : État de l'art

2.1 Introduction

Après avoir présenté dans le chapitre précédent les définitions des notions et des concepts relatifs aux architectures logicielles à base de composants ainsi qu'à leurs langages de description, nous abordons dans ce chapitre la problématique de l'évolution dans ces architectures logicielles. Nous nous intéressons plus spécifiquement à la problématique de l'évolution structurelle des architectures logicielles à base de composants. L'évolution structurelle d'une architecture logicielle se reflète par les différents changements dans sa structure et/ou dans celle de ses éléments constitutifs. Un changement dans une architecture est toujours engendré par une ou plusieurs opérations appliquées à cette architecture ou à l'un de ses éléments, telles que la suppression, l'ajout, la modification. Nous désignons ces opérations par *opérations d'évolution* [70, 71].

Il est à noter qu'actuellement la majorité des efforts de recherche dans le domaine des architectures logicielles se focalisent sur la spécification, le développement et le déploiement des architectures. Peu de travaux, à notre connaissance, se consacrent à l'analyse et la gestion de l'évolution de ces architectures logicielles. La plupart des travaux existants se limitent à des mécanismes offerts par l'ADL lui même, mais qui restent souvent influencés par le langage d'implémentation support et les techniques qu'il propose tels que le sous-typage et l'héritage [55].

Le but de ce chapitre est d'étudier les différentes approches proposées par les principaux ADL, pour faire face à la problématique de l'évolution structurelle des architectures logicielles. Nous identifions au préalable dans la section 2.2 un ensemble de critères pour pouvoir caractériser et comparer les approches d'évolution d'architectures logicielles, que nous dénommons *dimensions de l'évolution architecturale*. Nous considérons que cette manière d'aborder l'évolution des architectures logicielles a pour principal avantage de mettre en avant les caractéristiques de l'évolution architecturale, que nous jugeons importantes à prendre en compte pour toute problématique d'évolution et pour toute approche de gestion d'évolution. Ces caractéristiques formeront ainsi un référentiel de l'évolution architecturale permettant de positionner une approche d'évolution de manière indépendante et non par rapport à une autre approche d'évolution. En se basant sur ces dimensions, nous présentons, dans la section 2.3, les approches d'évolution proposées par les principaux ADL. Nous distinguons les approches d'évolution des ADL

de première génération des approches d'évolution des ADL de deuxième génération. Nous présentons ensuite une analyse globale de ces approches d'évolution par rapport aux dimensions de l'évolution identifiées. Nous nous appuyons sur cette analyse pour établir dans la section 2.4 un bilan récapitulatif. Ce dernier souligne à la fois les limites des approches d'évolutions étudiées ainsi que leurs points forts que nous jugeons nécessaires à prendre en compte par toute nouvelle approche d'évolution d'architectures logicielles.

2.2 Dimensions de l'évolution Architecturale

L'évolution structurelle d'une architecture logicielle peut être réalisée à l'étape de spécification ou de conception du système qu'elle décrit (évolution statique) ou au cours de l'exécution de ce système (évolution dynamique). Dans chacun de ces cas, il faut considérer que tout élément de l'architecture logicielle peut être amené à évoluer. Ainsi, il faut identifier ce qui peut évoluer, comment le faire évoluer, comment garantir la cohérence de l'architecture ayant évolué et ensuite comment répercuter ces évolutions de l'architecture logicielle sur le système qu'elle décrit. De manière générale, pour préparer un système ou une architecture à l'évolution, il est nécessaire de pouvoir [64] :

a. Formuler les changements afin d'atteindre l'objectif visé (le modèle d'arrivée à savoir le modèle ayant évolué). Autrement dit pouvoir spécifier le besoin d'évolution.

b. Gérer l'impact engendré par ces changements.

c. Établir le lien entre le modèle de départ et le modèle d'arrivée.

Plutôt que de classifier quelques approches d'évolution d'architectures logicielles en prenant comme repère des critères qui leur sont communs mais qui ne sont pas forcément exhaustifs, nous proposons une classification reposant sur des critères que nous considérons comme étant propre à l'évolution. Cette classification présente un double avantage : celui de cerner les aspects traités de ceux non ou peu traités de l'évolution par les ADL, et celui de pouvoir positionner les approches d'évolution existantes par rapport à un même référentiel. Nous nous sommes inspirés notamment de la taxonomie établie dans [85] pour l'évolution des modèles à base objets et d'une taxonomie proposé dans [57] pour caractériser l'évolution logicielle en générale.

2.2.1 Les trois dimensions de l'évolution architecturale

Les caractéristiques qui nous paraissent importantes pour aborder l'évolution dans les architectures logicielles sont introduites sous forme d'interrogations :

- Sur quoi porte l'évolution, autrement dit, quels éléments de l'architecture peuvent être objet de l'évolution : une configuration, un composant, un connecteur, une interface, etc. ?
- A quel moment du cycle de vie du système est réalisée l'évolution ? au cours de la spécification et/ou conception du système (évolution statique) ou durant l'exécution du système (évolution dynamique) ?
- Comment est exprimée et gérée l'évolution ? autrement dit, le support de l'évolution est-il, implicite, prédéfini ou explicite ?

De ces trois questions fondamentales, il nous apparaît que les trois principales caractéristiques devant être explicitées par une approche d'évolution d'architecture logicielle sont :

- L'objet de l'évolution : quel élément évolue ?
- Le type de l'évolution : de manière statique ou dynamique ?
- Le support de l'évolution : comment est exprimée et gérée l'évolution ?

Nous dénommons ces trois caractéristiques *dimensions de l'évolution architecturale*. Nous les définissons comme suit.

2.2.1.1 Objet de l'évolution

Cette dimension précise les éléments architecturaux sur lesquels porte l'évolution. L'objet de l'évolution peut être tout élément réifié et considéré comme entité de première classe par l'ADL de l'architecture logicielle à faire évoluer. Ainsi, par rapport aux concepts de base de description d'une architecture logicielle que nous avons identifié (section 1.4.1 du chapitre 1), une évolution peut porter sur une *configuration*, un *composant*, un *connecteur*, et/ou sur une *interface*. Une évolution peut concerner les **types** de ces éléments et/ou leurs **instances**, autrement dit elle peut concerner le *niveau Architectural* ou le *niveau Application.*

2.2.1.2 Type de l'évolution

L'évolution d'une architecture logicielle peut être réalisée soit durant :

- la phase de spécification et/ou de conception du système qu'elle décrit. Nous parlerons alors de l'évolution *statique*.
- la phase d'exécution du système qu'elle décrit, nous parlerons alors de l'évolution *dynamique*. Les approches d'évolution dynamique doivent alors garantir la répercussion des évolutions de l'architecture logicielle directement sur l'implémentation du système au cours de son exécution. Nous distinguons deux cas figures, l'évolution dynamique *anticipée* et l'évolution dynamique *non anticipée*.
  - l'évolution anticipée : nous parlerons d'évolution anticipée ou prévue si les besoins d'évolution sont identifiés et cernés dès le début. Ils sont alors spécifiés et pris en compte lors de la phase de conception ;
  - l'évolution non anticipée : nous parlerons d'évolution non anticipée, si l'évolution peut être assurée pour des besoins nouveaux qui apparaissent de façon imprévue durant le cycle de vie du système.

2.2.1.3 Support de l'évolution

Le support d'évolution représente le moyen (opérations d'évolutions et mécanismes) proposé pour exprimer et gérer l'évolution d'une architecture logicielle. Nous considérons qu'il existe trois cas de figures :

- Le support d'évolution implicite : nous considérons que le support d'évolution est implicite, si aucune opération d'évolution ni mécanisme ne sont offerts pour faire évoluer une architecture

logicielle. Si un besoin d'évolution est apparu, alors le concepteur soit, ne sera pas capable de répondre à ce besoin, soit il doit identifier d'une façon adhoc le moyen de réaliser cette évolution.

- Le support d'évolution prédéfini : nous considérons le support d'évolution comme étant prédéfini, s'il est limité à un ensemble d'opérations d'évolutions et/ou de mécanismes bien précis. Il n'existe alors aucun autre moyen d'exprimer une évolution sur une architecture logicielle en dehors de ceux prédéfinis.
- Le support d'évolution explicite : nous considérons que le support d'évolution comme étant explicite, s'il offre le moyen d'exprimer toute évolution que l'on souhaite appliquer sur l'architecture logicielle. Ceci implique que le support d'évolution précise un ensemble d'opérations d'évolution sur l'architecture mais également le moyen de combiner ces opérations pour exprimer toute autre évolution sur une architecture logicielle.

Nous considérons que dans le cas où le support d'évolution est prédéfini ou explicite, l'évolution peut être avec **rupture** ou **continue** :

– une évolution est dite avec **rupture** si la condition c. de la section 2.2 n'est pas vérifiée, c'est à dire que le lien entre l'architecture de départ et l'architecture d'arrivée (ayant évolué) ne peut pas être établi.

– Une évolution est dite **continue** si au contraire, la condition c. est vérifiée et assurée, c'est à dire que le lien peut être établi entre l'architecture de départ et l'architecture d'arrivée.

Nous illustrons les trois dimensions de l'évolution d'architecture logicielle par la représentation triptyque suivante :

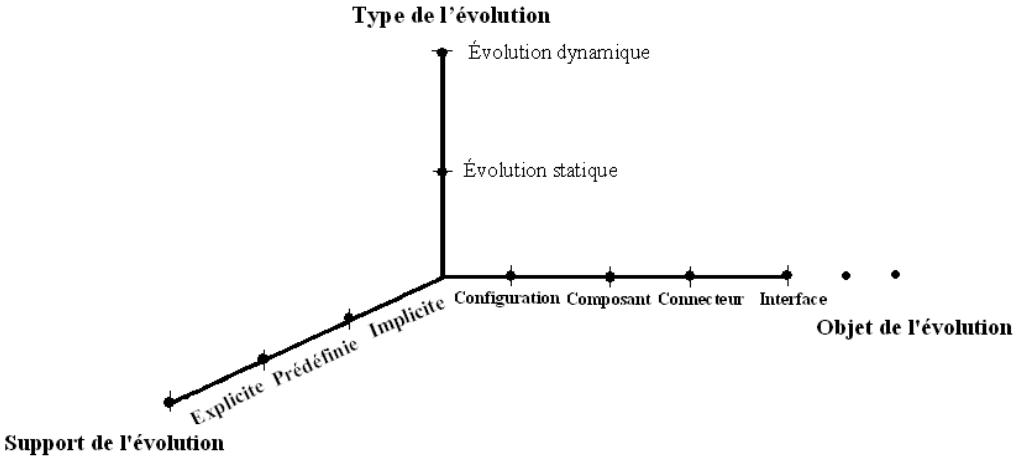

Figure 2.1 – Les trois dimensions de l'évolution architecturale

En considérant ainsi ces trois dimensions, une évolution peut être caractérisée par un triplet qui indique : l'élément architectural sur lequel porte cette évolution, le type de l'évolution considérée pour cet élément et le support de cette évolution. Ainsi par exemple, une évolution peut concerner un *composant type*, peut être réalisée *statiquement* et elle peut être exprimée et gérer *explicitement*.

Cette représentation triptyque peut être exploitée pour positionner soit une problématique d'évolution ou une approche d'évolution. Dans le premier cas, cette représentation servira à spécifier le besoin d'évolution, en précisant les caractéristiques que l'on souhaite associer à une évolution architecturale. Dans le second cas, elle servira à positionner et à comparer les approches d'évolution d'architectures logicielles. Nous illustrons dans la section suivante le second cas.

2.2.2 Comment positionner une approche d'évolution ?

Pour classer les approches d'évolution proposées dans les ADL, il suffit de répondre pour chacune d'entre elles aux trois questions suivantes :

1- Quels sont les éléments architecturaux considérés comme susceptibles d'évoluer ?

2- Pour chaque élément architectural, quel type d'évolution est considérée : évolution statique ou dynamique ?

3- Pour chaque élément architectural et pour chaque type d'évolution, le support d'évolution est-il explicite, implicite ou prédéfini ?

En se basant sur ces interrogations nous présentons dans la section suivante des approches d'évolution d'architectures logicielles au travers un ensemble d'ADL.

2.3 Évolution architecturale dans les ADL

Nous illustrons dans cette section la prise en compte de l'évolution d'architectures logicielles dans les ADL. Nous rappelons que l'évolution n'est pas abordée comme étant une problématique à part entière dans ces ADL. Notre premier travail a consisté en effet, à étudier un ensemble d'ADL pour pouvoir identifier leurs approches liées à l'évolution. Nous présentons dans ce qui suit les différentes approches d'évolution que nous avons identifié de l'études de ces ADL. Nous distinguons les approches d'évolution des ADL de première génération de celles des ADL de deuxième génération. Nous rappelons que la distinction en ADL de première et deuxième générations est établie selon que l'ADL est spécifique à un domaine bien particulier, ou à sa capacité de couvrir plusieurs domaines (cf. section 1.4.4 du chapitre 1). Nous verrons plus loin que l'intérêt de ces deux catégories d'ADL pour l'évolution est différent. Pour la présentation de chaque approche d'évolution nous apportons :

- Une présentation de l'ADL auquel elle est associée, au travers notamment des principaux concepts structuraux de cet ADL.
- Le positionnement de cette approche sur les trois dimensions de l'évolution, en répondant à chacune des questions de la section 2.2.2. Nous illustrons ce positionnement sous forme d'un tableau suivi d'une interprétation.
- Une évaluation de chaque approche d'évolution, en distinguant les points positifs en faveur de l'évolution des points négatifs ou non traités. Ces points seront respectivement précédés de (+) et de (-).

2.3.1 Évolution architecturale dans les ADL de première génération

Nous présentons dans cette partie les approches d'évolution proposées par les ADL : Wright [4, 2], Darwin [49, 48], C2SADEL [53]. Wright est dédiés aux systèmes parallèles communicants, Darwin et C2SADEL aux systèmes distribués. Wright et Darwin sont les plus référencés dans la littérature, notamment pour leur apport sur l'évolution dynamique des architectures logicielles. L'ADL C2SADEL est le premier ADL qui a abordé l'évolution statique dans les architectures logicielles.

2.3.1.1 Évolution architecturale dans Wright

a. Présentation de Wright

L'ADL Wright [4] est construit autour de trois principaux concepts architecturaux : les *composants*, les *connecteurs* et les *configurations*. Un *composant* est décrit par une *interface* et une *partie calcul*. Une interface est composée de *ports*, et chaque port représente une interaction dans laquelle le composant peut participer. Un *connecteur* représente une interaction entre une collection de composants. La description des connecteurs se résume à un ensemble de rôles et de glus. Chaque *rôle* indique comment se comporte un composant qui participe à l'interaction. La glu d'un connecteur décrit comment les participants travaillent ensemble pour créer une interaction. Une *configuration* est une collection d'instances de composants liés par des instances de connecteurs.

b. Positionnement de Wright sur les trois dimensions de l'évolution

Le tableau suivant positionne l'approche d'évolution proposée dans Wright par rapport aux trois dimensions d'évolution d'architecture logicielle identifiées.

Objet de l'évolution	***Type de l'évolution***	***Support de l'évolution***
Composant type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
Connecteur type		**Explicite** (Évolution avec rupture) - Définition des contraintes
Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contraintes
	Dynamique	**Prédéfini** (Évolution anticipée et avec rupture) Modification configuration par : - Ajout/ suppression composant et/ou connecteur - Ajout/ suppression attachements (attach et detach)
Composant	**Dynamique**	**Prédéfini** (Évolution anticipée et avec rupture) - Création/ suppression d'un composant
Connecteur	**Dynamique**	**Prédéfini** (Évolution anticipée et avec rupture) - Création/ suppression d'un connecteur

Table 2.1 – Positionnement de Wright sur les trois dimensions de l'évolution

- Wright supporte l'évolution *statique* des *composants* et des *connecteurs types*, en offrant le mécanisme de *composition hiérarchique*. En effet un composant type ou un connecteur type peut évoluer d'un niveau de détail élevé vers un niveau plus détaillé. Le détail d'un composant ou d'un connecteur type est décrit par une configuration.
- Wright offre aussi la possibilité de définir des *contraintes* relatives aux *composants* et *connecteurs types* et aux *configurations*. Son langage de contraintes est basé sur le langage de spécification formelle Z [81]. Grâce à ce langage, Wright décrit un ensemble d'opérateurs, des ensembles et de prédicats pour l'architecture logicielle.
- Initialement Wright n'a pas été conçu pour supporter l'évolution dynamique de l'architecture logicielle, Dynamic Wright [3] a été ensuite proposé pour combler ce manque. Dynamic Wright étend Wright pour supporter des manipulations pour les changements dynamiques de la configuration d'une architecture. Ces changements doivent être *connus* et *prévus* au moment de la spécification de l'architecture. Pour prendre en compte cette dynamique deux extensions ont été apportés à Wright :
  - L'ajout des événements de contrôle à la spécification des composants, au niveau de leurs interfaces. Ces événements de contrôle sont explicités dans la spécification des composants par le mot clé ≪ contrôl ≫.
  - L'ajout d'un composant particulier le configurateur ≪ Configurator ≫. La gestion et le contrôle de la dynamique d'une architecture sont alors centralisés au niveau de ce composant. Le configurateur décrit un ensemble de règles de reconfiguration à l'aide d'un langage ad hoc basée sur la notion d'actions et de règles. Les règles définissent les actions à effectuer en fonction de la réception de certains événements. Les actions quant à elles peuvent être la *création* ou la *destruction* des composants ou des connecteurs tel que : *new*, *attach*, *detach*. Un exemple de spécification d'évolution dynamique est donné dans l'annexe A.

Toutes les évolutions considérées dans Wright sont avec *rupture*, le lien entre l'architecture de départ et l'architecture d'arrivée (architecture ayant évolué) ne peut être établi.

c. Evaluation

(+) Wright est parmi les premiers ADL qui se sont intéressés à l'évolution dynamique des architectures logicielles ;

(+) La spécification de l'évolution dynamique est centralisée au niveau du configurateur. Le lien avec la spécification de l'architecture est établi grâce aux événements de contrôle ;

(+) La spécification des contraintes dans Wright permet de vérifier la cohérence de l'architecture après évolution ;

(-) L'évolution statique de l'architecture est peu abordée dans Wright. Aucune opération n'est proposée pour faire évoluer les composants et connecteurs types en dehors de la *composition hiérarchique* ;

(-) L'évolution dynamique considérée est anticipée. Dans Dynamic Wright, aucune autre évolution ne peut être réalisée durant l'exécution du système, en dehors de celles préalablement prévues et spécifiées dans l'architecture ;

(-) Les opérations spécifiées dans le *configurateur* concerne uniquement la modification d'une configuration. Il n' y a aucun moyen de modifier les *composants* et *connecteurs* types ainsi que leurs instances.

2.3.1.2 Évolution architecturale dans Darwin

a. Présentation de Darwin

Darwin [49, 48] est défini pour supporter l'analyse des transmissions de messages dans les systèmes parallèles distribués. Les principales abstractions gérées par Darwin sont les composants. Un composant type est décrit par une interface composée d'une collection de services qui sont fournis ou requis. Les connecteurs quant à eux ne sont pas considérés comme des entités de première classe. Chaque interaction est représentée par un lien entre un service requis et un service fourni de composants différents. Les configurations (composants composites) sont décrites par les déclarations d'instanciation des composants et par les liaisons entre les services requis et les services fournis de ces instances de composants.

b. Positionnement de Darwin sur les trois dimensions de l'évolution

Le tableau suivant positionne l'approche d'évolution proposée dans Darwin par rapport aux trois dimensions de l'évolution architecturale.

Objet de l'évolution	***Type de l'évolution***	***Support de l'évolution***
Composant Type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
Configuration	**Dynamique**	**Prédéfinie** (Évolution anticipée et avec rupture) Modification configuration par : - Instantiations dynamique et paresseuse d'un composant - Ajout d'un binding (bind)

Table 2.2 – Positionnement de Darwin sur les trois dimensions de l'évolution

- Darwin supporte le mécanisme de la *composition hiérarchique*. Un *composant* composite peut être décrit par une configuration interne.
- Une caractéristique importante de Darwin est sa capacité à décrire la création dynamique de composants par l'application. Ainsi, la configuration de l'application n'est plus considérée comme un ensemble de composants prévus dès la phase de conception, mais elle peut évoluer au cours de l'exécution du système. Darwin fournit deux principaux mécanismes pour la description de structures dynamiques l'instanciation paresseuse et l'instanciation dynamique :
  - L'instanciation paresseuse est une pré-déclaration des instances qui seront effectivement créés non pas lors de la phase d'initialisation, mais dès qu'un premier appel vers l'instance est effectué (cf. Annexe A).
  - L'instanciation dynamique permet de décrire la création d'un composant en lui fournissant des paramètres d'initialisation. La création dynamique d'un composant est réalisée au sein d'un composant composite et il est impossible d'accéder à ces composants depuis l'extérieur du composite.

Toutes les évolutions spécifiées dans Darwin sont avec *rupture*.

c. Evaluation

(+) Darwin permet la création des composants dynamiquement en offrant une syntaxe riche pour la spécification de cette dynamique ;

(-) La spécification des évolutions dynamiques sont imbriquées dans la spécification de l'architecture elle-même ce qui aboutit à des spécifications très complexes et difficiles à comprendre ;

(-) Toutes les évolutions qui doivent opérer dynamiquement doivent être connues au préalable. Cela exige, par avance, d'énumérer toutes les instances des configurations possibles de l'architecture. Aucune opération spécifique n'est proposée par Darwin pour faire évoluer les composants au delà de ce qui est prévu à la spécification de l'architecture ;

(-) Il n'est pas possible de supprimer des composants dynamiquement ;

2.3.1.3 Évolution architecturale dans C2SADEL

a. Présentation de C2SADEL

C2SADEL (Software Architecture Description and Evolution Language) [53] est l'un des ADL dont l'objectif est de décrire des architectures de systèmes distribués, évolutifs et dynamiques. Il s'appuit sur les règles du style C2 [54]. Les briques de base de C2SADEL sont aussi les *composants*, les *connecteurs*, et la *topologie* (configuration). Les composants et les connecteurs sont vus comme des types qui peuvent être instanciés plusieurs fois au sein d'une topologie. Un *composant* est défini par un nom, une interface composée d'éléments (un *top_port* ou/et *bottom_port*), un comportement et/ou une ou plusieurs implémentations. Les *connecteurs* sont définis aussi explicitement, leur principale tâche est de coordonner la communication entre les composants. Dans le style C2, un nombre quelconque de composants peut être attaché à un même connecteur. Ainsi, l'interface d'un connecteur est définie en fonction des interfaces des composants qui communiquent à travers elle. Le seul aspect modélisé des connecteurs est le mécanisme de filtrage des informations qui le traverse dénoté par le mot clé *message_filter*. La *topologie*, nommée dans la spécification par *Architecture*, désigne la connexion entres les instances de composants et de connecteurs.

b. Positionnement sur les trois dimensions de l'évolution

Le tableau suivant positionne l'approche d'évolution proposée dans C2SADEL par rapport aux trois dimensions d'évolution d'une architecture logicielle.

Objet de l'évolution	*Type de l'évolution*	*Support de l'évolution*
Composant Type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique - Sous-typage de *nom*, d'*interface* de *comportement* et d'*implémentation*
		Explicite (Évolution avec rupture) - Définition des invariants
Connecteur type	**Statique**	**Prédéfini** (Évolution avec rupture) - Modification du mode de filtrage du connecteur
Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Définitions des invariants
	Dynamique	**Prédéfini** (Évolution non anticipée et avec rupture) Modification configuration par : - Ajout/suppression/remplacement composant - Ajout/suppression/remplacement connecteur - Ajout/suppression d'attachement (*Weld*, *unweld*)
Composant	**Dynamique**	**Prédéfini** (Évolution non anticipée et avec rupture) - Création /suppression composant
Connecteur	**Dynamique**	**Prédéfini** (Évolution non anticipée et avec rupture) - Création /suppression d'un connecteur - Modification connecteur par : Ajout/ suppression d'interface Modification du mode de filtrage du connecteur

Table 2.3 – Positionnement de C2SADEL sur les trois dimensions de l'évolution

- C2SADEL supporte le mécanisme de *composition hiérarchique* pour l'évolution des *composants types*. Un Composant composite peut être décrit hiérarchiquement par une description interne (configuration).
- C2SADEL propose aussi le mécanisme de *sous-typage* pour l'évolution des *composants types*. En se basant sur la définition donnée dans la section précédente pour le concept composant, les auteurs dénombrent plusieurs déclinaisons du mécanisme de sous-typage : de *nom*, d'*interface*, de *comportement*, d'*implémentation*. Le sous-typage *hétérogène* est alors la combinaison de ces sous-typages par les clauses *and*, *or* et *not*. Des relations de compatibilité sont alors établies pour définir ces différents sous-typages. Nous citons à titre d'exemple la définition de la compatibilité d'interface : un composant *cp1* est considéré comme un sous-type du composant *cp2* par interface, si et seulement si le composant *cp2* spécifie au moins toutes les interfaces fournies et les plus importantes des interfaces requises de *cp1*, que les paramètres et les résultats de chaque élément de l'interface sont compatibles (ils doivent avoir des noms identiques, les paramètres de *cp1* sont des sous types de leurs correspondants paramètres de *cp2*, et le type du résultat de *cp1* est super type du résultat de *cp2*). La compatibilité d'interface est utile par exemple dans le cas où on voudrait remplacer un composant sans affecter les autres composants et connecteurs auxquels il est attaché.
- C2SADEL permet d'exprimer des *invariants* sur les *composants types* et sur les *configurations*. Les invariants sont définis comme une conjonction de prédicats exprimés en logique du premier ordre.

- Les connecteurs du style C2 sont par définition évolutifs, du fait qu'ils n'imposent pas une interface spécifique. Cette propriété est dénommée par *Interface Contexte réflexif*. Elle indique donc que l'interface du connecteur n'est pas fixe mais elle est construite en fonction des composants qui s'attachent et qui se détachent de ce connecteur. Une autre propriété des connecteurs du style C2, qui favorise leur évolution, est la variation du degré d'informations échangées entre les composants. Ceci est supporté par les différents *mécanismes de filtrage* de messages que les connecteurs proposent. Par exemple *sink filtering* qui indique que le connecteur ne tolère aucun échange d'information entre les composants qu'il connecte, ou encore le *no_filtering* qui indique que l'information reçue par le connecteur est diffusée sans aucune restriction à tous les composants qui lui sont attachés. Ainsi, un connecteur peut évoluer, par la modification du mécanisme de filtrage qu'il propose.
- C2SADEL spécifie un ensemble d'opérations pour l'*insertion*, la *suppression*, le *remplacement* et la *re-connexion* des éléments de l'architecture pendant l'exécution du système tel que : *addComponent*, *removeComponent*, *weld* et *unweld*. Ces évolutions sont réalisées d'une façon interactive via l'outil Archstudio1 [60] couplé à l'outil DRADEL (Developement of Robust Architectures using a Description and Evolution Language) [53]. DRADEL est un environnement pour supporter la modélisation, l'analyse et l'évolution des architectures décrites en C2SADEL.

  Toutes les évolutions spécifiées dans C2SADEL sont avec rupture.

c. Evaluation

(+) C2SADEL est le premier langage qui a abordé la problématique de l'évolution statique des architectures logicielles via le sous-typage hétérogène ;

(+) Le langage aborde l'évolution des trois briques de base d'une architecture logicielle : les composants types, les connecteurs types, et la topologie.

(-) L'évolution dynamique de la configuration, se fait via une interface interactive. Ni l'origine de la création ou de la suppression des ces composants, ni les moyens de gestion de ces composants, une fois crées ne sont spécifiés.

(-) Les primitives pour l'ajout, la suppression des composants et connecteurs ou de leur connexion ne précisent pas les impacts de ces opérations sur le reste de l'architecture.

(-) L'approche d'évolution n'est pas générique, elle est complètement dépendante du style architectural C2.

(-) L'ADL ne précise pas l'impact de l'évolution d'un type sur ses instances et inversement.

2.3.2 Evolution architecturale dans les ADL de deuxième génération

Nous présentons dans cette partie les approches d'évolution proposées par les ADL de deuxième génération. Nous rappelons que les ADL de deuxième génération sont des ADL qui ne sont pas spécifiques à un domaine particulier (cf. section 1.4.4 du chapitre 1). Nous avons sélectionné les ADL les plus représentatifs : ACME [32], SafArchie [7, 8], UML2.0 [35] , Mae [38, 68], xADL2.0 [21, 20].

2.3.2.1 Évolution architecturale dans ACME

a. Présentation de ACME

ACME [32] a été présenté par ces créateurs comme un langage d'échange d'architectures. Il représente en effet le plus petit dénominateur commun des ADL existants, plutôt qu'une définition d'un ADL. Dans ACME, la structure architecturale est décrite à l'aide de sept types de concepts : le *composant*, le *connecteur*, le *système*, le *port*, le *rôle*, la *représentation* et la *carte de représentations*. Le *composant* est une entité de calcul ou de stockage de données. Il est décrit par des interfaces composées de *ports* (fournis ou requis). Un composant peut être primitif ou composé. Le *connecteur* décrit une connexion entre les composants. Il est également décrit par des interfaces définies par un ensemble de *rôles*. Un *système* dans ACME décrit la configuration de l'architecture en terme d'instances de composants et de connecteurs reliés par les *attachements* et les *liaisons*. Un attachement relie le port d'un composant à un rôle d'un connecteur. Une liaison relie les ports d'une configuration ou d'un composant composite aux ports de ses composants internes.

b. Positionnement sur les trois dimensions de l'évolution

Le tableau suivant positionne l'approche d'évolution proposée dans ACME par rapport aux trois dimensions d'évolution d'une architecture logicielle.

Objet de l'évolution	***Type de l'évolution***	***Support de l'évolution***
Composant type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique - Sous-typage
Connecteur Type		**Explicite** (Évolution avec rupture) - Définition des contraintes
Configuration	**Dynamique**	**Prédéfini** (Évolution anticipée et avec rupture) via des constructions syntaxiques tel que : - La *multiplicité* d'un composant ou d'un connecteur - *Open* pour indiquer qu'un élément est ouvert

Table 2.4 – Positionnement de ACME sur les trois dimensions de l'évolution

- ACME propose le mécanisme de *composition hiérarchique* pour l'évolution *statique* des *composants* et *connecteurs types*. Ce mécanisme est supporté par ACME, grâce au concept de *représentation*. En effet, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé. Chaque nouvelle description d'un composant ou d'un connecteur est appelée *représentation*. La correspondance entre un élément et ses représentations est spécifiée grâce à la *carte de représentation* (rep-maps). Ainsi la spécification d'un composant ou d'un connecteur peut évoluer d'un niveau de détail élevé vers d'autres niveaux plus raffinés.
- ACME propose le mécanisme de *sous-typage*, pour l'évolution *statique* des *composants* et des *connecteurs types*. En effet, un composant ou un connecteur type peut être crée à partir d'un autre composant existant grâce à la relation du sous-typage. Ceci est explicité dans la description de l'architecture par la clause ≪ extend ≫ (cf. Annexe A).
- ACME permet aussi l'expression des contraintes architecturales, en utilisant le langage de contraintes basé sur la logique du premier ordre FOPL [41]. En plus des différentes constructions standards de

FOPL (conjonction, disjonction, ...), le langage offre d'autres fonctions spéciales qui se rapportent à des aspects spécifiques aux architectures.

- Dynamic ACME [89] propose des extensions à ACME pour gérer l'évolution dynamique des architectures logicielles. En effet Dynamic ACME étend la syntaxe d'ACME par des constructions syntaxiques pour spécifier des évolutions d'une architecture logicielle préalablement prévues et connues (anticipées). Pour se faire, il distingue deux catégories d'éléments architecturaux : les éléments *fermés* qui indique que la spécification de ces éléments est complète et les éléments *ouverts* qui indique que la spécification de ces éléments peut évoluer. Un élément décrit en ACME est implicitement fermé et pour expliciter qu'un élément peut évoluer, le modificateur *open* est ainsi utilisé. Dynamic ACME associe aussi une *multiplicité* à un composant ou un connecteur type qui indique le minimum et le maximum de ses instances.

Toutes les évolutions spécifiées en ACME sont avec rupture, le lien entre l'architecture de départ et l'architecture d'arrivée (ayant évolué) ne peut pas être établi.

c. Evaluation

(+) La spécification d'abord des types de composants et des connecteurs et l'utilisation des contraintes permet à ACME de contraindre l'évolution ainsi que la vérification de la cohérence après évolution ;

(+) L'évolution statique des composants et connecteurs types est considérés dans ACME via notamment la composition et le sous-typage ;

(-) Des exemples dans la littératures [89] montrent que les extensions d'ACME pour la spécification de l'évolution dynamique aboutissent à des spécifications très complexes, difficiles à comprendre ;

(-) ACME ne permet pas de spécifier explicitement des opérations pour la modification de l'architectures ou de l'un de ses éléments.

2.3.2.2 Evolution architecturale dans UML2.0

a. Présentation de UML2.0

UML2.0 [35] introduit la notion de *diagramme d'architecture*, appelée aussi de structure composite. Inspiré des ADL, UML2.0 propose les trois concepts fondamentaux des langages de description d'architectures logicielles : *composant*, *connecteur* et *configuration* (composant composite). Un *composant* est une entité modulaire et réutilisable fournissant et requérant des *interfaces*. Une interface contient un ensemble de méthodes et/ou de contraintes. Un *port* est une entité qui émerge d'un composant, pouvant être requis ou fourni. Un port se comporte comme un point d'interaction d'un composant avec son environnement. La connexion entre les ports requis et fournis se fait au moyen de *connecteurs* (les connecteurs d'assemblage et les connecteurs de délégation). Un connecteur d'assemblage permet d'assembler deux instances de composants en connectant un port fourni d'un composant à un port requis de l'autre composant. Le connecteur de délégation permet de connecter un port d'un composite à un port d'un de ses sous-composant interne.

b. Positionnement sur les trois dimensions de l'évolution

Le tableau suivant positionne les mécanismes d'évolution proposés dans UML2 .0 par rapport aux trois dimensions d'évolution architecture logicielle.

Objet de l'évolution	*Type de l'évolution*	*Support de l'évolution*
Composant type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique **Explicite** (Évolution avec rupture) - Définition des contraintes
Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contraintes

Table 2.5 – Positionnement d'UML2.0 sur les trois dimensions de l'évolution

- UML2.0 supporte pour l'évolution des composants type le mécanisme de *composition hiérarchique* et d'*héritage*. L'introduction du concept *partie* (part) dans UML 2.0 rend possible la description de la partie interne d'une classe. Une partie est donc une propriété de la classe qui a le même cycle de vie que l'objet de la classe à laquelle elle appartient. Les parties sont les instances des autres classes. Comme le concept de composant UML étend celui de la classe, la structure interne d'un composant est elle aussi décrite à l'aide des parties qui peuvent être des instances de classes ou des instances de composants connectées donc par un ensemble de connecteurs. Avec cette même analogie, le mécanisme d'*héritage* peut être utilisé pour faire évoluer les composants. Nous n'avons pas retrouvé dans la littérature des exemples de spécifications considérant le mécanisme d'héritage.
- UML2.0 permet aussi la définition des contraintes architecturales, notamment via son langage de contrainte OCL (Object Constraint Language).
- UML2.0 n'offre pas d'opérations ni de mécanismes pour gérer l'évolution dynamique d'une architecture logicielle.

c. Evaluation

(+) UML 2.0 offre des mécanismes pour contraindre l'évolution via les contraintes OCL ;

(-) UML 2.0 n'a pas considéré explicitement la problématique de l'évolution. Il n'offre pas des opérations permettant de décrire l'évolution d'une architecture, ni de mécanismes pour la gérer que ça soit statiquement ou dynamiquement.

2.3.2.3 Evolution architecturale dans SafArchie

a. Présentation de SafArchie

SafArchie [7, 8] est défini comme un modèle d'architecture logicielle pour l'analyse d'un assemblage de composants d'un point de vue structurel et comportemental. Il se base sur cinq concepts architecturaux : le *composant primitif*, le *composite*, le *port*, l'*opération* et la *liaison*. Le *composant* est une entité logicielle définie par l'intermédiaire de son interface avec l'extérieur en vue de sa composition et de sa

réutilisation. L'interface met en évidence les services que le composant fournit et requiert. La notion de service se définit par un ensemble d'*opérations* ainsi que par leur enchaînement. Chaque service est mis en oeuvre par l'intermédiaire d'un ou plusieurs *ports*. Une *opération* définit les structures de données échangées dans une interaction entre deux composants. Les *liaisons* matérialisent les interactions possibles entre composants. SafArchie est un modèle hiérarchique. Un composant peut contenir d'autres composants. Il est alors défini comme étant un *composite*. Des *liaisons de délégation* permettent alors de relier les ports du composite aux ports de ses composants internes. SafArchie propose la définition d'abord d'une architecture type. Une architecture type est composée d'un type de composite. Celui-ci est composé de type(s) de composants primitifs ou de types de composant (s) composite(s). Les composants types primitifs sont composés de types de ports.

SafArchie favorise la construction itérative de l'architecture à l'aide du canevas **TranSAT** (Transformation for Software Architecture Technics) [7]. Inspiré des travaux autour de la séparation des préoccupations et des aspects [44], [16], TranSAT suggère d'établir la spécification de l'architecture autour de ses fonctionnalités métiers puis d'établir la spécification des préoccupations techniques indépendamment, pour ensuite intégrer et regrouper les deux spécifications. TranSAT est centré autour des concepts du *patron architectural* et du *tisseur*. Le patron architectural regroupe l'ensemble des informations relatives à une préoccupation : le *plan*, *masque de jonction*, *règles de transformations*. Le tisseur permet d'intégrer le plan correspondant à la nouvelle préoccupation dans une architecture de base. Une description plus détaillée de TranSAT est donnée dans l'annexe A.

b. Positionnement sur les trois dimensions

Le tableau suivant positionne les mécanismes d'évolution proposés dans SafArchie (+ TranSAT) par rapport aux trois dimensions d'évolution architecture logicielle.

Objet de l'évolution	***Type de l'évolution***	***Support de l'évolution***
Composant Type	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
Interface type	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contrats
Composant primitif	**Statique**	**Prédéfini** (Évolution avec rupture) Modifier interface composant
Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Ajouter/supprimer liaison - Ajouter/supprimer composant - Ajouter/supprimer un composite
Interface	**Statique**	**Explicite** (Evolution avec rupture) Modifier interface par - Ajout d'un port et /ou d'une opération - Suppression d'un port et/ou d'une opération - Déplacement d'une opération

Table 2.6 – Positionnement de SafArchie sur les trois dimensions d'évolution

- Pour l'évolution des *composants types*, SafArchie offre le mécanisme de *composition hiérarchique*, ainsi un composant type peut être décrit par un ensemble d'autres composants.

- Pour appuyer la vérification de la compatibilité, le modèle de type de SafArchie est enrichi avec des *contrats* [58] tels que les *contrats d'assertion* et les *contrats de ports*. Le *contrat de ports* par exemple permet de préciser les dépendances entre les opérations d'un type de port d'un composant.
- L'outil TranSAT permet la construction incrémentale de l'architecture. Il se base sur un ensemble de règles de transformation décrites par un ensemble de *primitives*. Ces primitives sont scindées en deux, les *primitives d'introduction* et les *primitives de reconfiguration*.
  - Primitives d'introduction : elles sont utilisées pour changer localement la structure d'un composant. Elles permettent de *modifier* l'interface d'un composant, en lui *ajoutant* de nouveaux ports et aussi de nouvelles opérations, mais aussi de *supprimer* des ports ou encore de *déplacer* des opérations.
  - Primitives reconfiguration : ces primitives prennent en charge les modifications des interactions entre composants lors de l'intégration d'une nouvelle préoccupation. Ces modifications concernent la *création* et la *suppression* des connexions, la *création* et la *suppression* de composites et l'*ajout* et le *retrait* de composants au sein d'un composite.

Toutes les évolutions spécifiées avec SafArchie sont avec rupture.

c. Evaluation

(+) La spécification de l'évolution est indépendante de la spécification de l'architecture elle-même. Elle est définie indépendamment dans le *patron architectural* ;

(+) Le modèle de types et la spécification des contrats permettent de contraindre l'évolution, contribuant ainsi à la sauvegarde de la cohérence de l'architecture ;

(+) SafArchie permet la spécification hiérarchique des composants ;

(-) Lors de l'intégration de nouveaux composants ou d'une nouvelle préoccupation, la recherche de points de compatibilité est une tâche assez difficile et peu être assez lente, notamment que cette tâche est peu assistée par TranSAT ;

(-) L'approche se préoccupe de l'évolution de l'architecture par ajout de nouvelles préoccupations, mais elle ne précise pas comment spécifier l'évolution de l'architecture ainsi que son impact en supprimant un ou plusieurs composants.

(-) TranSAT est conçu spécifiquement pour le modèle architectural SafArchie, il ne peut être réutilisé dans le cadre d'autres modèles architecturaux ;

(-) Même si TranSAT travaille au niveau logique (sur les instances de composants), l'approche ne précise pas comment la modification du système est opérée dynamiquement, autrement dit comment les évolutions peuvent être considérées au niveau du système en cours d'exécution.

(-) Lors de l'évolution au niveau des instances, les impacts sur le niveau des types ne sont pas précisés, c'est à dire si l'interface d'une instance d'un composant est modifiée, un nouveau type par exemple n'est pas automatiquement crée.

2.3.2.4 Evolution architecturale dans Mae

a. Présentation de Mae

Mae est un environnement pour la spécification et la gestion de l'évolution des architectures logicielles [38, 68]. Mae combine les concepts architecturaux communément admis avec les concepts du domaine de la gestion des configurations "configuration management" [13, 14]. Les concepts de base de Mae sont les concepts communément admis par la majorité des ADL : *composant* (type et instance), *connecteur* (type et instance), *interface* (type et instance) et *configuration*. Le *composant type* est la brique de base du modèle architectural. Il fournit et requiert des services via son interface. Un composant type est décrit aussi par un comportement et un ensemble de contraintes. Un composant type peut être instanticié plusieurs fois au sein d'une architecture. Un *connecteur type* a la même structure qu'un composant type, c'est la sémantique et leur rôle dans l'architecture qui diffère. Un connecteur peut être instancié aussi plusieurs fois dans une architecture. L'*interface type* décrit les services requis ou fournis d'un composant type ou d'un connecteur type. Elle peut être spécifiée via un ensemble d'éléments (*interfaceElements*), qui détaille la signature des méthodes exposées par le composant ou le connecteur type. Une instance d'une interface type est définie par un nom qui la distingue des autres instances d'interfaces et sa *direction* (in, out, in/out). Une configuration représente un ensemble d'instances de composants reliés par un ensemble d'instances de connecteurs.

b. Positionnement de Mae sur les trois dimensions de l'évolution

Le tableau 2.7 positionne l'approche d'évolution proposée dans Mae par rapport aux trois dimensions d'évolution d'architecture logicielle.

<table>
<tr><th>Objet de l'évolution</th><th>Type de l'évolution</th><th>Support de l'évolution</th></tr>
<tr><td>Composant type</td><td rowspan="2">Statique</td><td>Prédéfini (Évolution Continue)
- La composition hiérarchique ;
- Sous-typage hétérogène ;
- Création de versions ;
- Création de composants/ connecteurs variants.</td></tr>
<tr><td>Connecteur type</td><td>Explicite (évolution continue)
- Utilisation des gardes booléennes
- Définition des contraintes</td></tr>
<tr><td>Interface Type</td><td>Statique</td><td>Prédéfini (Évolution continue)
- Création de la version d'une interface</td></tr>
<tr><td>Configuration</td><td>Statique</td><td>Prédéfini (Évolution continue)
Modifier de la configuration par :
- Ajout, suppression ou remplacement d'une version d'un composant, d'un connecteur ou d'une interface par une autre.</td></tr>
<tr><td>Composant</td><td rowspan="2">Statique</td><td rowspan="2">Explicite (Évolution continue)
Rendre un un composant ou un connecteur optionnel via les gardes booléennes</td></tr>
<tr><td>Connecteur</td></tr>
</table>

Table 2.7 – Positionnement de Mae sur les trois dimensions d'évolution

- Inspiré de C2SADEL, Mae utilise le mécanisme de *composition hiérarchique* et le *sous-typage hétérogène* (cf. section 2.3.1.3) pour l'évolution des composants et des connecteurs types.
  - Mae permet la *composition hiérarchique* en permettant de décrire la structure interne d'un composant ou d'un connecteur en terme d'instances d'autres composants et connecteurs. La clause *InterfaceMapping* permet de définir la correspondance entre les interfaces du composite et les interfaces des instances de composants et de connecteurs qui le composent.
  - Le *sous-typage* utilisé dans Mae est hétérogène, un composant/connecteur type peut être crée en préservant une ou plusieurs propriétés d'un autre composant/connecteur (nom, interface, comportement).

  En plus de ces mécanismes, Mae emprunte d'autres notions issues de la technique de *gestion des configurations* pour l'évolution des éléments architecturaux. Pour capturer l'évolution des éléments architecturaux, Mae a emprunté les notions suivantes :

  - *Variabilité* : un composant (respectivement un connecteur) type *variant* est défini en terme de plusieurs autres composants (respectivement connecteurs) types, auxquels des gardes booléennes exclusives sont associées. A la construction de la configuration, c'est l'élément dont la garde est évaluée à vrai qui sera intégré. Ce concept est introduit pour exprimer le fait qu'à certains endroits de l'architecture, un ou plusieurs composants ou connecteurs peuvent être intégrés alternativement.
  - *Optionalité* : certains composants ou connecteurs peuvent être optionnels dans une configuration et ne sont intégrés que dans certaines conditions. Pour exprimer qu'un composant ou un connecteur est optionnel, une garde booléenne lui est associée au moment de son instantiation.
  - *Revision* : le versionnement est utilisé dans Mae pour capturer les différentes incarnations d'un élément de l'architecture (composant, connecteur et interface types). Un numéro est alors associé à chaque élément de l'architecture via le champ *Revision*. Chaque évolution d'un élément donne lieu à l'incrémentation de sa révision. Les champs *Ascendant/descendant* permettent de relier les différentes évolutions d'un type. Le champ *subtype* permet de spécifier ce qu'un composant/connecteur types préserve de son prédécesseur.
  - *Inter-file branches* : dans certains cas, à partir d'un même élément, deux ou plusieurs chemins divergeant d'évolution sont possibles. Chaque chemin est formé en créant un nouveau type qui aura son propre nom et qui peut avoir ensuite sa propre évolution. Les champs *Ascendant/descendant* et *subtype* sont aussi utilisés comme dans le cas d'une révision, pour décrire les liens entre les composants/connecteurs types d'une branche.

Mae permet ainsi de capter, de relier et de garder les traces des différentes évolutions d'une architecture. Mae considère la technique de *check out/check in*, ainsi, l'élément à faire évoluer est d'abord isolé, les modifications lui sont apportées, puis il est intégré dans le système sous forme d'une nouvelle version.

L'évolution spécifiée dans Mae est *continue*. Le lien entre l'architecture de départ et l'architecture d'arrivée peut être établi. L'ensemble des versions d'un élément architectural sont reliées entre elles et il est toujours possible de revenir à une ancienne version d'un élément architectural. L'évolution est réalisée dans Mae d'une façon interactive les auteurs ne précisent pas de moyens pour répercuter les évolutions de l'architecture logicielle sur le code source.

Des outils récents ont tenté de répondre à cette préoccupation, tel que ArchEvol (Architecture-Based Software Evolution)[59]. ArchEvol propose un environnement conçu à partir de trois outils existants :

Archstudio 3 [29], Eclipse (environnement de développement Java) [28] et Subversion [17] un système de gestion de configurations. Le but principal de cet outil est de maintenir la cohérence entre la description architecturale d'un système et son implémentation (le code source) au fur et à mesure de leurs évolutions. En partageant l'outil ArchStudio, ArchEvol peut être couplé avec Mae pour répondre à l'évolution dynamique des architectures logicielles.

c. Evaluation

(+) L'exploitation des concepts de la *gestion de configuration* a rendu Mae l'une des premières approches permettant de garder la trace des évolutions d'une architecture logicielle (évolution explicite continue). Cette trace est importante et indispensable dans certains systèmes tel que les systèmes bancaires ;

(+) Permet la spécification *hiérarchique* des composants et des connecteurs et exploite le mécanisme du *sous-typage hétérogène* pour l'évolution des composants et des connecteurs ;

(+) L'approche définie un cadre pour contraindre l'évolution notamment via le typage, les contraintes et les gardes booléennes ;

(+) L'approche dispose des supports pour assister le concepteur dans la spécification et l'évolution d'une architecture ;

(-) La propagation des impacts n'est pas traitée explicitement, par exemple, à l'évolution d'un composant vers une autre version ou à la suppression d'un composant, l'impact vers les autres éléments ne peut pas être défini automatiquement. Toute incohérence n'est alors détectée qu'au niveau de l'analyse de toute la configuration ;

(-) Mae ne précise pas des opérations d'évolution explicites permettant de passer d'une version d'un composant ou d'un connecteur type vers une autre version ;

(-) Les évolutions au niveau architectural ne sont pas reflétées automatiquement sur le système au cours d'execution.

2.3.2.5 L'évolution architecturale dans xADL 2.0

a. Présentation de xADL 2.0

xADL 2.0 [21, 20, 43] a été conçu comme un langage de description flexible et extensible pour les architectures logicielles. En effet, xADL 2.0 est construit comme un ensemble extensible de schémas XML [12], autour de xArch [40]. xArch est un standard pour la représentation des architectures logicielles basé sur des schémas XML, et dont le noyau est basé sur les concepts communément admis par la majorité des ADL. A partir de cet ensemble de base de concepts, le concepteur peut alors étendre l'ADL par d'autres concepts pour prendre en compte ses besoins particuliers. xADL2.0 offre plusieurs schémas pour la spécification architecturale. Chaque schéma correspond à un domaine de préoccupation particulier (conception architecturale, correspondance vers l'implémentation, l'implémentation, etc.). Nous présentons à titre d'illustration dans ce qui suit les concepts du schéma de conception.

Le schéma de conception architecturale (*schéma Types et Structures*) sépare la description architecturale en deux types de blocs : *ArchTypes* et *ArchStructure*. *ArchTypes* décrit les **types** de *composants*, de *connecteurs* et d'*interfaces*. *ArchStructure* décrit les *composants*, les *connecteurs*, les *liens*, les *groupes* et les *SubArchitectures* (une subArchitecture décrit la topologie du système ou la structure interne d'un

composant ou d'un connecteur composite). Chaque élément de ArchStructure fait référence à un type décrit dans le bloc ArchTypes.

b. Positionnement sur les trois dimensions de l'évolution

Le tableau suivant positionne l'approche d'évolution proposée dans xADL 2.0 par rapport aux trois dimensions d'évolution architecture logicielle.

Objet de l'évolution	Type de l'évolution	Support de l'évolution
Composant type	**Statique**	**Explicite** (Évolution continue) En utilisant les schémas de : - Versions - Variants
Connecteur type		
Interface Type	**Statique**	**Explicite** (continu) - schéma de versions
Subarchitecture (configuration)		
Composant	**Statique**	**Explicite** (continu) - schéma d'options
Connecteur		
Lien		

Table 2.8 – Positionnement de xADL2.0 sur les trois dimensions d'évolution

- xADL 2.0 est fortement inspiré par Mae. Il offre des constructions permettant de gérer l'évolution statique d'une architecture et notamment de garder trace de ces évolutions. Ces constructions consistent en un ensemble de schémas : schémas d'*Options*, de *Variants* et de *Versions*.
  - Schéma d'*Options* : ce schéma offre la possibilité de spécifier que certains composants, connecteurs et liens sont optionnels au moment de l'instantiation de l'architecture. Chaque élément optionnel est annoté par une condition (garde). L'évaluation de la garde au moment de l'instantiation déterminera l'introduction ou non de l'élément dans l'architecture.
  - Schéma de *Variants* : ce schéma offre la possibilité de spécifier que le type de certains composants et certains connecteurs peuvent varier au moment de l'instantiation. Un composant ou connecteur type Variant est défini en terme de plusieurs autres types, auxquels des gardes booléennes exclusives sont associées. Ainsi, au moment de l'instanciation, l'élément dont la garde est évaluée à vrai sera intégré.
  - Schéma de *Versions* : l'architecture d'un système évolue, ainsi de nouvelles versions de composants, de connecteurs et d'interfaces sont spécifiées. Une nouvelle version peut remplacer une ancienne version, mais aussi plusieurs versions d'un élément peuvent être utilisées dans un même système. Le schéma de versions permet de définir le *graphe de versions* de chaque *composant, connecteur* et *interface types*, et de garder le lien entre les différentes versions d'un élément.
- L'ensemble de ces schémas présentés étend les schémas de *Structures et Types*.
- L'évolution dans xADL2.0 est continue, le lien entre les différentes versions d'un élément peut toujours être établi via les graphes de versions de chaque élément

D'autres travaux sont proposés dans la littérature autour de xADL 2.0, destinés notamment à maîtriser l'évolution des architectures des *systèmes de familles de produits*. Parmi ces travaux, nous citons les propositions de Van der Westhuizen [88], autour de deux algorithmes. Le premier est un algorithme de détermination de différences (*Diff*) (cf. Annexe A) entre deux architectures décrites en xADL 2.0, en terme d'ajouts et de suppressions de composants et de connecteurs. Le second est un algorithme qui permet la propagation des différences identifiées par l'algorithme *Diff* vers une troisième architecture. Plus de détails sur ces deux algorithmes sont disponibles dans la page web de l'ADL xADL2.0 [20].

c. Evaluation

(+) Approche d'évolution permettant de garder et de relier les différentes évolutions d'une architecture logicielle et de celle de ses éléments ;

(+) Offre un ensemble de schémas XML permettant à l'architecte d'exprimer les différentes évolutions via une interface graphique pour modifier et supprimer des composants et des connecteurs.

(-) xADL2.0 ne gère pas les impacts de l'évolution de l'architecture sur le système en exécution.

2.3.3 Positionnement récapitulatif des approches étudiées sur les dimensions de l'évolution

Nous avons présenté dans les sections précédentes les différentes approches proposées par un ensemble d'ADL pour faire face à la problématique de l'évolution d'architectures logicielles. Dans cette section, nous présentons une comparaison et une analyse globale de ces approches d'évolution suivant les trois dimensions de l'évolution, et aussi suivant les deux générations d'ADL. Un tableau illustrant le positionnement global des différentes approches sur les dimensions de l'évolution est présenté en Annexe B.

2.3.3.1 Analyse du tableau

a. Analyse par rapport aux dimensions d'évolution

- **Objet d'évolution :**
  - La plupart des ADL considèrent l'évolution des *composants types* et des *configurations*. Pour les *connecteurs* et *interfaces types*, uniquement les ADL qui les réifient et les considèrent comme entités de premières classes abordent leurs évolutions tels que Mae et xADL2.0.
  - Peu d'ADL aborde l'évolution des *composants*, des *connecteurs* et des *interfaces*.
- **Type d'évolution :**
  - La plupart des ADL considèrent uniquement l'évolution *statique* des architectures logicielles ;
  - Les ADL tels que Dynamic ACME, Dynamic Wright abordent l'évolution *dynamique anticipée* d'une architecture logicielle. Autrement dit, ils permettent d'exprimer des besoins d'évolutions prévus et préalablement connus.
  - Aucun des ADL étudiés ne permet de refléter automatiquement les évolution de l'architecture vers le code source. De ce que nous avons étudié, seuls les outils ArchStudio (associé à

C2SADEL) et *ArchEvol* ont tenté de répondre à cette préoccupation, qui constitue l'un des défis actuels de la communauté d'architectures logicielles.

- **Support d'évolution :**
  - Pour l'évolution *statique* des *composants* et des *connecteurs types*, la plupart des ADL se limitent au mécanisme de *composition hiérarchique*. D'autres ADL tels que C2SADEL et Mae proposent aussi le *sous-typage hétérogène*.
  - Des ADL étudiés seul SafArchie propose des opérations d'évolutions ajout/suppression/ déplacement pour faire évoluer les *composants* et les *interfaces*.
  - xADL2.0 et Mae adoptent le *versionnent* pour faire évoluer une architecture logicielle.
  - L'évolution *statique* considérée dans la majorité des ADL est avec *rupture* (la trace de la situation avant l'évolution, n'est pas sauvegardée) à l'exception d'*xADL2.0* et *Mae*.
  - La plupart des ADL offrent des mécanismes et opérations prédéfini pour l'évolution de l'architecture logicielle.
  - Tous les ADL proposent des mécanismes pour exprimer des *contraintes*, des *invariants* et des *contrats* sur les élément d'une architecture. Ces mécanismes sont importants pour le maintien de la cohérence de l'architecture après évolution.

b. Analyse par rapport aux ADL de première et deuxième générations

Nous présentons de cette section une analyse des approches d'évolution étudiées suivant les deux générations d'ADL considérées :

- Les ADL de **première génération** : les ADL de première génération notamment, Wright et Darwin ne se sont pas préoccupés de la problématique de l'*évolution statique* d'une architecture logicielle. Néanmoins nous avons identifié certains mécanismes offerts par ces ADL qui permettaient l'évolution, notamment des *composants types* tel que la *composition hiérarchique*. Pour l'*évolution dynamique*, elle a été traitée d'une façon *explicite anticipée*. Les évolutions dynamiques susceptibles d'arriver sont ainsi spécifiées au moment de la spécification de l'architecture. Des ADL de la première génération, C2SADEL est le premier ADL ayant abordé l'évolution statique des composants via le sous-typage hétérogène. Des travaux autour de C2SADEL ont également permis d'apporter des propositions pour permettre quelques évolutions dynamiques d'une architecture logicielle, via l'outil *ArchStudio 1* [60].

- Les ADL de **deuxième génération** : la majorité des ADL de deuxième génération réifient et considèrent comme entités de premières classes tous les concepts de base de description d'une architecture logicielle : composant, connecteur, interface, configuration. Par conséquent la majorité de ces ADL abordent l'évolution de ces concepts réifiés. Les ADL Mae et xADL2.0 se sont focalisés sur le versionnement des architectures logicielles. L'avantage de ces ADL est leur capacité à prendre en compte l'évolution d'une architecture logicielle quel que soit son langage de description. Ces ADL permettent en effet, la projection des concepts des autres ADL vers leurs propres concepts. TranSAT proposé dans SafArchie offre des opérations et une démarche explicite pour l'évolution des composants, des interfaces et de la configuration. L'inconvénient de cet outil à sa dépendance de l'ADL SafArchie qui ne réifie pas le concept Connecteur.

2.4 Bilan sur les approches d'évolution étudiées

A partir des approches d'évolution étudiées et des analyses précédentes, nous présentons dans cette section un bilan récapitulatif. L'objectif de ce bilan est de recenser les points importants abordés par ces approches d'évolution et qui peuvent nous servir de supports pour définir un modèle d'évolution d'architectures logicielles. Ce bilan est scindé en deux parties : la première partie représente un bilan sur l'ensemble des opérations permettant de faire évoluer une architecture logicielle, la deuxième partie représente un ensemble de constatations globales établi à partir des évaluations des approches d'évolutions étudiées.

2.4.1 Taxonomie d'opérations d'évolution

Une opération d'évolution est toute opération pouvant être appliquée sur un élément de l'architecture provoquant son évolution. Nous constatons que les opérations d'évolution de base qui doivent être considérées par une approche d'évolution sont : l' **Ajout**, la **Suppression**, la **Modification** et la **Substitution** (remplacement). Ces mêmes opérations peuvent s'appliquer sur tout élément réifié dans une architecture logicielle. D'autres opérations plus complexes peuvent être construites à partir de ces opérations de base tels que la *fusion*, le *transfert*, le *versionnement*.

Nous constatons au travers des approches d'évolution étudiées que ces opérations d'évolution peuvent avoir plusieurs sémantiques. Nous présentons ces sémantiques dans le tableau suivant :

Opérations	**Sémantiques**
Ajout	signifie soit : - l'ajout d'un élément architectural existant à un autre élément architectural exp. : ajouter un composant existant à une configuration ; - l'ajout d'un élément architectural à un autre élément avec au préalable sa création (exp. : créer un composant puis l'ajouter à une configuration) ; - la création d'un élément architectural.
Suppression	signifie soit : - une suppression physique de l'élément architectural ; - une suppression logique (désactivation) de l'élément architectural) ;
Modification	signifie soit : -une modification de l'élément architectural, s'il est élémentaire (exp. : changement de valeur, renommage d'un service,...) ; - la conséquence d'autres opérations sur ses éléments constitutifs, s'il s'agit d'un élément architectural composite (exp. : supprimer un composant de la configuration revient à la modification de cette configuration en supprimant ce composant de la liste des composants de cette configuration).
Substitution	signifie : le remplacement d'un élément architectural par un autre

Table 2.9 – La sémantique des opérations d'évolution

Le tableau précédent présente les sémantiques associées à chaque opération d'évolution. La sémantique à considérer pour une opération d'évolution est déterminée par l'opération technique (code source) qui l'implémente dans l'approche d'évolution concernée. Ainsi, par exemple, même si l'opération de

suppression d'un composant ou d'un connecteur se ressemble puisqu'il s'agit de l'opération de suppression, les opérations techniques qui les implémentent sont forcément différentes.

2.4.2 Critères d'évolution architecturale

Nous présentons dans cette section un ensemble de constatations que nous avons établi suite aux différentes évaluations des approches d'évolution étudiées. Ces constatations représentent soit les manques ou les points positifs de ces approches. A partir de ces constations nous définissons un ensemble de critères que nous considérons comme importants à prendre en compte par une nouvelle approche d'évolution d'architecture logicielle.

- L'imbrication de la spécification et de la gestion des évolutions dans la spécification de l'architecture elle-même conduit généralement à des spécifications très complexes et difficiles à faire évoluer. Ceci est le cas, par exemple, de Wright et Darwin.

  C1 : Rendre indépendante autant que possible la spécification de l'évolution vis à vis de la spécification de l'architecture logicielle elle-même ;

- L'un des points fort de Mae est sa capacité à faire évoluer une architecture logicielle quel que soit son ADL, contrairement à d'autres tels que TranSAT de SafArchie.
  - **C2** : Il est important qu'une approche d'évolution soit indépendante de tout ADL pour qu'elle puisse être appliquée à toute architecture logicielle.

- Les éléments architecturaux peuvent être considérés sur plusieurs niveaux d'abstraction : niveau architectural (éléments architecturaux types) et le niveau application (instances de types). Certains ADL offre des mécanismes pour faire évoluer les éléments architecturaux types et d'autres uniquement pour faire évoluer les éléments architecturaux instances de ces types.
  - **C3** : une approche d'évolution doit pouvoir gérer l'évolution des éléments architecturaux sur ces différents niveaux d'abstraction, ainsi que les différents impacts qui peuvent exister entre ces différents niveaux.

- La majorité des ADL offrent des mécanismes différents pour faire évoluer chaque élément de l'architecture.
  - **C4** : il est important d'offrir des mécanismes uniformes pour l'évolution de tout élément de l'architecture et sur n'importe quel niveau d'abstraction. Ceci faciliterait la tâche au concepteur.

- La propagation des impacts de l'évolution est très peu abordée par les ADL que nous avons étudié.
  - **C5** : La propagation des impacts d'une évolution est une étape importante et indispensable pour garantir la cohérence globale de l'architecture après évolution.

- La majorité des ADL ont eu recours à la définition des contraintes, invariants, contrats. pour contraindre l'évolution de l'architecture logicielle.
  - **C6** : la définition des contraintes (invariants, contrat, etc.) est nécessaire pour pouvoir contrôler la cohérence de l'architecture au cours de l'évolution.

Nous nous basons sur ces différents critères pour définir le modèle d'évolution que nous proposons pour faire face à la problématique de l'évolution dans les architectures logicielles.

2.5 Conclusion

Nous avons abordé dans ce chapitre les approches d'évolution d'architectures logicielles au travers d'un ensemble d'ADL. Un des premiers aspects a été de proposer une classification de l'évolution d'architecture logicielle par rapport à ses caractéristiques que nous dénommons dimensions de l'évolution : l'*objet de l'évolution*, le *type de l'évolution*, et le *support de l'évolution*. La classification de l'évolution d'architecture logicielle par rapport à ces dimensions permet d'offrir un référentiel commun indépendant de toute approche d'évolution d'architecture logicielle. Il est alors plus facile de comparer les approches d'évolution par rapport à ce même référentiel.

En se basant sur cette classification nous avons étudié un ensemble d'approches d'évolution proposées par des ADL de première et de deuxième générations. Nous avons décrit ces approches suivant les différentes dimensions de l'évolution d'architecture logicielle que nous avons identifié. Ceci nous a permis de distinguer, pour chaque approche, les aspects de l'évolution qui ont été traités de ceux qui ne le sont pas.

Nous avons conclu cette étude en apportant notre propre évaluation de ces approches d'évolution. En se basant sur ces dernières, nous avons établi un bilan récapitulatif. Dans la première partie de ce bilan nous avons recensé les différentes opérations proposées indifféremment dans toute approche d'évolution, pour faire évoluer une architecture logicielle, ainsi que les sémantiques associées à chacune de ces opérations. Dans la deuxième partie de ce bilan nous avons établi un ensemble de critères que nous jugeons importants à prendre en compte dans une nouvelle approche d'évolution. Nous nous basons sur ces critères pour définir dans le chapitre suivant un modèle d'évolution d'architecture logicielle, le modèle SAEV.

CHAPITRE 3

SAEV : un modèle d'évolution pour les architectures logicielles

3.1 Introduction

Pour gérer l'évolution structurelle des architectures logicielles à base de composants au vu de notre problématique, nous proposons un modèle d'évolution baptisé SAEV pour Software Architecture EVolution model. Nous mettons principalement en avant, dans le modèle SAEV, la nécessité d'abstraire et de traiter l'évolution d'une architecture logicielle, indépendamment du comportement proprement dit de ses éléments et la nécessité d'offrir des mécanismes uniformes pour la gestion de l'évolution de n'importe quel élément architectural. Il est important de souligner que nous ne considérons aucun ADL particulier. En effet, SAEV doit être capable de gérer l'évolution d'une architecture logicielle, quel que soit son langage de description, et à n'importe quelle phase de développement du système qu'elle décrit. SAEV se veut un modèle uniforme et indépendant de tout ADL.

Nous présentons premièrement, dans la section 3.2, les objectifs assignés à SAEV au vu des critères établis à partir des approches d'évolutions des différents ADL, étudiées dans le chapitre 2. Nous présentons ensuite dans la section 3.3 les différents concepts de SAEV ainsi que son mécanisme opératoire pour spécifier et gérer une évolution. Dans la section 3.4, nous montrons comment positionner le modèle SAEV par rapport aux niveaux d'abstraction de description d'une architecture logicielle définis dans la section 1.4.3 du chapitre1. Nous concluons le chapitre par un bilan qui resitue le modèle SAEV par rapport aux objectifs fixés en section 3.2.

3.2 Préoccupations et objectifs de SAEV

Nous proposons le modèle SAEV (Software Architecture EVolution model) pour répondre à la problématique de l'évolution structurelle des architectures logicielles à base de composants. Le modèle d'évolution SAEV doit permettre au concepteur d'exprimer et de spécifier les changements qu'il souhaite appliquer sur son architecture logicielle. Il doit ensuite se charger de répercuter ces changements sur cette architecture et de gérer leurs impacts.

SAEV prend, en entrée une architecture logicielle cohérente, sur laquelle il répercute les différents changements exprimés par le concepteur et leurs impacts induits. Il doit aboutir ensuite à une architecture logicielle ayant évolué et cohérente.

Pour répondre à ces préoccupations et en tenant compte des différentes critères établis dans le chapitre précédent, SAEV doit :

- Offrir un niveau d'abstraction de l'évolution des architectures logicielles, en dissociant complètement le traitement de l'évolution des éléments d'une architecture logicielle, de leurs comportements proprement dit. Il s'agit en effet de spécifier explicitement l'évolution pour pouvoir la traiter **explicitement** et **indépendamment** de la spécification des éléments architecturaux eux-mêmes, et donc indépendamment de leurs langages de description (**C1**, **C2**).
- Permettre la spécification et la gestion d'une manière uniforme de l'évolution de tout élément architectural pouvant être réifié par un ADL (composant, connecteur, configuration, etc.) et à travers les différents niveaux d'abstraction de description d'architectures logicielles identifiés dans la section 1.4.3 du chapitre 2 (**C3**, **C4**).
- Prendre en compte explicitement la propagation des impacts de l'évolution d'un élément architectural sur les autres éléments concernés de l'architecture, tout en sauvegardant la cohérence globale de cette architecture (**C5**).
- Permettre l'évolution statique et dynamique des architectures logicielles. En effet, les besoins d'évolution d'une architecture logicielle peuvent apparaître à n'importe quelle phase du cycle de vie du système qu'elle décrit. Pour couvrir l'ensemble de ces besoins, SAEV doit être capable de traiter l'évolution statique d'une architecture donc à la phase de sa spécification (ou à l'arrêt du système qu'elle décrit), comme il doit être capable de traiter son évolution dynamique donc au cours de l'exécution du système qu'elle décrit.
- Favoriser la réutilisation des spécifications d'évolution. En effet, le modèle SAEV doit faciliter au concepteur la spécification des évolutions qu'il souhaite répercuter sur l'architecture en lui permettant de réutiliser des spécifications d'évolution déjà définies.

3.3 Description de SAEV

Avant de présenter le modèle SAEV, nous rappelons notre perception de l'évolution structurelle d'une architecture logicielle. Nous considérons que l'évolution structurelle d'une architecture logicielle se reflète par les différents changements dans sa structure et/ou dans celle de ses éléments constitutifs. Un changement dans une architecture est toujours engendré par une ou plusieurs opérations appliquées à cette architecture ou à l'un de ses éléments : ajouter un composant, supprimer un connecteur, modifier une interface, etc. Ces opérations expriment en effet les besoins d'évolution qui doivent être appliqués sur l'architecture.

Pour caractériser ainsi une évolution au sein d'une architecture logicielle et donc pour pouvoir la réifier, il faut identifier, au minimum, l'élément architectural sur lequel porte l'évolution, l'opération à appliquer sur cet élément, les impacts de cette opération sur l'élément architectural considéré et sur les autres éléments concernés de l'architecture.

Le modèle d'évolution SAEV offre ainsi, au vu de ces caractéristiques et des objectifs précédemment cités, des concepts définis au sein de son méta-modèle permettant à la fois la spécification et la gestion de l'évolution d'une architecture logicielle ainsi qu'un processus opératoire décrivant les étapes à suivre pour mener une évolution. Nous les présentons dans les deux sections suivantes.

3.3.1 Méta modèle de SAEV

Nous nous appuyons sur une modélisation objet pour la description des concepts du Méta-modèle de SAEV. Nous utiliserons précisément le formalisme du diagramme de classe de la notation UML [[35]. Ainsi, les concepts clés de *classe*, d'*héritage*, de *composition* et d'*association* sont utilisés, ainsi que la notion de *multiplicité* sur les relations.

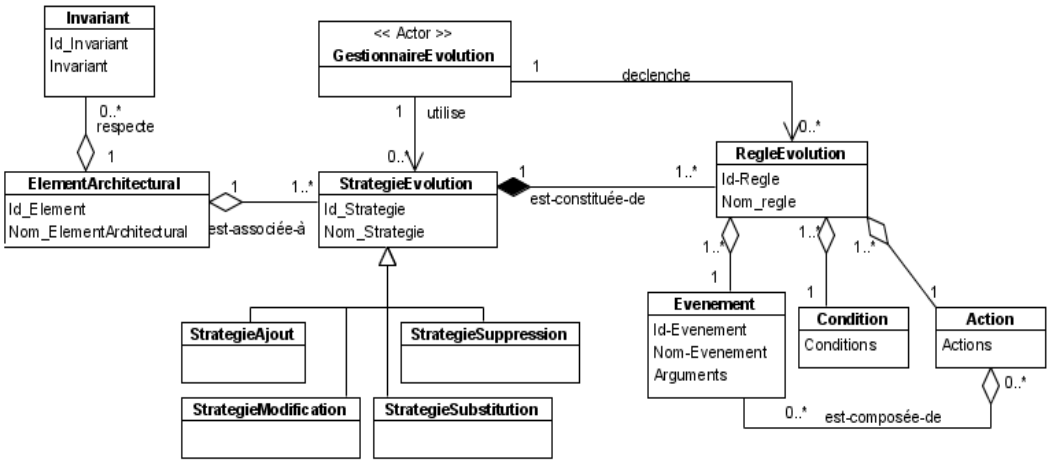

Figure 3.1 – Méta-modèle de SAEV

Le diagramme de classes met en évidence les concepts proposés par SAEV, ainsi que les associations qui existent entre eux. Il se focalise principalement sur la manière dont sont organisés les concepts et non sur la description détaillée de ces concepts.

Le concept d'*Elément Architectural* représente tout élément réifié de l'architecture à faire évoluer. A chaque élément architectural correspond des *Invariants* et des *Stratégie d'évolution*. Une stratégie regroupe les *Règles d'évolution* d'un élément architectural. Une règle d'évolution décrit l'application d'une opération d'évolution sur un élément architectural. Une règle est déclenchée si l'*Événement* lui correspondant survient et selon certaines *Conditions*. Une règle peut déclencher d'autres règles, si nécessaire, pour propager l'impact de l'opération qu'elle décrit. Ainsi, une action d'une règle peut correspondre à un évènement. Nous décrivons en détail ces concepts dans la section suivante.

3.3.2 Concepts de SAEV

Nous présentons dans cette section les concepts proposés par SAEV pour la spécification et la gestion de l'évolution. Pour chaque concept nous apportons sa définition et un exemple illustratif.

3.3.2.1 Élément architectural

a. Définition : l'élément architectural représente tout élément significatif dans une architecture logicielle. Nous avons introduit ce concept dans SAEV pour modéliser et réifier tout élément de l'architecture logicielle à faire évoluer. Pour SAEV, à chaque niveau d'abstraction (cf. section 1.4.3 chapitre 1) de description d'architecture logicielle (nivea Méta, niveau Architectural et niveau Application), un élément

architectural peut être aussi bien une *Configuration*, un *Composant*, un *Connecteur*, une *Interface* ou tout autre concept supporté par l'ADL qui a servi à la description de l'architecture logicielle à faire évoluer.

b. Exemples : au niveau Méta, l'élément architectural peut représenter le concept *Composant*, au niveau Architectural, il peut représenter le *Composant type Client* et au niveau Application, il peut représenter le *Composant Client Banque*.

Il est important d'identifier le niveau d'abstraction des éléments de l'architecture à faire évoluer. En effet, nous nous basons sur le principe suivant : pour gérer l'évolution d'une architecture logicielle définie à un niveau d'abstraction donné, il faut d'abord identifier le niveau d'abstraction supérieur qui a permis l'instanciation (la création) de l'architecture. Toute évolution sera alors spécifiée sur la base des éléments de ce niveau supérieur pour qu'elle puisse être appliquée à l'évolution de toute architecture instance de ce niveau. Ceci est comparable à la définition des classes et des objets dans le paradigme objet, où les structures et les comportements des objets sont toujours décrits par leurs classes mais qui s'appliquent aux objets. Nous reviendrons sur ce principe dans la section 3.4 de ce chapitre.

3.3.2.2 Invariant

a. Définition : un invariant représente une contrainte structurelle sur un élément architectural ou sur l'ensemble de l'architecture. Un invariant doit être respecté tout au long du cycle de vie de l'élément architectural, quelle que soit l'opération d'évolution qui lui est appliquée. Tout changement dans l'architecture logicielle doit garantir le maintien de chaque invariant pour sauvegarder la cohérence globale de l'architecture. Ainsi, au niveau d'abstraction Méta, sont définis des invariants relatifs à la définition des concepts selon l'ADL considérée et au niveau Architectural, sont définis les invariants relatifs à la structure d'une architecture type.

b. Exemples :
- Au niveau Méta, les invariants suivants relatifs à une *configuration* (à noter que le concept de configuration est réifié au niveau méta) dans l'ADL COSA [76] peuvent être définis : une configuration doit être composée au moins d'un composant ; un connecteur, dans une configuration doit relier au minimum deux composants ;

- Au niveau Architectural, en considérant une architecture Client/Serveur, les invariants suivants peuvent être définis : le nombre de Clients reliés à un Serveur ne doit pas dépasser 30, une Base de données est reliée à un et un seul Serveur.

A l'évolution d'une architecture logicielle décrite à un niveau d'abstraction donné, les invariants définis au niveau d'abstraction supérieur doivent être respectés. Ainsi, à l'évolution d'une architecture au niveau Architectural, les invariants définis pour les éléments du niveau Méta doivent être respectés et à l'évolution d'une architecture décrite au niveau Application, les invariants définis au niveau Architectural et par conséquent au niveau Méta, doivent être respectés. Ces invariants permettent à SAEV de guider l'évolution d'une part et de vérifier le maintien de la cohérence globale de l'architecture après toute évolution d'autre part.

3.3.2.3 Stratégie d'évolution

a. Définition : une stratégie d'évolution encapsule ce qui permet de décrire et d'appliquer les différentes évolutions possibles sur un élément architectural. Une stratégie d'évolution regroupe, en effet, les

règles permettant de spécifier l'évolution d'un élément architectural.

Nous avons souligné dans la section 2.4.1 du chapitre 2, que les évolutions minimales d'un élément architectural se traduit par l'une des opérations d'évolution suivantes : Ajout, Suppression, Modification, Substitution. Ainsi, pour chaque élément architectural, de chaque niveau d'abstraction sont associées au minimum quatre stratégies :

- **Stratégie d'ajout** : regroupe toutes les règles décrivant l'ajout d'un élément architectural ;
- **Stratégie de suppression** : regroupe toutes les règles décrivant la suppression d'un élément architectural ;
- **Stratégie de modification** : regroupe toutes les règles décrivant la modification d'un élément architectural ;
- **Stratégie de substitution** : regroupe toutes les règles décrivant la substitution d'un élément architectural.

Nous notons que d'autres stratégies peuvent être construites, si des besoins de spécifier des opérations d'évolution plus complexes sont apparus.

b. Exemple :

SSComp : **S**tratégie de **S**uppression d'un **Comp**osant au niveau *architectural* ;

SSAComp_Client : **S**tratégie de **S**uppression d'un **Compo**sant *Client* au niveau ***A**pplication*.

Nous dissocions ces quatre stratégies d'évolution pour faciliter l'identification de la stratégie d'évolution et par conséquent l'identification de la règle nécessaire pour faire évoluer un élément architectural. Ceci, permet ainsi de réduire le temps de la gestion de l'évolution. La présentation détaillée d'une règle d'évolution est donnée dans la section suivante.

3.3.2.4 Règle d'évolution

a. Définition : le concept de règle d'évolution permet d'exprimer et de spécifier les évolutions qui peuvent être menées sur une architecture logicielle. Une règle d'évolution permet la description de l'application d'une opération d'évolution (ajout/suppression/modification/ substitution) sur un élément architectural, en spécifiant les conditions nécessaires pour le faire, ainsi que les éventuels impacts qu'elle peut engendrer sur les autres éléments architecturaux. En effet, la propagation des impacts est nécessaire notamment pour sauvegarder la cohérence de l'architecture ayant évolué. Les impacts d'une évolution peuvent concernés les éléments du même niveau ou les éléments des niveaux inférieurs quand ils existent.

Nous avons choisi de modéliser les règles d'évolution par le formalisme ECA (évènement/ Condition/ Action) [22]. Nous jugeons que ce formalisme est adapté pour permettre à une architecture logicielle d'être réactive, c'est à dire de réagir automatiquement à des évènements d'évolution qu'elle peut recevoir. En effet, grâce à cette approche événementielle, la règle à déclencher est élue dynamiquement dès la réception de l'évènement qui lui est associé. Le formalisme ECA permet aussi de spécifier le déclenchement automatique de la propagation des impacts d'une évolution. Chaque règle d'évolution de SAEV est ainsi composée :

- **D'un évènement** : qui représente le message d'invocation qui peut être émis par l'environnement (concepteur ou par une autre règle) vers l'élément à faire évoluer. L'évènement représente ainsi le

besoin d'évolution. C'est la survenance de l'évènement qui peut déclencher une règle d'évolution ;

- **D'une partie condition** : cette partie exprime ce qui doit être nécessairement vérifié pour exécuter une régle évolution. La partie condition peut correspondre à une seule condition qui porte sur un élément architectural ou une conjonction de conditions sur un ou plusieurs éléments architecturaux.
- **D'une partie action** : la partie action décrit les changements, à proprement parlé, à appliquer sur l'élément architectural auquel est associée la règle d'évolution, ainsi que les impacts de ces changements à déclencher sur les autres éléments architecturaux. La propagation des impacts d'une évolution est exprimée sous forme d'invocation d'autres règles. Ainsi, une action dans une règle d'évolution peut correspondre à :
  - un évènement (une référence vers une autre règle) , dans ce cas il sera redirigé par le gestionnaire d'évolution vers d'autres règles d'évolution. Le but de l'évènement est alors de déclencher d'autres règles d'évolution pour propager les impacts de l'évolution traitée. Par exemple, une règle de suppression d'un port peut déclencher la règle de suppression de l'attachement associé à ce port ;
  - une opération élémentaire à exécuter sur un élément architectural et qui provoque son évolution. Une opération élémentaire invoque directement le code qui implémente l'opération d'évolution à exécuter.

Les actions d'une règle d'évolution sont exécutées séquentiellement action par action. Dans SAEV, la sémantique associée à une règle d'évolution est la suivante : une règle se trouve dormante jusqu'à l'occurrence de l'évènement qui la compose. Quand l'évènement a eu lieu, et si les conditions sont satisfaites, alors la partie action est exécutée apportant une réponse appropriée à l'évènement.

La convention utilisée pour la spécification des règles d'évolution de SAEV est donnée par la table suivante :

Désignation
Evénement :
Condition :
Action :

Table 3.1 – Convention de notation d'une règle

- La désignation d'une règle est notée :

  Ri : **Règle de { Opération d'Evolution } {Elément architectural}**

  i : représente une chaîne de caractère.

  Exemple :

  R1 : Règle de suppression Composant ;
  RA1 : Règle de suppression Composant Client

- L'évènement d'une règle est noté :

  Opération Évolution-Élément Architectural (**Elt** : **Elément Architectural**, [**autres paramètres**]).

  Exemple : Supprimer-Composant(C :Composant, Cf : Configuration)

- La partie condition d'une règle est exprimée sous forme d'une conjonction de prédicats sur les éléments architecturaux ou sur des ensembles de ces éléments architecturaux. Un ensemble peut être par exemple :

  Composants(Cf) : l'ensemble des composants de la configuration **Cf**.

  Un prédicat sur cet ensemble peut être par exemple

  C $\in$ **Composants(Cf)** : qui renvoie **vrai** si le composant **C** est bien un composant de la configuration **Cf**.

- Une action est notée en utilisant la convention de description d'un évènement, si elle correspond à un évènement. Si l'action correspond à une *opération élémentaire*, elle sera notée par :

  Elt.Execute. Operation Evolution([Paramètres])

Les actions d'une règle peuvent être imbriquées dans des structures itératives et conditionnelles.

Les règles d'évolution sont répertoriées par type de stratégies d'évolution. Ainsi, par exemple, les règles relatives à la suppression d'un élément seront mémorisées dans sa *stratégie de suppression* et les règles relatives à son ajout seront mémorisées dans sa *stratégie d'ajout*.

b. Exemple :

Nous présentons dans ce qui suit un exemple d'une règle d'évolution de la *stratégie de suppression* d'un composant. Cette règle décrit l'application de l'opération de *suppression* sur l'élément architectural *composant*.

R1 :Règle-Suppression-Composant
Evenement : supprimer-composant(C : composant, Cf : Configuration) ;
Condition : C $\in$ Composants(Cf), $\exists$ B tel que B $\subset$ Bindings(Cf,C) et B$\neq \phi$;
Action : **Pour tout** b $\in$ B Supprimer-binding(b,Cf,C) ; (1) **FinPour** **Pour tout** I $\in$ interface-composant(C) Supprimer-interface-composant(I,Cf,C) ; (2) **FinPour** Modifier-configuration(Cf, C) ; (3) *C.Execute-supprimer-composant(Cf).* (4)

Table 3.2 – Exemple de règle d'évolution

La règle *R1* décrit la suppression d'un composant *C* qui se trouve à l'extrémité d'une configuration *Cf*. Ceci est exprimé par la condition ($\exists B tel que B \subset Bindings(Cf, C) et B \neq \phi$) qui indique, qu'il existe

des bindings entre la configuration *Cf* et le composant *C*. La partie action de cette règle indique que pour la suppression de ce composant, il faut d'abord :

(1) : Supprimer les bindings auxquels les interfaces de ce composant participent ;

(2) : Supprimer toutes les interfaces du composant ;

(3) : Modifier la configuration à laquelle appartient le composant, en supprimant le composant de la liste des composants de cette configuration.

(4) : Déclencher la suppression proprement dite du composant C.

Les trois premières actions de *R1* correspondent réellement à des évènements qui déclenchent des règles d'évolution relatives respectivement à un *binding*, une *interface composant* et une *configration*. La dernière action invoque le code source qui implémente l'opération de suppression d'un composant.

3.3.2.5 Gestionnaire d'évolution

Tous les concepts précédemment définis ont un rôle descriptif. Le gestionnaire d'évolution quant à lui est chargé de gérer l'évolution d'une architecture logicielle donnée au vu des stratégies d'évolution définies et de leurs règles d'évolution. Son rôle est d'intercepter tout évènement d'évolution émanant du concepteur ou des règles d'évolution vers l'architecture ou l'un de ses éléments, puis d'identifier la stratégie d'évolution correspondante. Suivant cette stratégie, il déclenche la ou les règles d'évolution associées à l'évènement reçu. Il est chargé ensuite de propager les impacts des règles d'évolution exécutées. Le gestionnaire d'évolution doit vérifier aussi le maintien des invariants. L'ensemble de ces tâches du gestionnaire d'évolution sera détaillé dans le mécanisme opératoire de SAEV décrit dans la section suivante.

3.3.3 Processus de SAEV

Nous présentons dans cette étape le processus général de SAEV. Ce processus est composé de deux phases : la spécification de l'évolution et l'exécution d'une évolution.

3.3.3.1 Phase de spécification de l'évolution

SAEV offre les concepts nécessaires pour spécifier une évolution ainsi que ses impacts. Spécifier une évolution en SAEV revient à spécifier les éléments de l'architecture à faire évoluer, selon le niveau d'abstraction considéré. Pour chaque élément architectural, il faut spécifier :

- ses invariants ;
- ses stratégies d'évolution ;
- ses règles d'évolution associées à chacune des stratégies d'évolution.

3.3.3.2 Phase d'exécution d'une évolution

Le mécanisme opératoire du modèle SAEV pour exécuter une évolution est illustré dans la figure 3.2, via le diagramme d'activité de la notation UML [35]. Ce mécanisme opératoire est composé de quatre étapes :

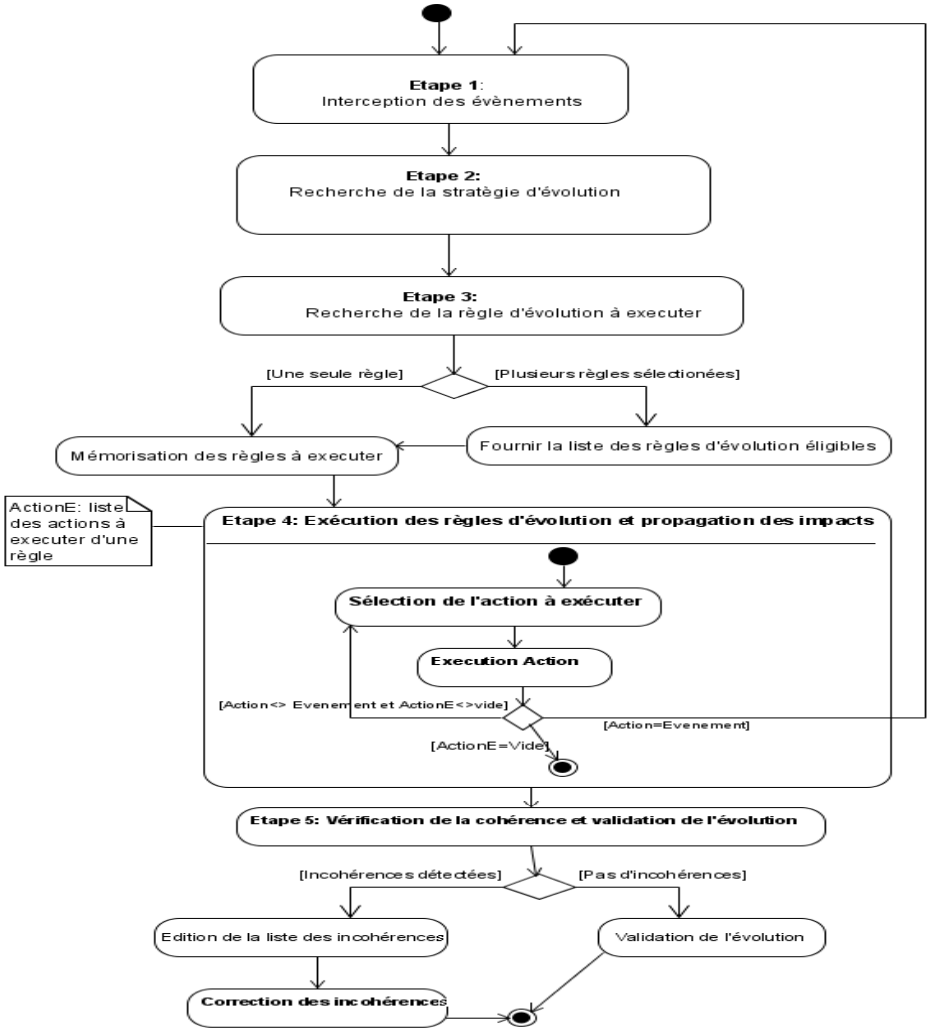

Figure 3.2 – Mécanisme opératoire de SAEV

Etape1 : Interception des évènements

Le gestionnaire d'évolution doit intercepter deux types d'évènements : les évènements émanant du concepteur et les évènements émanant des règles d'évolution :

- Les évènements émanant du concepteur : pour réaliser une évolution, le concepteur sélectionne l'élément architectural qu'il souhaite faire évoluer ainsi que l'opération à exécuter sur cet élément architectural. Le message représentant ainsi les choix du concepteur est intercepté par le gestionnaire d'évolution comme étant l'évènement de l'évolution.
- Les évènements émanant des règles d'évolution : l'exécution d'une règle d'évolution peut invoquer d'autres évènements pour propager l'impact qu'elle provoque. Ces évènements sont également interceptés par le gestionnaire d'évolution.

Étape 2 : Recherche de la stratégie d'évolution

A l'interception d'un évènement d'évolution, le gestionnaire d'évolution recherche la stratégie d'évolution correspondante à l'évènement intercepté. La stratégie d'évolution est identifiée via les champs "**Elément Architectural**" et "**Opération Evolution**" de l'évènement reçu.

Par exemple, via l'évènement *Supprimer-composant(C : composant, Cf :configuration)*, la stratégie de *Suppression* de *composant* sera sélectionnée.

Étape 3 : Recherche de la règle d'évolution à exécuter

Une fois la stratégie d'évolution identifiée, le gestionnaire d'évolution recherche la ou les règles d'évolution à exécuter. Toutes les règles dont la partie *évènement* correspondent à l'évènement reçu, et dont la partie *Condition* est évaluée à vrai, sont sélectionnées. Deux cas se présentent alors :

- Le gestionnaire d'évolution sélectionne une seule règle d'évolution ; dans ce cas il déclenche son exécution.
- Le gestionnaire d'évolution sélectionne plusieurs règles d'évolution. Il fournit alors au concepteur la liste de ces règles d'évolution éligibles. Ce dernier doit alors intervenir pour le choix d'une seule règle d'évolution à déclencher.

Afin de permettre au concepteur d'annuler son choix de règle d'évolution, et par ailleurs lui permettre de sélectionner une autre règle, l'ensemble des règles d'évolution éligibles est mémorisé jusqu'à ce que l'évolution entamée soit validée.

Étape 4 : Exécution des règles d'évolution et la propagation des impacts

Le déclenchement d'une règle revient à l'exécution de sa partie action, séquentiellement action par action. Si une action correspond à un évènement de propagation, ce dernier sera alors intercepté, et de la même façon une nouvelle règle sera exécutée. Le processus se réitère pour chaque règle exécutée. Chaque action d'une règle est exécutée ainsi que ses propagations, avant de passer à l'action suivante. Par exemple, si une règle *R1* (figure 3.3), constituée des actions *a1*, *a2* est exécutée. Si *a1* correspond à un évènement qui déclenche la règle *R2*, alors toutes les actions de *R2* (a3, a4) doivent être exécutées avant de passer à l'action *a2* de *R1*.

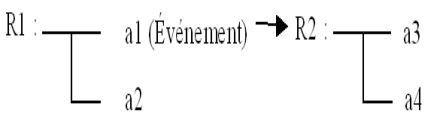

Figure 3.3 – Execution séquentielle des actions

L'arborescence des règles déclenchées et par conséquent des actions exécutées suite à un évènement de l'utilisateur, ainsi que les contextes de leurs exécution sont mémorisés, jusqu'à validation de l'évolution.

Etape 5 : la vérification de la cohérence et validation de l'évolution

La vérification de la cohérence se fait au vu des invariants associés aux éléments architecturaux des règles d'évolution exécutées. La vérification de la cohérence peut être effectuée selon deux modes : le mode *Immédiat* et le mode *Différé*.

Le mode Immédiat : la vérification de la cohérence est lancée, après chaque exécution d'une opération d'évolution élémentaire d'une règle. Ceci permet au concepteur de suivre l'impact de chaque opération d'évolution élémentaire pas à pas. Les incohérences, si elles sont provoquées, sont détectées immédiatement à l'exécution de l'opération élémentaire. Le concepteur peut alors annuler l'exécution de la règle et/ou refaire un autre choix de règles à exécuter, si elles existent. Dans le cas ou l'exécution des règles ne provoque aucune incohérence, l'évolution est validée et l'architecture est mise à jour.

Le mode Différé : la vérification de la cohérence est lancée, une fois que toutes les actions de la règle exécutée, suite à un évènement émis par le concepteur, sont exécutées, ainsi que leurs propagations. Dans ce mode, il n'est plus nécessaire de vérifier la cohérence après chaque opération d'évolution élémentaire. En effet, dans certains cas même, si momentanément une opération d'évolution élémentaire provoque une incohérence, elle peut être suivie d'un évènement qui déclenche d'autres règles et qui lèveront cette incohérence, sans l'intervention du concepteur.

Par exemple, si une vérification de la cohérence est lancée juste après la seule opération de *suppression* d'un *composant*, l'invariant lié à l'interface qui indique que toute interface doit être liée à un composant sera indiqué comme étant violée. Mais, dans le cas où cette opération est suivie d'un évènement qui déclenche la suppression des interfaces de ce composant, à la fin des deux opérations la cohérence est maintenue. A la vérification de la cohérence deux cas peuvent se présenter :

- Aucune incohérence n'est détectée, alors l'évolution est validée et l'architecture est mise à jour.
- Des incohérences sont détectées, une liste des invariants violés est alors présentée au concepteur. Ce dernier peut alors naviguer dans l'arborescence des règles et actions exécutées et peut revenir à tout état antérieur, en annulant l'exécution de certaines règles. Il peut aussi faire d'autres choix de règles en se basant sur la liste des règles éligibles, jusqu'à aboutir à un état cohérent.

Le processus d'évolution présenté sera réitéré pour chaque évènement d'évolution émis par le concepteur vers l'architecture à faire évoluer.

Nous avons illustré dans les deux sections précédentes la description des différents concepts du modèle d'évolution SAEV pour spécifier et gérer l'évolution d'une architecture logicielle, ainsi que son mécanisme opératoire pour mener une évolution. Dans la section suivante nous montrons comment positionner le modèle SAEV par rapport aux différents niveaux d'abstraction d'une description d'architecture logicielle.

3.4 SAEV et les niveaux d'abstraction d'une architecture logicielle

Nous avons identifié dans la section 1.4.3 du chapitre 1, que la description d'une architecture logicielle peut se faire au travers de trois niveaux d'abstraction : le *niveau Méta*, le *niveau Architectural* et le *niveau Application*. Nous rappelons que le *niveau Méta* décrit l'ensemble des concepts ainsi que les liens entre ces concepts que propose un ADL pour la spécification d'une architecture logicielle. Le *ni-*

veau Architectural est le niveau de description d'une architecture en instanciant un ou plusieurs concepts du niveau Méta. Le *niveau Application* décrit l'architecture du système en instanciant un ou plusieurs éléments architecturaux du niveau Architectural.

Dans SAEV, il est important d'identifier le niveau d'abstraction de description de l'architecture à faire évoluer. En effet, ce niveau permettra d'identifier les éléments architecturaux du niveau d'abstraction *au-dessus* qui a permis l'instanciation de l'architecture à faire évoluer. L'évolution de chaque élément du niveau supérieur sera alors spécifiée. Ces évolutions seront ainsi appliquées aux éléments de l'architecture à faire évoluer. SAEV permet de spécifier et de gérer l'évolution d'une architecture d'une manière uniforme aussi bien au niveau Architectural qu'au niveau Application (figure 3.4).

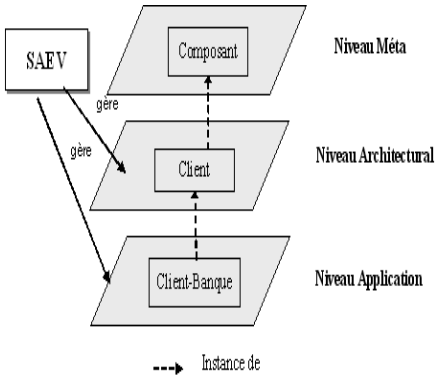

Figure 3.4 – SAEV et les niveaux d'abstraction

Nous montrons l'application de SAEV sur les deux niveaux d'abstraction dans les deux paragraphes suivants.

3.4.1 Évolution au niveau Architectural

Pour faire évoluer une architecture au niveau Architectural, l'évolution est décrite au niveau Méta. Les éléments du *niveau Méta* (les concepts de l'ADL qui a servi à la description de l'architecture) sont spécifiés via le concept d'**Elément Architectural** de SAEV. A chaque élément architectural et donc à chaque concept du *niveau méta* sont spécifiés ses **invariants**, ses **stratégies d'évolution** et ses **règles d'évolution**. Ces évolutions ainsi exprimées sur les éléments du *niveau Méta* seront appliquées à l'évolution de toute architecture créee à partir de ce niveau Méta [62]. Un exemple est illustré dans la figure suivante :

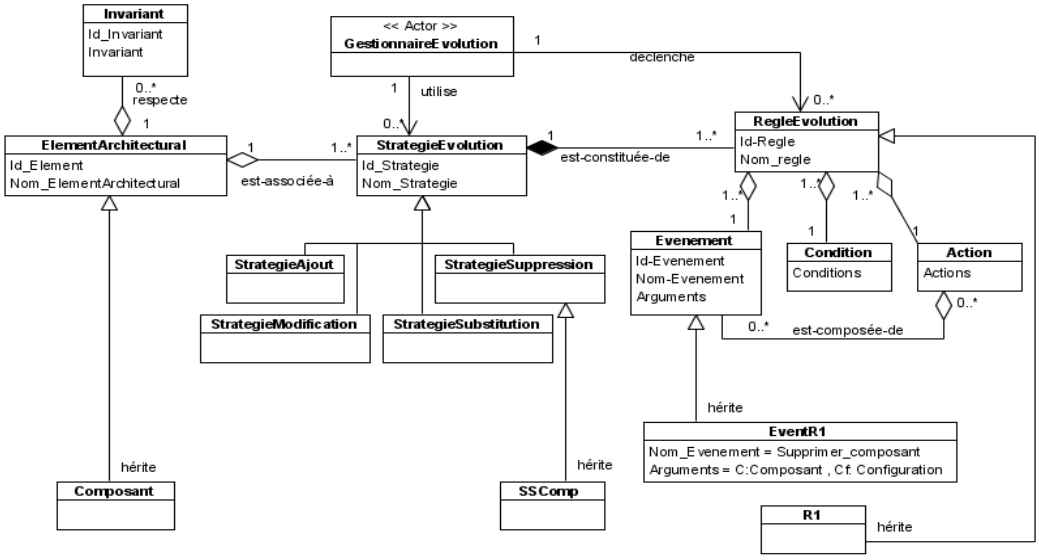

Figure 3.5 – Description de SAEV au niveau Méta

La figure 3.5 illustre la stratégie de suppression d'un composant *SSComp* et la règle de suppression d'un composant *R1 : Suppression composant* associés au concept *Composant*. Ainsi à la suppression de tout composant d'une architecture au niveau Architectural, la stratégie *SSComp* sera sélectionnée par le gestionnaire d'évolution. Dans cette stratégie, la règle *R1*, sera sélectionnée et exécutée si ses conditions sont satisfaites

A titre d'illustration nous présentons la règle d'évolution R1 associée au concept *Composant*.

R1 : Suppression d'un Composant
Evènement : Supprimer-composant(C :Composant, Cf :Configuration)
Condition : C $\in$ composants(Cf)
Action : Pour tout I $\in$ interface-composant(C) Supprimer-interface-composant (I, C,Cf) Finpour Modifier-configuration(Cf, C) *C.Execute-supprimer-composant()*

Table 3.3 – Règle de suppression d'un composant

Cette règle ainsi associée au concept *composant* sera appliquée à la suppression de tout composant au niveau Architectural.

3.4.2 Évolution au niveau Application

Pour gérer l'évolution d'une architecture au *Niveau Application*, le concepteur doit spécifier les **stratégies d'évolution**, **règles d'évolution** et **invariants** associés à chaque **élément** défini au niveau Architectural et qui a servi à la description de l'architecture de l'application. Ces évolutions décrites ainsi au

niveau Architectural seront appliquées à l'évolution de tout élément d'une application créée à partir de ce niveau Architectural. Un exemple est illustré dans la figure suivante :

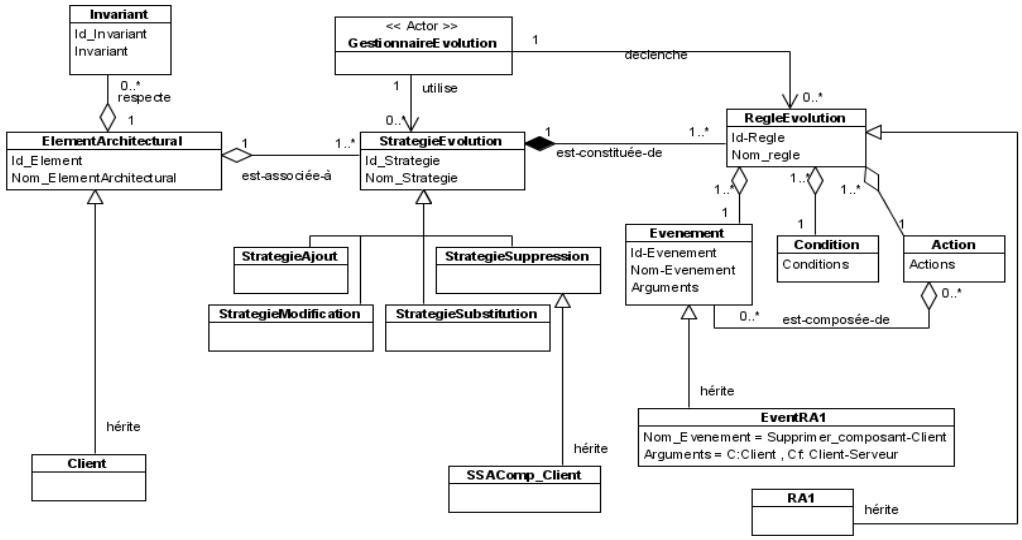

Figure 3.6 – Description de SAEV au niveau Architectural

La figure illustre la stratégie *SSACompClient* : suppression d'un composant *Client* associée au composant Client, Ainsi que la règle RA1 décrivant la suppression d'un composant Client. Nous présentons à titre d'exemple une règle de suppression du composant *Client*.

RA1 : Suppression d'un Composant Client
Évènement : Supprimer-composant-Client(C :Composant-Client, Cf :Configuration-Client-serveur)
Condition : C $\in$ Composants-Client(Cf)
Action : Pour tout I $\in$ interface-composant-Client(C) Supprimer-interface-composant-Client (I, C,Cf) Finpour Modifier-configuration-Client-Serveur(Cf, C) *C.Execute-supprimer-composant-Client()*

Table 3.4 – Règle de suppression d'un composant Client

Cette stratégie sera appliquée à la suppression de tout composant de type Client, au niveau application.

3.5 Bilan

Nous avons présenté dans ce chapitre le modèle d'évolution SAEV, que nous proposons pour faire face à la problématique de l'évolution structurelle dans les architectures logicielles. SAEV offre les mêmes concepts pour décrire et gérer l'évolution d'une architecture logicielle au niveau Architectural et au niveau Application. Il s'appuie principalement sur le concept de *stratégie d'évolution* qui encapsule

les *règles d'évolution* permettant de décrire et de gérer l'évolution d'un élément architectural et sur le concept d'*invariant* permettant de vérifier le maintien de la cohérence de l'architecture ayant évolué. SAEV offre aussi un *gestionnaire d'évolution* permettant d'exécuter et de gérer une évolution d'une architecture logicielle.

Dans le tableau suivant, nous positionnons SAEV par rapport aux dimensions de l'évolution architecturale que nous avons défini dans la section 2.2 du chapitre 2 :

Objet de l'évolution	***Type de l'évolution***	***Support de l'évolution***
Composant type **Connecteur type** **Configuration type** **Interface type** **Composant** **Connecteur** **Configuration** **Interface**	**Statique**	**Explicite** (Evolution avec rupture)

Table 3.5 – Positionnement de SAEV sur les trois dimensions de l'évolution

• *Objet de l'évolution* : SAEV permet la spécification et la gestion de l'évolution de tout élément architectural (composant, connecteur, interface, etc.) pouvant être réifié par un ADL, au niveau Architectural et au niveau Application.

• *Type de l'évolution* : nous avons abordé dans SAEV uniquement l'évolution statique des architectures logicielles.

• *Support de l'évolution* : SAEV offre les mêmes opérations d'évolutions et le même mécanisme opératoire pour gérer l'évolution de tout élément architectural. Le modèle SAEV ne permet pas de garder le lien entre l'architecture avant évolution et après évolution.

Le modèle SAEV répond à la majorité des objectifs fixés dans la section 3.2 , en effet :

• Il réifie l'évolution en proposant des concepts pour la décrire et pour la gérer explicitement.

• Il traite l'évolution indépendamment du comportement des éléments architecturaux eux mêmes, donc indépendamment de leurs langages de description.

• Il est uniforme : il offre les mêmes concepts pour décrire et gérer l'évolution de n'importe quel élément architectural au niveau Architectural et au niveau Application.

• Il permet de décrire à la fois l'évolution d'un élément architectural ainsi que la propagation des impacts de cette évolution grâce notamment aux règles d'évolution décrites à l'aide du formalisme E/C/A. Grâce à ce formalisme, nous permettons aussi à une architecture d'être réactive aux évènements d'évolution.

Mais néanmoins deux points ne sont pas considérés jusque là par le modèle SAEV :

– SAEV ne permet pas de répercuter les différentes évolutions d'une architecture logicielle sur le système au cours de l'exécution (l'évolution dynamique). Ce point constitue l'une des perspectives autour du modèle SAEV.

– Nous notons que jusque là, le gestionnaire d'évolution propage les impacts uniquement suivant les règles d'évolution spécifiées par le concepteur. Dans le cas où plusieurs règles d'évolution sont éligibles, le gestionnaire d'évolution est incapable de déterminer automatiquement les règles d'évolution à déclencher pour la propagation des impacts. Il doit à chaque fois consulter le concepteur pour le choix de la propagation. Le concepteur doit alors faire des simulations, pour arriver à une propagation correcte qui maintient la cohérence globale de l'architecture. Cette limite de SAEV est due notamment au fait que la spécification d'une architecture logicielle ne véhicule pas assez d'informations sur les différents liens entre ses éléments architecturaux et qui permettrait à SAEV de déterminer automatiquement les propagations à déclencher, sans avoir recours au concepteur. Nous proposons une réponse à cette limite dans le chapitre suivant.

3.6 Conclusion

Nous avons présenté dans ce chapitre le modèle d'évolution SAEV, que nous proposons pour prendre en compte la problématique de l'évolution structurelle des architectures logicielles. Nous avons mis en avant dans le modèle SAEV la nécessité de dissocier la spécification de l'évolution de la spécification de l'architecture logicielle elle-même. C'est ainsi que SAEV propose des concepts pour la spécification et la gestion de l'évolution d'une architecture logicielle, indépendamment de tout ADL. Nous avons souligné au travers de SAEV, la nécessité de considérer les différents niveaux d'abstraction de description d'une architecture logicielle, et la nécessité de gérer l'évolution au travers de ces différents niveaux d'abstraction de manière uniforme. En effet, dans SAEV, les évolutions d'une architecture logicielle sont spécifiées sur la base des éléments d'un niveau d'abstraction et elles s'appliquent à l'évolution des éléments du niveau inférieur. Ainsi une évolution est spécifiée au niveau Méta pour gérer l'évolution au niveau Architectural, et une évolution est spécifiée au niveau Architectural pour gérer l'évolution au niveau Application. Il est alors nécessaire de distinguer le niveau d'abstraction de l'architecture à faire évoluer. Nous avons mis en avant aussi l'importance de la propagation des impacts pour la sauvegarde de la cohérence globale de l'architecture. Nous avons conclu à la nécessité d'automatiser autant que possible cette phase du processus de SAEV, pour minimiser le recours au concepteur dans les choix des propagations. Pour apporter cette amélioration à SAEV, nous proposons dans le chapitre suivant d'enrichir la description d'architectures logicielles avec plus d'informations sur les liens entre les éléments architecturaux.

CHAPITRE 4

Propriétés sémantiques des connecteurs

4.1 Introduction

Nous avons présenté, dans le chapitre précédent, le modèle SAEV permettant de spécifier et de gérer l'évolution structurelle des architectures logicielles à base de composants. Nous avons souligné au travers de SAEV, la nécessité de propager les impacts de l'évolution d'un élément architectural vers les autres éléments tout en maintenant la cohérence globale de l'architecture ayant évolué. Nous avons montré que cette étape est réalisée dans SAEV au travers de règles d'évolution de type ECA et de leur exécution. Cependant, dans les cas où plusieurs règles d'évolution sont éligibles pour la propagation des impacts, le modèle SAEV ne peut pas déterminer automatiquement la règle d'évolution à déclencher. Le recours au concepteur est alors indispensable pour effectuer ce choix.

Le but de ce chapitre est de répondre à cette limite en permettant à SAEV d'automatiser autant que possible la détermination et la propagation des impacts d'une évolution. Étant donné que la propagation des impacts d'une évolution s'appuie notamment sur les liens existants entre l'élément architectural ayant évolué et son environnement, il parait naturel de les exploiter. Aussi, si ces liens expriment certaines informations quant au degré de corrélation entre les éléments architecturaux, la propagation d'impacts gagnerait en automatisation, réduisant ainsi les interactions avec le concepteur. Dans le cas des architectures logicielles à base de composants, ce lien n'est autre qu'un connecteur. Nous proposons donc d'enrichir ce concept avec des propriétés sémantiques. Évidemment, seuls les ADL qui réifient et qui considèrent le concept de connecteur comme entité de première classe peuvent exploiter ces propriétés sémantiques.

Nous présentons premièrement dans ce chapitre, le rôle des connecteurs dans la propagation des impacts de par leurs positions d'intermédiaires entre les éléments architecturaux. Dans la section 4.3, nous illustrons les propriétés sémantiques que nous proposons pour enrichir le concept de connecteur. Nous présentons dans cette section la description des propriétés sémantiques, la spécification des valeurs de ces propriétés ainsi que les contraintes à respecter lors de la combinaison de ces propriétés au sein d'un même connecteur. Nous montrons ensuite, dans la section 4.3.4 l'apport des propriétés sémantiques pour l'évolution d'une architecture logicielle et notamment pour la propagation des impacts. Nous terminons le chapitre en montrant que les propriétés sémantiques proposées peuvent être également exploitées au niveau méta pour exprimer les liens sémantiques entre les concepts structuraux d'un ADL.

4.2 La propagation d'impacts et les connecteurs

Une évolution sur un élément architectural peut engendrer, outre sa propre évolution, des répercussions sur les autres éléments architecturaux concernés. Ces répercussions doivent être prises en charge, tout en sauvegardant la cohérence de cette architecture. Pour pouvoir déterminer et propager automatiquement, et ce autant que possible, les impacts d'une évolution, SAEV doit disposer dans les descriptions d'architectures logicielles des informations sur le degré de corrélation qui existe entre leurs éléments architecturaux, au-delà de la seule sémantique d'échange de services entre eux. Ces informations qui peuvent concerner non seulement le degré de corrélation mais également la dépendance qui peut exister entre ces éléments architecturaux ou des restrictions particulières qui peuvent concerner l'échange de services entre ces éléments. Par exemple, si un composant fournit ses services pour un composant donné, peut-il les fournir aussi pour les autres composants ? Ou sont-ils réservés à l'usage d'un seul composant ? De la même manière, si un composant qui fournit un service est supprimé, quel serait alors l'impact sur le composant qui requiert ce service ? SAEV doit disposer des informations qui lui permettraient de répondre à ces interrogations.

Nous proposons ainsi de capitaliser et de regrouper les informations que nous jugeons nécessaires à SAEV pour la détermination et la propagation des impacts sous forme de *propriétés sémantiques* que nous associons au concept de connecteur. En effet, nous jugeons que les connecteurs, de par leur position d'intermédiaires entre les éléments architecturaux, et de part leur encapsulation des liens entre ces éléments architecturaux, comme les supports idéaux pour véhiculer les informations sur les différentes corrélations entre ces éléments architecturaux qu'ils relient.

Ainsi, la définition explicite des connecteurs dans une architecture, aide non seulement à la structuration de l'architecture mais peut également aider à décider automatiquement des différents changements à véhiculer entre les éléments architecturaux.

4.3 Propriétés sémantiques pour les connecteurs

Avant de définir les propriétés sémantiques que nous proposons pour enrichir la sémantique du concept **connecteur**, nous précisons que nous ne basons sur aucun ADL particulier et que nous ne nous ferrons aucune hypothèse particulière sur la structure d'un connecteur. Nous adoptons la définition la plus accepté du concept de connecteur (cf. section 1.4.1.2 du chapitre 1) :

Un connecteur est un bloc de constructions utilisé pour modéliser les interactions entre les composants. Un connecteur relie précisément l'interface fournie d'un composant à l'interface requise d'un autre composant .

4.3.1 Définition d'une propriété sémantique

Nous définissons une *propriété sémantique* comme la propriété associée à un connecteur pour exprimer le degré de corrélation ou de dépendance qui existe entre les deux interfaces composant fournie et requise qu'il relie. Nous désignons :

- L'interface composant fournie attachée à un connecteur sous le terme d'*interface source* du connecteur ;
- L'interface composant requise attachée à ce même connecteur sous le terme d'*interface cible* du connecteur.

Nous notons la propriété *P* définie par un connecteur *N* entre l'interface source *Is* et l'interface cible *Ic* par **PN : Is, Ic**.

Pour qu'un connecteur puisse avoir des propriétés sémantiques, celles ci doivent être spécifiées au plus haut niveau d'abstraction, autrement dit sur le concept *connecteur* au *niveau Méta*. Les propriétés sémantiques seront ainsi renseignées à la spécification de l'architecture *au niveau Architectural* et s'appliqueront à l'évolution d'une architecture au *niveau Application*.

En considérant les différents besoins de propagation cités dans la section 4.2 de ce chapitre et en s'inspirant notamment des travaux sur la sémantique des liens de composition et d'héritage entre les classes dans le paradigme objet [50], [45] utilisées pour la propagation des changements dans les schémas de classes, nous proposons les propriétés sémantiques suivantes [72], [84] :

- L'Exclusivité / Partage ;
- La Dépendance / Indépendance ;
- La Prédominance / Non prédominance ;
- La Cardinalité /Cardinalité inverse.

Nous définissons ces propriétés dans les paragraphes suivants.

4.3.2 Description des propriétés sémantiques

Pour chaque propriété sémantique proposée seront présentées :

a. Sa définition ;

b. Ses contraintes de définition

c. Sa formalisation

d. Des exemples illustratifs

e. Des cas particuliers, le cas échéant.

Nous adoptons la représentation graphique suivante pour présenter les exemples illustratifs des propriétés.

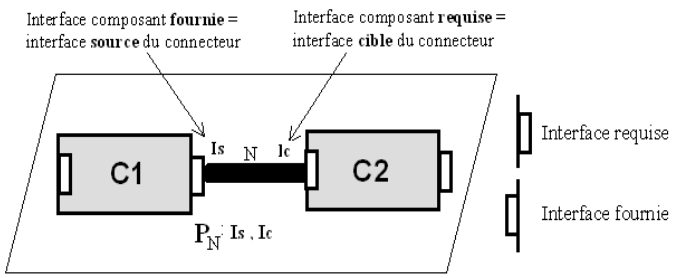

Figure 4.1 – Illustration d'une propriété sémantique

Nous illustrons dans la figure 4.1 une architecture composée de deux composants *C1* et *C2* reliés par un connecteur *N* définissant la propriété **P**.

4.3.2.1 Propriété d'Exclusivité/Partage

La propriété d'exclusivité/partage exprime les contraintes sur les échanges de services entre les interfaces composants reliées par un connecteur.

a. Définition

- **L'Exclusivité** : un connecteur définissant la propriété d'exclusivité (E), dit connecteur *exclusif*, indique que les deux interfaces source et cible de ce connecteur interagissent uniquement entre elles. Ceci implique que l'interface source ne peut fournir ses services que vers la seule interface cible de ce connecteur et l'interface cible ne peut recevoir ses services requis que de la seule interface source de ce connecteur. Ainsi, aucun autre connecteur ne peut être attaché aux interfaces source et cible d'un connecteur exclusif.
- **Le Partage** : à l'inverse, un connecteur définissant la propriété de partage (P), dit connecteur *partagé* indique que les interfaces source et cible de ce connecteur peuvent interagir avec un nombre quelconque d'interfaces. Ainsi, plusieurs connecteurs peuvent être attachés aux interfaces source et cible de ce connecteur.

b. Contraintes de définition de la propriété

Les contraintes associées à la propriété d'exclusivité/partage sont :

(1) Un seul connecteur exclusif au plus peut être relié à une interface ;

(2) Si un connecteur exclusif est attaché à une interface, alors aucun autre connecteur (partagé ou exclusif) ne peut être attaché à cette interface ;

(3) Si un connecteur partagé est attaché à une interface, alors aucun connecteur exclusif ne peut être attaché à cette interface ;

(4) Si un connecteur partagé est attaché à une interface alors plusieurs autres connecteurs partagés peuvent être attachés à cette interface.

c. Formalisation

Nous formalisons les contraintes précédentes associées à la propriété d'exclusivité/partage de la manière suivante. Nous considérons :

E(I) : l'ensemble des connecteurs exclusifs attachés à l'interface I ;

P(I) : l'ensemble des connecteurs partagés attachés à l'interface I ;

Card (X) : le cardinale de l'ensemble X ;

$\forall$ I une interface composant (requise ou fournie)	
- Card(E(I)) $\leq$1	(1)
- Card(E(I)) = 1 $\Rightarrow$ Card(P(I))= 0	(2)
- Card(P(I)) > 0 $\Rightarrow$ Card(E(I))= 0	(3)

d. Exemples illustratifs

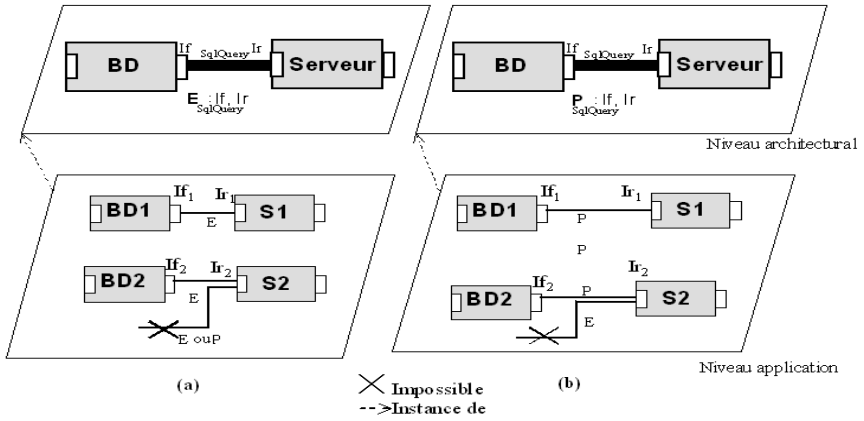

Figure 4.2 – Propriété d'Exclusivité (a) et de Partage (b) : illustration

Nous illustrons dans la figure 4.2, un extrait d'une architecture Client/Serveur décrite au niveau Architectural et au niveau Application. L'architecture est composée de deux composants *BD* et *Serveur* et d'un connecteur *SqlQuery*. Nous présentons deux cas :

(a) Au niveau Architectural, le connecteur *SqlQuery* qui relie l'interface fournie du composant *BD* et l'interface requise du composant *Serveur* est **exclusif**. Ainsi, au niveau Application, une interface fournie d'un composant *BD* ne peut fournir son service que vers une et une seule interface requise d'un composant *Serveur*. De même une interface requise d'un composant *Serveur* ne peut recevoir son service requis que d'une seule interface fournie d'un composant *BD*. Dans l'exemple, aucun autre connecteur ne peut être attaché aux interfaces fournies *If1* et *If2* ni aux interfaces requises *Ir1* et *Ir2* (illustré par un connecteur barré).

(b) Au niveau Architectural, le connecteur *SqlQuery* qui relie l'interface fournie du composant *BD* et l'interface requise du composant *Serveur* est **partagé**. Ainsi, au niveau Application, une interface fournie d'un composant *BD* peut fournir ses services vers les interfaces requises de plusieurs composants *Serveur*. De même, une interface requise d'un composant *Serveur* peut recevoir ses services requis de plusieurs composants *BD*. Plusieurs connecteurs partagés peuvent être attachés à une même interface fournie du composant *BD* ou à une même interface requise d'un composant *Serveur*, mais aucun connecteur exclusif ne peut être attaché à ces interfaces. Dans l'exemple, aucun connecteur exclusif ne peut être attaché aux interfaces fournies *If1* et *If2* ni aux interfaces requises *Ir1* et *Ir2*.

e. Cas particuliers d'Exclusivité/Partage :

La propriété d'Exclusivité /Partage peut être définie par un connecteur dans un seul sens. Un connecteur peut être partagé source uniquement ou exclusif cible uniquement et inversement. Ainsi, nous distinguons deux cas particuliers d'exclusivité/partage : l'*Exclusivité source-Partagé cible* (**Es-Pc**) et *Partagé source-l'Exclusivité cible* (**Ps-Ec**) .

- **Exclusivité source-Partagé cible** : un connecteur définissant la propriété d'exclusivité source-partagé cible (Es-Pc) indique que l'interface source de ce connecteur ne peut fournir son service

qu'à la seule interface cible de ce connecteur. Par contre, l'interface cible de ce connecteur peut interagir avec plusieurs autres interfaces.

- **Partagé source-Exclusivité cible** : un connecteur définissant la propriété de partagé source - exclusif cible (Ps-Ec), indique que l'interface source du connecteur peut interagir avec un nombre quelconque d'interfaces cibles. Ainsi, plusieurs connecteurs peuvent être reliés à cette interface source. L'interface cible de ce connecteur ne peut interagir qu'avec la seule interface source de ce connecteur.

En considérant les différentes règles associées à la propriété d'exclusivité/partage définie précédemment, nous définissons les règles associées aux propriétés d'exclusivité/partage source et cible comme suit :

(**1)** Un connecteur Es-Pc (respectivement Ps-Ec) au plus peut être relié à une interface source (respectivement cible) ;

(2) Si un connecteur Es-Pc (respectivement Ps-Ec) est attaché à une interface source (respectivement cible), alors aucun autre connecteur (partagé ou exclusif) ne peut être attaché à cette interface ;

(3) Si un connecteur Ps-Ec (respectivement Es-Pc) est attaché à une interface source (respectivement cible), alors aucun connecteur exclusif ou Es-Pc (respectivement Ps-Ec) ne peut être attaché à cette interface ;

(4) Si un connecteur Ps-Ec (respectivement Es-Pc) est relié à une interface source (respectivement cible) alors plusieurs autres connecteurs partagés, ou Ps-Ec sources (respectivement Es-Pc) peuvent être attachés à cette interface source (respectivement cible).

Nous formalisons ces règles par les notations suivantes. Nous considérons :

Es-Pc (I) : l'ensemble des connecteurs *Exclusifs source-Patagés cible* attachés à l'interface I ;

Ps-Ec(I) : l'ensemble des connecteurs *Partagés source-Exclusifs cible* attachés à l'interface I ;

$\forall$ I une interface composant fournie	
- Card(Es-Pc(I)) $\leq$1	(1)
- Card(Es-Pc(I)) = 1 $\Rightarrow$ Card(Ps-Ec(I))= 0 Card(E(I))=0 et Card(P(I))=0	(2)
- Card(Ps-Ec(I)) $>$ 0 $\Rightarrow$ Card(E(I))= 0 et Card(P(I))=0	(3)
$\forall$ I une interface composant requise	
- Card(Ps-Ec(I)) $\leq$ 1	(1)
- Card(Ps-Ec(I)) $=$ 1 $\Rightarrow$ Card(Es-Pc(I))= 0 , card(E(I))=0 et Card(P(I))= 0	(2)
- Card(Es-Pc(I)) $>$ 0 $\Rightarrow$ Card(E(I))= 0 card(P(I))=0	(3)

4.3.2.2 Propriété de Dépendance / Indépendance

Cette propriété exprime une dépendance existentielle ou non entre les interfaces composants fournies ou requises reliées par le connecteur.

a. Définitions

- **La Dépendance** : la propriété de dépendance indique qu'il existe une dépendance existentielle de l'interface cible d'un connecteur vis-à-vis de son interface source. Nous distinguons deux types de dépendances : la dépendance forte (DF) et la dépendance faible (Df).
  - *Dépendance forte (DF)* : un connecteur définissant la propriété de *dépendance forte* (DF) indique que l'existence de l'interface cible de ce connecteur est complètement liée à l'existence de son interface source. Ainsi, la suppression de l'interface source de ce connecteur implique la suppression de son interface cible. Cette propriété exprime le fait que le service requis par un composant est fourni impérativement par l'interface fournie du composant auquel il est attaché par ce connecteur de dépendance forte, et telle que la suppression de l'interface fournie de ce composant impliquerai la suppression de l'interface requise à laquelle elle est attachée via ce connecteur.
  - *Dépendance faible(Df)* : un connecteur définissant la propriété de *dépendance faible* (Df) indique que plusieurs autres connecteurs de dépendance faible peuvent être reliés à l'interface cible de ce connecteur. Ainsi, l'existence de l'interface cible est reliée à l'existence de l'ensemble de toutes les interfaces sources de ces connecteurs de dépendance faible. La suppression d'une interface source d'un connecteur de dépendance faible impliquera la suppression de son interface cible si aucun autre connecteur de dépendance faible n'est relié à cette interface. La suppression de toutes les interfaces sources des connecteurs de dépendances faibles reliés à une même interface cible impliquera la suppression de cette interface cible. La dépendance faible exprime le fait qu'un service requis par un composant peut être fourni par plusieurs autres composants. Uniquement lorsque, les interfaces fournies de tous ces composants sont supprimées, que l'interface requise à laquelle elles sont attachées sera aussi supprimée.
- L'**Indépendance (I)** : à l'inverse, un connecteur *indépendant* (I) indique que l'existence de l'interface cible du connecteur est indépendante de l'existence de son interface source.

b. Contraintes de définition de la propriété

Les règles associées à la propriété de dépendance /indépendance sont :

(1) Si un connecteur de dépendance forte est attaché à une interface cible alors aucun autre connecteur de dépendance faible ne peut être attaché à cette interface.

(2) Si un connecteur de dépendance faible est attaché à une interface cible alors aucun connecteur de dépendance forte ne peut être attaché à cette interface.

(3) La suppression de l'interface source du connecteur de dépendance forte implique la suppression automatiquement de l'interface cible de ce connecteur ;

(4) La suppression de l'interface source du connecteur de dépendance faible implique la suppression de l'interface cible de ce connecteur uniquement s'il n'existe pas d'autres connecteurs de dépendance faible attachés à cette interface cible.

(5) La suppression de l'interface source du connecteur d'indépendance n'implique pas la suppression automatiquement de l'interface cible de ce connecteur.

c. Formalisation

Nous formalisons ces propriétés de dépendance/indépendance par les formulations suivantes. Nous considérons :

DF(I) : l'ensemble des connecteurs de dépendance forte reliés à une interface I ;

Df(I) : l'ensemble des connecteurs de dépendance faible reliés à une interface I ;

Ind(I) : l'ensemble des connecteurs Indépendants reliés à une interface I ;

$\forall$ I, une interface composant *requise*

- Card(DF (I))= 1 $\Rightarrow$ Card(Df(I))=0 (1)
- Card(Df(I)) $\geq$ 1 $\Rightarrow$ Card(DF(I))= 0 (2)
- Si $\exists$ N : connecteur, et N $\in$ DF(I), la suppression de source(N) $\Rightarrow$ automatiquement la suppression de la cible(N) ; (3)

- Si $\exists$ N : connecteur, et N $\in$ Df(I), la suppression de source(N) $\Rightarrow$ la suppressionde la cible(N) uniquement s'il n'existe pas de N' : connecteur tel que N' $\in$ Df(I) (4)

- Si $\exists$ N : connecteur et N $\in$ Ind(I), la suppression de la source(N) n'implique pas la suppression de la cible(N) ; (5)

d. Exemples illustratifs

L'exemple ci-dessous illustre la propriété de dépendances forte et faible. Il présente un extrait d'une architecture *Client/Serveur* décrite au niveau Architectural et au niveau Application. L'architecture est composée de deux composants *Client* et *Serveur* et d'un connecteur *RPC*.

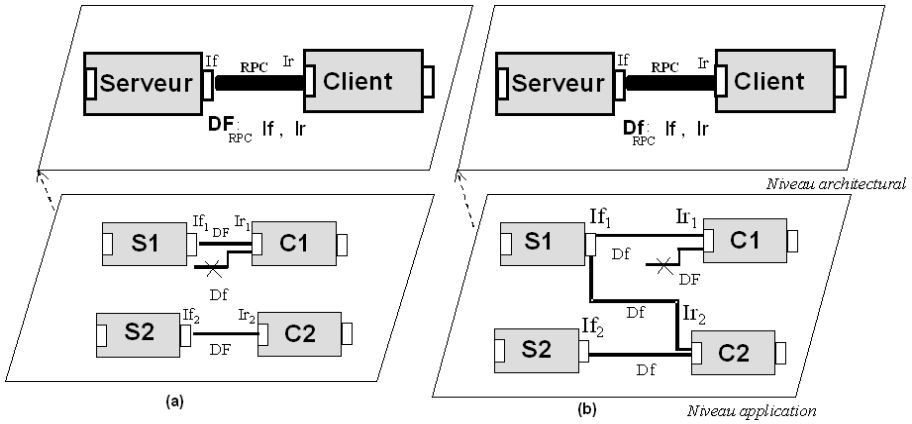

Figure 4.3 – Propriété de Dépendances forte (a) et faible(b) : illustration

(a) Au niveau Architectural, le connecteur *RPC* entre l'interface fournie du composant *Serveur* et l'interface requise du composant *Client* définit la propriété de dépendance forte. Au niveau Architectural, aucun connecteur de dépendance faible ne peut être attaché au composant *Client*.

(b) Au niveau Architectural, le connecteur *RPC* entre l'interface fournie du composant *Serveur* et l'interface requise du composant *Client* définit la propriété de dépendance faible. A niveau Architectural, aucun connecteur de dépendance forte ne peut être attaché au composant *Client*.

Nous illustrons l'évolution de l'application **Client/Serveur** de la figure précédente (cas a et b) par la suppression de l'interface *If1* du niveau Application (figure 4.4).

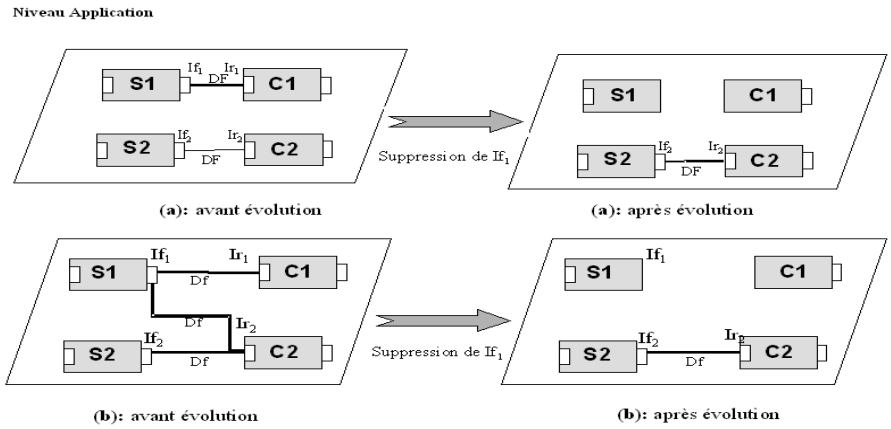

Figure 4.4 – Propriété de Dépendances forte (a) et faible(b) à l'évolution

a) La suppression de l'interface fournie d'un composant *Serveur* implique automatiquement la suppression de l'interface requis d'un composant *Client*. Ainsi, la suppression de l'interface *If1* implique la suppression de l'interface requise *Ir1*.

b) La suppression d'une interface fournie d'un composant *Serveur* n'implique pas la suppression automatique de l'interface requise d'un composant *Client* auquel elle est reliée, s'il existe d'autres connecteurs qui relient cette interface à d'autres interfaces fournies. Dans notre exemple, la suppression de l'interface *If1* n'implique pas la suppression de l'interface *Ir2*, car il existe un autre connecteur de dépendance faible attaché à cette interface.

e. Cas particulier de dépendance forte

Nous pouvons contraindre plus la propriété de dépendance forte pour apporter plus de précision sur la dépendance forte entre les deux interfaces composants reliées par un connecteur. Nous distinguons pour cela deux cas particuliers de dépendance forte : la *Dépendance forte unique* (**DFu**) et la *Dépendance forte multiple* (**DFm**).

- *Dépendance forte unique (DFu)* : indique que l'interface cible du connecteur dépend fortement et *uniquement* de la seule interface source de ce connecteur. Ainsi, si d'autres connecteurs sont reliés à cette interface cible, tous ces connecteurs doivent être indépendants.
- *Dépendance forte multiple (DFm)* : indique que l'interface cible de ce connecteur dépend fortement des interfaces sources de plusieurs connecteurs. Ainsi, la suppression de l'une de ces interfaces sources implique la suppression de l'interface cible de ce connecteur.

Par défaut la *Dépendance Forte* d'un connecteur est *multiple*, c'est à dire qu'une interface cible peut dépendre de plusieurs autres interfaces sources. Les règles associées à ces deux cas particuliers sont :

(1) Un seul connecteur de *dépendance forte unique* au plus peut être relié à une interface cible ;

(2) Si un connecteur définit la propriété de *dépendance forte unique*, alors aucun connecteur définissant la propriété de dépendance forte multiple ou de dépendance faible ne peut être attaché à l'interface cible de ce connecteur ;

(3) Si un nombre quelconque de connecteurs de *dépendance faible* sont attachés à une interface cible, aucun autre connecteur de dépendance forte multiple ou de dépendance forte unique ne peut être relié à cette interface ;

Nous formalisons ces règles par les notations suivantes. Nous considérons :
DFu (I) : l'ensemble des connecteurs de dépendance forte unique attachés à I ;
DFm (I) : l'ensemble des connecteurs de dépendance forte multiple attachés à I ;

$\forall$ I, une interface composant *requise*	
- Card(DFu (I))$\leq$ 1	(1)
- Card(DFu (I))= 1 $\Rightarrow$ Card(Df(I))= 0 et Card(DFm (I))=0	(2)
- Card(Df(I)) > 0 $\Rightarrow$ Card(DFu (I))= 0 et Card(DFm (I))= 0	(3)

4.3.2.3 Propriété de Prédominance/Non Prédominance

La propriété de prédominance indique s'il existe ou non une dépendance existentielle de l'interface source d'un connecteur vis à vis de son interface cible.

a. Définitions

- **Prédominance** : cette propriété exprime une dépendance existentielle, le sens de cette dépendance existentielle est symétrique à celui de la propriété de dépendance. En effet, la prédominance met l'accent sur le fait que l'existence d'une interface source d'un connecteur est liée à l'existence de son interface cible, alors que la dépendance exprime le fait que l'existence de l'interface cible dépend de l'existence de l'interface source de ce connecteur. Nous distinguons deux types de prédominance : la prédominance forte (PRF) et la prédominance faible (PRf) :

  – *Prédominance forte(PRF)* : un connecteur définissant la propriété de *prédominance forte* (PRF), indique que l'existence de l'interface source du connecteur est fortement liée à celle de l'interface cible de ce connecteur. Ainsi, la suppression de cette interface cible implique la suppression de l'interface source de ce connecteur et la création d'une interface source implique automatiquement la création de l'interface cible à laquelle elle sera attachée. Cette propriété exprime le fait qu'une interface fournie est spécifiquement créée pour fournir le service requis par un autre interface composant.

  – *Prédominance faible(PRf)* : un connecteur définissant la propriété de *prédominance faible* (PRf) indique que plusieurs autres connecteurs peuvent être attachés à l'interface source de ce connecteur. Tous ces connecteurs doivent être de prédominance faible. L'existence de l'interface source est reliée à l'existence de l'ensemble de toutes les interfaces cibles de ces connecteurs. La suppression d'une interface cible d'un connecteur de prédominance faible impliquera la suppression de son interface source si aucun autre connecteur de prédominance faible n'est relié à cette

interface. La suppression de toutes les interfaces cibles des connecteurs de dépendance faible reliés à une même interface source impliquera la suppression de cette interface source.

- **Non prédominance** : la propriété de *non prédominance* (NPR) indique que la suppression d'une interface cible d'un connecteur n'entraîne pas la suppression de son interface source.

b. Contraintes de définition de la propriété

(1) Un seul connecteur de prédominance forte peut être attaché à une interface source.

(2) Si un connecteur de *prédominance forte* est attaché à une interface fournie, alors aucun connecteur de *prédominance faible* ne peut être attaché à cette interface.

(3) Si un connecteur de *prédominance faible* est attaché à une interface fournie, alors aucun connecteur de *prédominance forte* ne peut être attaché à cette interface.

(4) La suppression de l'interface cible du connecteur de *prédominance forte* implique la suppression de l'interface source de ce connecteur ;

(5) La suppression de l'interface cible du connecteur de *prédominance faible* implique la suppression de l'interface cible de ce connecteur, uniquement s'il n'existe pas d'autres connecteurs de *prédominance faible*, attaché à cette interface source.

(6) La suppression de l'interface cible du connecteur de *Non Prédominance* n'implique pas la suppression de l'interface cible de ce connecteur.

c. Formalisation

Nous formalisons ces propriétés de prédominance et non prédominance par les notations suivantes. Nous considérons

PRF(I) : l'ensemble de connecteurs de prédominance forte reliés à l'interface I ;
PRf(I) : l'ensemble de connecteurs de prédominance faible reliés à l'interface I ;

$\forall$ I, une interface composant *fournie*

- Card(PRF(I)) ≤ 1, (1)
- Card(PRF(I))= 1 $\Rightarrow$ Card(PRf(I))=0, (2)
- Card(PRf(I)) > 0 $\Rightarrow$ Card(PRF(I))= 0, (3)
- Si $\exists$ N : connecteur et N $\in$ PRF(I) la suppression de cible(N) $\Rightarrow$ automatiquement la suppression source(N) (4)
- Si $\exists$ N : connecteur et N $\in$ PRf(I) la suppression de cible(N) $\Rightarrow$ la suppression de la source(N) uniquement s'il n'existe pas de N' : connecteur tel que N' $\in$ PRf(I) (5)
- Si $\exists$ N : connecteur et N $\in$ NPR(I), la suppression de cible(N) n'implique pas la suppression de la source(N) ; (6)

d. Exemples illustratifs

Nous reprenons l'extrait de l'architecture *Client/Serveur* de la section 3.1.1.

a) Au niveau Architectural le connecteur *SqlQuery* définit la propriété de prédominance forte entre

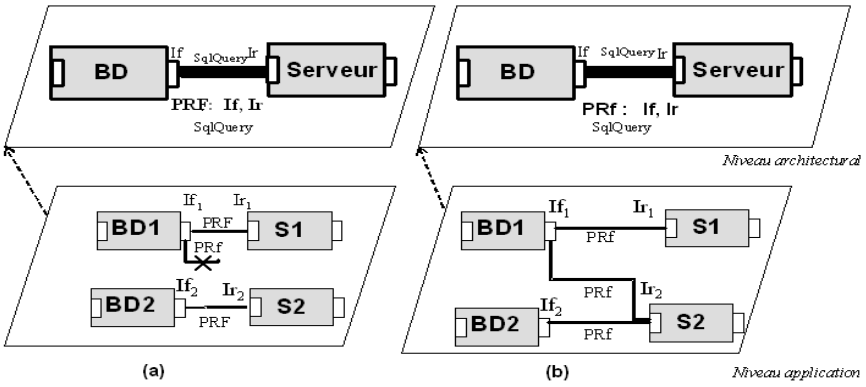

Figure 4.5 – Propriété de Prédominances forte (a) et faible (b) : illustration

l'interface requise du composant *Serveur Ir* et l'interface fournie du composant *BD If*. Au niveau Application aucun connecteur de prédominance faible ne peut être attaché à un composant *BD*.

b) Au niveau Architectural, le connecteur *SqlQuery* définit la propriété de prédominance faible entre l'interface requise du composant *Serveur* et l'interface fournie du composant *BD*. Au niveau Application, aucun connecteur de prédominance forte ne peut être attaché à un composant *BD*.

Nous illustrons l'évolution de l'application, **Client/Serveur** de la figure précédente (dans les deux cas a et b) par la suppression de l'interface *Ir1* ou *Ir2* au niveau Application (figure 4.6).

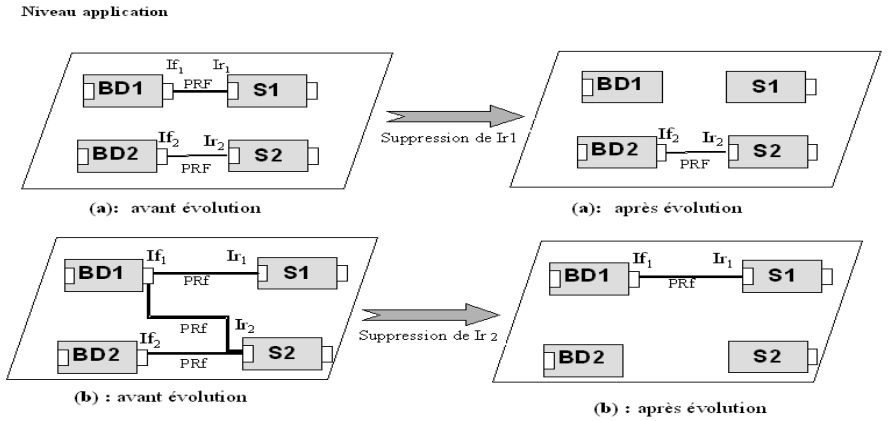

Figure 4.6 – Propriété de Prédominances forte (a) et faible (b) à l'évolution

a) Au niveau Application, la suppression d'une interface requise d'un composant *Serveur* implique automatiquement la suppression de l'interface fournie du composant *BD* (la suppression de *Ir1* implique la suppression de *If1* et la suppression de *Ir2* implique la suppression de *If2*).
b) Au niveau Application la suppression d'une interface requise d'un composant *Serveur* n'implique pas la suppression de l'interface fournie du composant *BD*, s'il existe d'autres connecteurs de dépendance faible qui relient cette interface vers d'autres interfaces fournies. Dans l'exemple, la suppression de *Ir2* implique la suppression automatique de *If2* mais n'implique pas la suppression

de *If1*.

Nous notons, même si les deux propriétés de *dépendance* et de *prédominance* invoquent la dépendance existentielle entre les interfaces composant, les sémantiques véhiculées sont différentes. La propriété de *dépendance* exprime le fait qu'une interface requise dépend d'un ou plusieurs autres services fournis par d'autres interfaces. La propriété de *prédominance* exprime le fait qu'une interface fournie existe juste pour servir une ou plusieurs autres interfaces requises.

4.3.2.4 Cardinalité et Cardinalité inverse

a. Définitions

La *Cardinalité* d'un connecteur indique le nombre d'instances de l'interface cible de ce connecteur qui peuvent être attachées à une même instance de l'interface source de ce connecteur (autrement dit, le nombre d'instances de ce connecteur qui peuvent être attachées à une même instance d'interface source). La *Cardinalité inverse* d'un connecteur indique le nombre d'instances de l'interface source qui peuvent être attachées à une même instance de l'interface cible de ce connecteur (autrement dit, le nombre d'instances de ce connecteur qui peuvent être attachées à une même instance de l'interface cible). La cardinalité et la cardinalité inverse peuvent être exprimées sous forme d'une valeur fixe ou sous forme d'un intervalle en indiquant la *valeur minimale* et la *valeur maximale* de cet intervalle.

Lorsque la cardinalité (respectivement cardinalité inverse) est fixe, cela indique que le nombre d'instances de l'interface cible (respectivement de l'interface source) du connecteur qui doivent être reliées à une même instance source (respectivement cible) du connecteur est toujours le même.

Lorsque la cardinalité et la cardinalité inverse sont exprimées par un intervalle, cela indique que le nombre d'instances de l'interface cible (respectivement de l'interface source) du connecteur qui doivent être reliées à une même instance source (respectivement cible) du connecteur varie entre le minimum et le maximum de cet intervalle. Si le maximum n'est pas connu, cette valeur peut être remplacé par [*]. Ces propriétés permettent ainsi de contraindre le nombre d'instances de connecteurs attachés à une même instance d'interface source ou cible au cours de l'évolution.

b. Contraintes de définition de la propriété

- si un connecteur définit une cardinalité (respectivement cardinalité inverse) fixe égale à *n* (n$\geq$ 1) alors à une instance de l'interface source (respectivement cible) de ce connecteur sont reliées exactement *n* instances de l'interface cible (respectivement source) de ce connecteur via les instances de ce connecteur ;
- si un connecteur définit une cardinalité (respectivement cardinalité inverse) variable de [Min, Max] alors à une instance de l'interface source (respectivement cible) de ce connecteur est relié un nombre d'instances de l'interface cible (respectivement cible) de ce connecteur compris dans l'intervalle [Min, Max] ;

c. Exemple illustratif

Dans la figure suivante nous illustrons un exemple d'un connecteur *RPC* entre l'interface fournie du composant *Serveur* et l'interface requise du composant *Client* qui définit une *cardinalité variable* de [1,10] et une *cardinalité inverse* égale à 1.

Cette cardinalité et cardinalité inverse indiquent qu'au niveau Application, il est possible d'attacher au maximum 10 instances d'interfaces du composant Client à une même instance d'interface du compo-

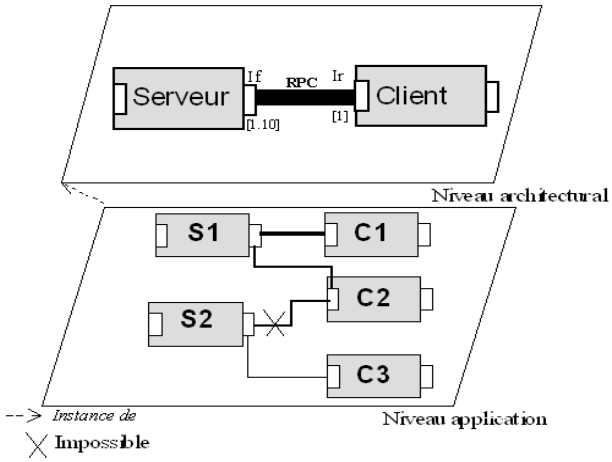

Figure 4.7 – Cardinalité et Cardinalité inverse : illustration

sant Serveur, par contre une instance de l'interface composant Client est reliée uniquement à une seule instance de l'interface du composant Serveur.

Autrement dit, il est possible d'attacher 10 instances du connecteur RPC à une même instance de l'interface du composant Serveur, par contre une seule instance du connecteur RPC peut être attachée à une instance de l'interface du composant Client.

Après avoir donné les définitions des différentes propriétés sémantiques que nous proposons pour le concept connecteur, nous illustrons dans la section suivante la spécification des valeurs de ces propriétés au moment de la spécification de l'architecture ainsi que les restrictions à respecter lors de cette spécification.

4.3.3 Spécification des valeurs des propriétés sémantiques

Nous avons montré qu'en plus de sa sémantique fonctionnelle d'intermédiaire entre les composants, un connecteur peut définir les propriétés sémantiques suivantes :

- *Exclusivité / Partage* : il peut être ainsi soit :
  - Exclusif (E),
  - Partagé(P),
  - Exclusif source-Partagé cible (Es-Pc),
  - Partagé source-Exclusif cible (Ps-Ec) ;
- Dépendance /Indépendance : il peut être ainsi :
  - de dépendance forte unique (DFu),
  - dépendance forte multiple (DFm),
  - dépendance faible (Df),
  - indépendant (I) ;
- Prédominance/Non Prédominance : il peut être ainsi :
  - de prédominance forte (PRF),

  - de prédominance faible (PRf),
  - non prédominant(NPR) ;
- Une cardinalité et cardinalité inverse fixe ou variable.

Nous présentons dans ce qui suit les valeurs par défauts des propriétés sémantiques d'un connecteur (au niveau Méta), puis nous illustrons les différentes restrictions à respecter lors de la spécification des propriétés d'un connecteur (au niveau Architectural).

4.3.3.1 Valeurs par défauts des propriétés

Nous considérons que, par défaut, un connecteur ne véhicule aucune restriction entre les interfaces composant qu'il relie, ainsi un connecteur par défaut est :

- Partagé ;
- Indépendant ;
- Non prédominant ;
- A une cardinalité et une cardinalité inverse variable de [*].

Ces valeurs s'interprètent comme suit :

- Une interface source d'un connecteur (interface composant fournie) peut fournir ses services vers une ou plusieurs autres interfaces cibles (interfaces composants requises) ;
- Une interface cible d'un connecteur (interface composant requise) peut recevoir son service requis d'une ou plusieurs autres interfaces sources (interfaces composants fournies).
- La suppression d'une interface source d'un connecteur n'implique pas la suppression de son interface cible ;
- La suppression de l'interface cible du connecteur n'implique pas la suppression de son interface source.

4.3.3.2 Restrictions associées aux valeurs des propriétés sémantiques

Plusieurs combinaisons de ces propriétés sémantiques peuvent être spécifiées au sein d'un même connecteur. Par exemple, un connecteur peut définir à la fois la propriété de *dépendance forte* et la propriété de *prédominance forte*. Ainsi, la suppression de l'interface source du connecteur implique la suppression de son interface cible et la suppression de l'interface cible de ce connecteur implique la suppression de son interface source.

Certaines propriétés combinées ensemble au sein d'un connecteur ont des influences entre elles, imposant ainsi des restrictions à considérer au moment de la définition des propriétés d'un connecteur. Par exemple, il existe une influence directe entre les propriétés d'exclusivité/partage et de cardinalité/cardinalité inverse, étant donnée qu'elles sont relatives toutes les deux aux nombres d'interfaces qui peuvent interagir avec une autre interface.

Nous illustrons dans ce qui suit les différentes influences entre les propriétés, en distinguant les restrictions : entre la propriété d'exclusivité/partage et les propriétés de cardinalité/cardinalité inverse, de celles entre les propriétés d'exclusivité/partage et les autres propriétés (les influences entre les propriétés de cardinalité/cardinalité inverse vis-à-vis des autres propriétés sont toujours déductibles à partir des autres restrictions) :

a. Entre les propriétés d'exclusivité/partage et la cardinalité /cardinalité inverse :

Les influences entre les propriétés d'exclusivité/partage (E, Es-Pc, Ps-Ec, P) et la cardinalité/cardinalité inverse sont illustrées par le tableau suivant :

Propriétés	**Cardinalité**	**Card-inverse**
Exclusif(E)	1	1
Partagé(P)	2..*	2..*
Exclusif source-Partagé cible (Es-Pc)	1	2..*
Partagé source-Exclusif cible (Ps-Ec)	2..*	1

Table 4.6: Restrictions liées aux propriétés d'Exclusivité/Partagé et de cardinalités

A la définition des propriétés d'un connecteur, les restrictions suivantes s'imposent :

- Un connecteur exclusif (E ou Es-Pc) a une cardinalité toujours égale à 1 ;
- Un connecteur exclusif (E ou Ps-Ec) a une cardinalité inverse égale toujours à 1 ;
- Un connecteur partagé (P ou Ps-Ec) a une cardinalité minimale supérieure à 1 ;
- Un connecteur partagé (P ou Es-Pc) a une cardinalité inverse minimale supérieure à 1 ;

b. Entre les propriétés d'exclusivité/partage et les autres propriétés :

Il existe des influences entre la propriété d'exclusivité/partage (E, P, Es-Pc, Ps-Ec) et les propriétés de dépendance/indépendance (DFu, DFm, Df, I) et de prédominance /non prédominance (PRF, PRf et NRP) et, par conséquent entre ces propriétés et la cardinalité/cardinalité inverse. Nous résumons ces différentes influences dans le tableau suivant :

Propriétés		**Exclus./Partage**				**Depend./Independ.**				**Prédom./non prédom.**		
Exclus./Partage		E	P	Es-Ps	Ps-Ec	DFu	DFm	DRf	I	PRF	PRf	NRP
	E					√	×	×	√	√	×	√
	P					√	√	√	√	√	√	√
	Es-Pc					√	√	×	√	√	×	√
	Ps-Ec					√	×	√	√	√	×	√
Dépend./Indép.	**DFu**									√	√	√
	DFm									√	√	√
	DRf									√	√	√
	I									√	√	√

Table 4.7: Influences entre les propriétés sémantiques

$\surd$: la combinaison est possible
$\times$: la combinaison est impossible.

Le tableau résume les différentes restrictions entre les propriétés sémantiques (les cases vides peuvent être obtenues par symétrie). Nous distinguons trois groupes de restrictions :

1. Entre les propriétés d'exclusivité/partage et de dépendance/ indépendance :
   - Il est impossible de combiner la propriété d'exclusivité (E) ou de partagé source-exclusivité cible (Ps-Ec) avec la propriété de dépendance faible (Df). Un connecteur de dépendance faible doit être partagé ou partagé source.
   - Il est impossible de combiner la propriété d'exclusivité(E) ou partagé source-exclusivité cible (Ps-Ec) avec la propriété de dépendance forte multiple (DFm). Un connecteur de dépendance forte multiple doit être partagé ou partagé cible.
2. Entre les propriétés d'exclusivité/partage et les propriétés de prédominance/non prédominance :
   - Il est impossible de combiner la propriété d'exclusivité (E) ou d'exclusivité source- partagé cible (Es-Pc) avec la propriété de prédominance faible (PRf). Un connecteur de prédominance faible doit être partagé ou partagé cible ;
3. Entre les propriétés de dépendance/indépendance et de prédominance/non prédominance :
   - Aucune contrainte entre ces propriétés, toutes les combinaisons sont possibles.

Les restrictions concernant la définition de ces propriétés ainsi que leurs combinaisons seront vérifiées à la spécification d'un connecteur au niveau Architectural. En dehors des cinq combinaisons interdites toutes les autres restent possibles au sein d'un même connecteur.

Après avoir présenté la sémantique de chaque propriété et les différentes restrictions à leurs combinaisons, nous illustrons dans ce qui suit comment ces propriétés interviennent à l'évolution d'une architecture logicielle.

4.3.4 Prise en compte des propriétés sémantiques à l'évolution d'une Application

Nous rappelons que nous avons proposé d'enrichir le concept de connecteur par des propriétés sémantiques, afin de pouvoir les exploiter pour déterminer et propager automatiquement autant que possible les impacts d'une évolution au sein d'une architecture. Nous rappelons aussi que ces propriétés sémantiques sont définies à la spécification des connecteurs au niveau Architectural et elles s'appliquent au niveau Application.

En effet, les propriétés sémantiques proposées interviennent au niveau application, à trois moments à :

- l'*ajout* d'un *connecteur* entre deux *interfaces* ;
- la *suppression* d'une *interface composant* (fournie ou requise) ;
- l'*ajout* d'une *interface composant* fournie

Suite à ces trois opérations et selon les conditions qui sont satisfaites, d'autres opérations peuvent être invoquées pour propager les impacts d'une évolution. Nous résumons ces différents cas dans le tableau suivant. Pour la présentation du tableau :

- Nous considérons à titre d'illustration une application composée de deux composants *C1* et *C2*. *C1* est décrit par une interface fournie *If* et *C2* par une interface requise *Ir*.

- Pour chaque propriété sémantique, nous indiquons les opérations d'évolution pour lesquelles elle intervient et sous quelles conditions ainsi que les impacts de ces opérations d'évolution à déduire en considérant cette propriété sémantique.

Propriétés	Opération d'évolution	Conditions	Impacts
Exclus./Partage	**Ajout** d'un **connecteur** *N* : *If*, *Ir* (instantiation de *N* à partir du niveau Architectural)	*N* est exclusif (**E**) Il existe un connecteur attaché à If et/ou à Ir	Ajout impossible
		N est partagé (**P**) et Il existe un connecteur exclusif (**E**, **Es-Pc** ou **Ps-Ec**) attaché à *If* et/ou à *Ir*	Ajout de *N* impossible
		N est **Es-Pc** et Il existe un connecteur attaché à *If* et/ou un connecteur exclusif (**E** ou **Es-Pc**) attaché à *Ir*	Ajout de *N* impossible
		N est **Ps-Ec** Il existe un connecteur attaché à *Ir* et/ou un connecteur exclusif (**E**, **Es-Pc**) attaché à *If*	Ajout de *N* impossible
Dépend./Independ.	**Ajout** d'un **connecteur** *N* : *If*, *Ir* (instantiation de *N* à partir du niveau Architectural)	*N* est de **DFu** et Il existe un connecteur de **DFm** ou **Df** attaché à *Ir*	Ajout de *N* impossible
		N est de **DFm** ou **Df** et Il existe un connecteur de **DFu** attaché à *Ir*	Ajout de *N* impossible
	Suppression d'une **interface composant fournie** (*If*)	Il existe un connecteur de **DFu** ou **DFm** attaché à cette interface	Suppression de l'interface composant *cible* de ce connecteur
		Si il existe un connecteur de **Df** attaché à cette interface et il n'existe pas d'autres connecteurs de **Df** attachés à l'interface cible de ce connecteur	Suppression de l'interface composant *cible* de ce connecteur
Prédom./non prédom.	**Suppression** d'une **interface composant requise** *Ir*	Si le connecteur attaché à cette interface est **PRF**	Suppression de l'interface composant *source* de de ce connecteur
		Si le connecteur attaché à cette	Suppression de l'interface

... suite page suivante...

Propriétés	Opération d'évolution	Conditions	Impacts
		interface est de **PRf** et il n'existe pas d'autres connecteurs de **PRf** attachés à l'interface source de ce connecteur	source de ce connecteur
	Ajout d'une **interface composant fournie**	Si le connecteur type attaché à l'interface type de cette interface est de **PRF**	Ajout instance connecteur type et Ajout interface *cible* du connecteur ajouté
Card./card-inverse	**Ajout** d'un **connecteur** entre deux interfaces	Si le nombre d'interfaces reliées à l'interface source (respectivement cible de ce connecteur a atteint la card. (respectivement card-inverse) maximale	Ajout impossible

Table 4.8: Prise en compte des propriétés sémantiques à l'évolution

Nous avons illustré dans le tableau précédent à quel moment intervient chaque propriété sémantique lors de l'évolution d'une application. Nous distinguons trois cas :

- A l'*ajout* d'un *connecteur* entre deux interfaces, toutes les propriétés peuvent intervenir. Les propriétés sémantiques associées au connecteur à ajouter et les propriétés des autres connecteurs attachés aux interfaces auxquelles sera attaché le connecteur ajouté détermineront si l'ajout est possible ou non.
- A la *suppression* d'une *interface composant fournie*, la propriété de dépendance intervient ;
- A la s*uppression* d'une *interface composant requise* et/ou à l'*ajout* d'une *interface composant requise*, la propriété de prédominance intervient.

Les propriétés sémantiques proposées doivent être prise en compte par le modèle SAEV, lors de la détermination et la propagation des impacts d'une évolution. En effet, les restrictions liées à la définition des propriétés sémantiques doivent être ajoutées au *invariants* du concept *connecteur*. Les différents cas d'intervention des propriétés sémantiques présentés dans le tableau précédent seront décrits au niveau des *règles d'évolution* associées aux connecteurs. Ainsi selon l'événement intercepté et selon les valeurs de ces propriétés, SAEV déterminera les propagations à déclencher. Des exemples de spécifications de règles d'évolution et d'invariants considérants les propriétés sémantiques sont présentés dans le chapitre suivant (cf. section 5.5).

Nous avons proposé dans ce chapitre d'enrichir la sémantique des connecteurs par un ensemble de propriétés sémantiques pour automatiser autant que possible la détermination et la propagation des impacts d'une évolution. Nous avons montré que ces propriétés sont renseignées à la spécification d'une architecture au niveau Architectural et s'appliquent au niveau Application. Nous montrons dans la section suivante que ces propriétés peuvent être exploitées et spécifiées au niveau Méta pour améliorer

encore plus la propagation automatique des impacts à l'évolution d'une architecture logicielle sur les deux niveaux d'abstraction.

4.4 Propriétés sémantiques au niveau Méta : pourquoi et comment ?

Pour augmenter encore plus l'automatisation de la propagation des impacts et avec la même analogie qu'au niveau Architectural, nous proposons d'utiliser les propriétés sémantiques proposées au niveau *Méta* pour exprimer le degré de corrélation entre les concepts proposés par un ADL. Ceci nous permettra à l'évolution d'une instance d'un concept au niveau Architectural ou au niveau Application de déterminer automatiquement les impacts sur les instances des autres concepts. Par exemple, si un composant est supprimé au niveau Architectural, peut-on déclencher automatiquement la suppression de son interface ? Ou encore si l'interface fournie d'un composant est supprimée, peut-on déduire automatiquement la suppression du composant ?

Pour répondre à cette préoccupation, nous proposons de réifier les liens structuraux proposés par les ADL pour relier les différents concepts (liens d'association de composition d'héritage, etc), autant que connecteurs de premières classe que nous désignons par connecteurs Niveau-Méta. Les propriétés sémantiques proposées peuvent être ainsi associées à ces connecteurs Niveau-Méta.

4.4.1 Connecteur Niveau-Méta

Nous considérons un connecteur Niveau-Méta comme un connecteur particulier reliant deux concepts d'un ADL au niveau méta (la source et la cible de ce connecteur sont des **concepts**). Un connecteur Niveau-Méta peut être tout lien structurel reliant deux concepts au niveau *méta*, lien d'*héritage*, de *composition*, d'*association* ou tout autre lien proposé par un ADL pour relier ses concepts.

Nous illustrons un exemple de connecteur de *composition* entre le concept *Composant* et le concept *Interface* dans la figure suivante.

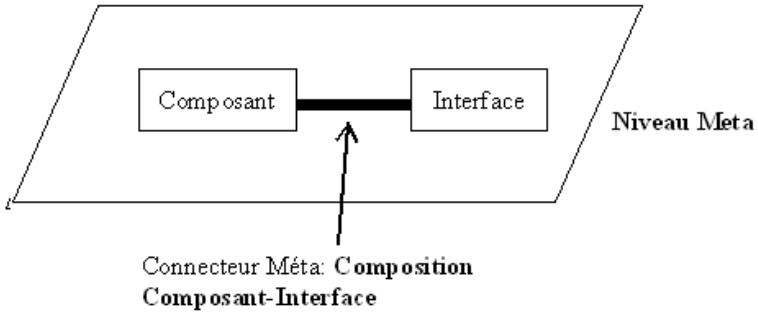

Figure 4.8 – Connecteur de composition Composant-Interface

4.4.2 Propriétés sémantiques des connecteurs Niveau-Méta

Nous associons les mêmes propriétés sémantiques proposées aux connecteurs Niveau Méta : l'Exclusivité/Partage, la Dépendance/Indépendance, Prédominance /Non prédominance, Cardinalité et Cardinalité inverse. Ces propriétés sémantiques seront ainsi renseignées à la spécification des concepts d'un

ADL au niveau méta et interviendront à l'évolution d'une architecture au niveau Architectural et/ ou au niveau Application.

Les valeurs par défaut des propriétés sémantiques des connecteurs Niveau-Méta dépendent de l'ADL considéré au niveau *Méta*. Les valeurs de ces propriétés ainsi définies au niveau Méta interviendront à l'évolution d'une architecture du niveau Architectural et au niveau Application. A titre d'exemple, en définissant le connecteur de composition entre le concept **composant** et le concept **interface** comme étant de *dépendance forte unique*, la suppression d'un composant (*source* du connecteur de composition) déclenche automatiquement la suppression de l'interface de ce composant (*cible* du connecteur de composition).

Les valeurs des propriétés sémantiques des connecteurs de *Niveau-Méta* peuvent être redéfinies au niveau *Architectural*. Ces valeurs du niveau *Architectural* peuvent ainsi intervenir à l'évolution d'une application. A titre d'exemple, en définissant la propriété de *prédominance forte* sur un connecteur de composition entre un composant type *Client* et son service *S1*, la suppression d'une instance de ce Service *S1* au niveau *Application* implique automatiquement la suppression de l'instance du composant *Client* auquel elle appartient.

Les connecteurs Niveau-Méta ainsi que leurs propriétés sémantiques, seront abordés avec plus de détail dans le chapitre suivant dans le cadre de l'ADL COSA [80].

4.5 Conclusion

Nous avons mis en avant dans ce chapitre le rôle que nous associons aux connecteurs pour la propagation automatique des impacts d'une évolution. En effet, l'évolution d'un élément Architectural peut provoquer des impacts sur les autres éléments concernés de l'architecture. Il est alors nécessaire de propager ces impacts tout en sauvegardant la cohérence globale de cette architecture. Nous avons souligné alors la nécessité d'automatiser autant que possible cette propagation d'impacts.

Pour répondre à cette préoccupation, nous avons proposé d'enrichir les connecteurs par un ensemble de propriétés sémantiques qui véhiculent des informations sur le degré de corrélation entre les éléments architecturaux. Ces propriétés sémantiques sont : l'Exclusivité /Partage, la Dépendance/Indépendance, la Prédominance/Non prédominance, la Cardinalité et Cardinalité inverse. Ainsi, à l'évolution d'un élément architectural, les valeurs des propriétés sémantiques des connecteurs qui le relient aux autres éléments architecturaux permettront de déterminer et de propager les impacts de cette évolution vers ces éléments architecturaux.

Nous avons montré que les propriétés sémantiques des connecteurs sont renseignées à la spécification de ces connecteurs au niveau Architectural et elles interviennent à l'évolution de toute application construite à partir de ce niveau Architectural. Nous avons proposé ensuite d'exploiter ces mêmes propriétés au niveau méta et cela en réifiant les différents liens entre les concepts structuraux d'un ADL comme étant des connecteurs de premières classes, que nous désignons sous le terme de connecteurs Niveau-Méta. Un connecteur Niveau-Méta est ainsi un connecteur reliant deux concepts d'un ADL. Les propriétés sémantiques définies pour les connecteurs Niveau-Méta, permettront d'augmenter l'automatisation de la détermination et de la propagation des impacts de l'évolution d'une architecture au niveau Architectural et au niveau Application.

Les propriétés sémantiques proposées ont été définies jusque la, sans tenir compte d'aucun ADL particulier. Dans le chapitre suivant nous validons et illustrons ces propriétés sémantiques dans le cadre

de l'ADL COSA [76] en tenant compte des spécificités particulières de ses connecteurs. Nous illustrons aussi la prise en compte de ces propriétés par le modèle d'évolution SAEV.

CHAPITRE 5

Illustration au travers de COSA

5.1 Introduction

Nous présentons dans ce chapitre une illustration de nos propositions dans le cadre de l'ADL COSA (Component-Object based Software Architecture)[42] [76]. Nous illustrons premièrement, l'extension de l'ADL COSA par les propriétés sémantiques proposées dans le chapitre 4. Nous présentons ensuite une application du modèle d'évolution SAEV, proposé dans le chapitre 3, sur une architecture et sur une application *Client/Serveur* décrite au travers de l'ADL COSA enrichi par les propriétés sémantiques.

Les propriétés sémantiques que nous avons proposées ont été définies jusque là sans tenir compte d'aucun ADL en particulier. Afin de valider ces propriétés sémantiques et de faire ressortir leur intérêt et leur importance pour la gestion et la propagation des impacts d'une évolution, nous les appliquons à un ADL précis, l'ADL COSA. Dans la première partie de ce chapitre, nous montrons comment ces propriétés sémantiques seront intégrées dans l'ADL COSA en tenant compte des spécificités et de la structure particulière des connecteurs de cet ADL. Nous illustrons ensuite une spécification d'une architecture *Client/Serveur* en COSA, tout en tenant compte des propriétés sémantiques proposées. Dans la dernière partie, nous exploitons cette spécification de l'architecture Client/Serveur en COSA pour montrer une application du modèle d'évolution SAEV et le rôle des propriétés sémantiques dans la gestion de l'évolution.

5.2 L'ADL COSA (Component-Object based Software Architecture)

COSA est un ADL hybride, proposé par notre équipe [76, 42]. Il se base sur la modélisation par objets et la modélisation par composants, pour la description des systèmes logiciels. COSA propose une description descendante de type top-Down. La contribution principale de cet ADL est, d'une part, d'emprunter le formalisme des ADL et de les étendre grâce aux concepts et mécanismes objets, et d'autre part, d'expliciter les connecteurs en tant qu'entités de première classe pour traiter les dépendances complexes entre les composants.

Dans COSA, les composants, les connecteurs et les configurations sont des classes qui peuvent être instanciées pour établir différentes architectures. Comme COSA sépare la description d'architecture de son déploiement, il est possible de déployer un composant architectural donné de plusieurs manières, sans réécrire le programme de configuration/déploiement. Les éléments de base de COSA sont : les composants, les connecteurs, les interfaces, les configurations, les contraintes et les propriétés non fonctionnelles.

5.2.1 Pourquoi l'ADL COSA ?

Notre choix pour l'ADL COSA se justifie notamment par les raisons suivantes :

- COSA réifie la majorité des concepts utilisés dans une modélisation structurelle d'architectures logicielles et représente explicitement leurs caractéristiques et les relations qui existent entre eux
  - Ces concepts ainsi réifiés peuvent être manipulés et par conséquent nous pouvons les faire évoluer. Ainsi, l'utilisation de COSA nous permettra d'illustrer des exemples d'évolution de l'ensemble de ces concepts architecturaux.
  - Cette réification des concepts nous permet aussi de décrire tous les éléments architecturaux, avec les mêmes niveaux d'abstraction Méta, Architectural, Application. Ainsi nous pouvons illustrer l'utilisation du modèle SAEV au travers de ces différents niveaux d'abstraction.

- COSA confère aux connecteurs une importance et un niveau de granularité et de réutilisation semblable à celui des composants. Un connecteur COSA est défini explicitement et réifié comme une entité de première classe. Nous pouvons ainsi définir une extension du concept connecteur de COSA par les propriétés sémantiques proposées

- Des stratégies sont proposées dans [76, 80] pour permettre le passage d'une spécification en COSA vers d'autres ADL tels UML2.0 [35]. UML2.0 constitue un des ADL les plus acceptés dans la communauté industrielle. Ainsi une spécification en COSA peut toujours être transformée en une spécification en UML2.0.

5.2.2 Méta modèle de COSA

L'ADL COSA décrit les systèmes en termes de classes et d'instances. Les éléments architecturaux sont des classes qui peuvent être instanciées pour construire plusieurs architectures. Les concepts de base de l'architecture COSA sont : le composant, le connecteur, la configuration, l'interface, les contraintes et les propriétés (fonctionnelles et non-fonctionnelles). Ces concepts sont illustrés par le méta modèle de la figure 5.1.

Le méta-modèle COSA est décrit en utilisant le diagramme de classes de la notation UML [35]. Ainsi, les concepts de COSA sont représentés sous formes de classes, leurs caractéristiques sous formes d'attributs et les différents liens entre les concepts sont représentés par des associations (simple association, composition, héritage). Par ailleurs, des classes abstraites qui n'ont pas de correspondances conceptuelles sont rajoutées uniquement comme support de factorisation, telles que les classes *Elément-Architectural* et *Interface*.

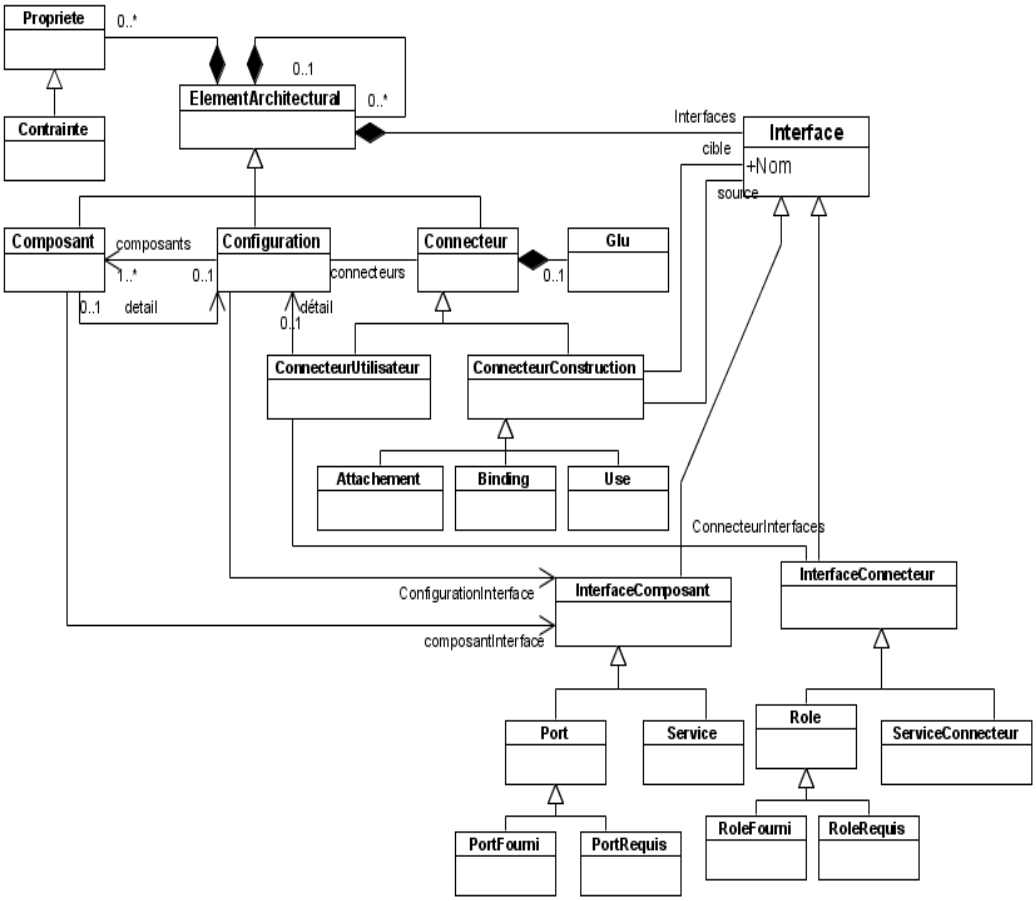

Figure 5.1 – Méta modèle de COSA

La figure décrit le méta-modèle COSA où certains détails ne sont pas représentés par souci de clarté. Cette figure montre entre autres, 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 (appelée *port*) et l'interface d'un connecteur (appelée *rôle*) et que les interfaces fournissent des points de connexion entre les composants et les connecteurs. Nous détaillons les principaux concepts de COSA dans les paragraphes suivants.

5.2.2.1 La configuration dans COSA

Dans COSA, la configuration est une classe qui peut être instanciée plusieurs fois et qui donnera plusieurs architectures d'un système. 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 seront 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 5.2 définit les principales parties d'une configuration COSA.

Une configuration COSA est ainsi composée d'interfaces et de connecteurs ainsi que d'au moins un

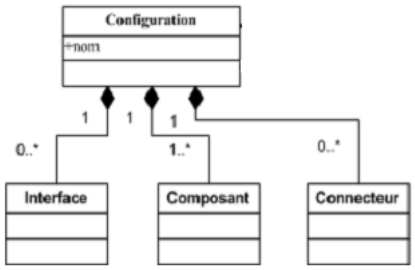

Figure 5.2 – Structure d'une configuration de COSA

composant.

5.2.2.2 Le composant dans COSA

Un composant représente un élément de calcul et de stockage de données d'un système logiciel. Chaque composant possède une ou plusieurs interfaces ayant plusieurs ports. Un composant peut avoir plusieurs implémentations. La figure 5.12 décrit un composant d'une architecture COSA.

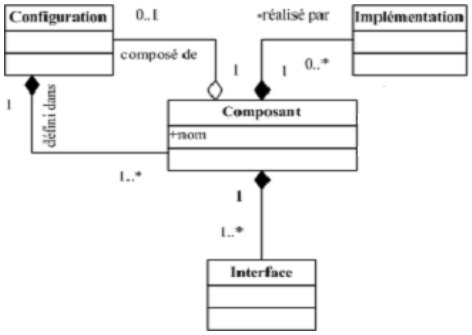

Figure 5.3 – Structure d'une composant de COSA

Un composant peut être composite, dans ce cas il sera décrit par une configuration interne.

5.2.2.3 L'interface dans COSA

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 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.

- Le *point de connexion* : appelé également *port* pour le point de connexion de composants et de configuration et *rôle* pour le point de connexion de connecteurs, peut être soit un port fourni/rôle fourni (souvent appelé port service/rôle service) soit un port requis/rôle requis (souvent appelé port besoin/rôle besoin).
- Le *service* : décrit d'une part, le comportement fonctionnel de l'élément architectural en expliquant ce qu'il fait, on parlera alors de *services fournis*, et d'autre part, les fonctionnalités dont il a besoin

pour fonctionner, il s'agit alors de *services requis*. Un service peut utiliser un ou plusieurs points de connexion pour exécuter sa tâche. La figure 5.4 décrit l'interface de COSA.

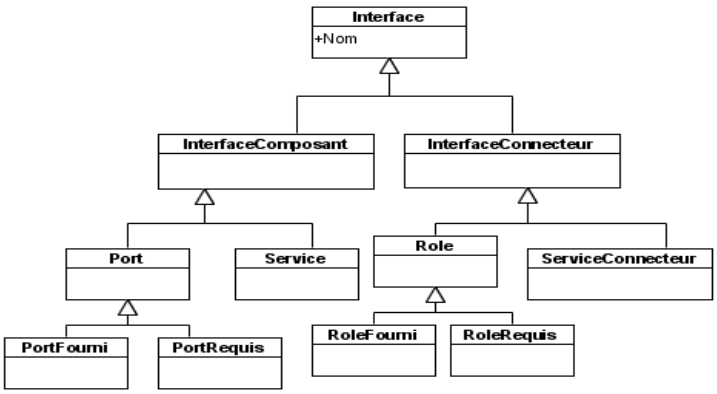

Figure 5.4 – Structure d'une interface de COSA

5.2.2.4 Le connecteur dans COSA

Dans COSA, les connecteurs sont définis comme des entités de première classe et utilisés pour connecter des composants, des configurations ou des interface [78], [63]. COSA distingue deux catégories de connecteurs : les *Connecteurs d'utilisateur* et les *Connecteur de construction* illustrés par la figure suivante :

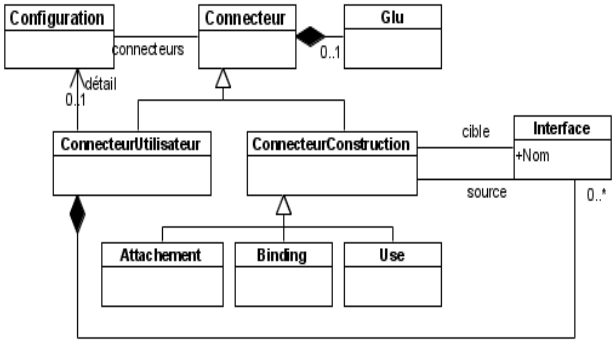

Figure 5.5 – Structure d'un connecteur de COSA

a. Connecteurs d'utilisateur

Un connecteur d'utilisateur est principalement défini par une *interface* et une *glu*. 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 la conversion de format de données, le transfert ou l'adaptation. Les connecteurs peuvent avoir leur propre architecture interne qui contient des calculs

et du stockage de données. Dans le cas d'un connecteur composite, les sous-connecteurs et les sous-composants (qui sont définis à l'aide d'une configuration) de ce connecteur représentent la glu de ce connecteur.

b. Connecteurs de construction : Attachements, Bindings et Use

COSA distingue trois types de connecteurs pour relier les interfaces des différents éléments architecturaux : Attachments, Bindings et Use.

b.1. Attachement (Liaison) : une configuration d'un système en COSA est définie en énumérant un ensemble d'attachements qui lient les ports d'un composant ou d'une configuration aux rôles d'un connecteur. Un port requis d'un composant (ou d'une configuration) peut être relié à un rôle fourni d'un connecteur et un port fourni d'un composant (ou d'une configuration) peut être relié à un rôle requis d'un connecteur.

b.2. Binding (Correspondance) : quand un composant ou un connecteur et a fortiori une configuration disposent d'une description interne, une correspondance entre leurs descriptions externes et internes doit être mise en place. Le Binding définit cette correspondance. Un Binding fournit donc une association entre les ports (respectivement rôles) internes et les ports (respectivement rôles) externes des composants et des configurations (respectivement connecteurs).

b.3. Use (Utilise) : le lien Use relie des services aux ports des composants ou aux rôles des connecteurs. Par exemple, un port fourni d'un composant est associé à un service fourni de ce composant et un port requis est associé à un service requis.

Nous parlerons alors dans COSA de quatre types de connecteurs :

- Connecteur Utilisateur : qui relie une ou plusieurs interfaces composant fournies et une ou plusieurs interfaces requises d'un autre composant ;
- Attachement : qui relie une interface requise (respectivement interface fournie) d'un composant à une interface fournie (respectivement requise) d'un connecteur ;
- Binding : qui relie une interface fournie (respectivement requise) d'une configuration , d'un composant ou d'un connecteur et une interface fournie (respectivement requise) de leurs éléments constitutifs.
- Use : qui relie des services aux ports (ou aux rôles pour les connecteurs).

Après avoir présenté les principaux concepts de COSA ainsi que les différents liens qui existent entre eux, nous présentons dans la section suivante l'exemple sur lequel nous nous basons pour illustrer nos propositions.

5.3 Architecture Client/Serveur : exemple de l'illustration

Nous nous basons sur l'exemple didactique du Client/Serveur tiré de [32] pour illustrer une utilisation des propriétés sémantiques et une application du modèle d'évolution SAEV en tenant compte de ces propriétés sémantiques proposées. Cet exemple est très répandu dans la communauté des architectures logicielles et est traité explicitement par la plupart des ADL.

L'architecture Client/Serveur que nous présentons est décrite au niveau *Architectural*. Elle est composée de deux composants **Client** et **Server** qui communiquent par l'intermédiaire d'un connecteur **RPC**. La spécification graphique de cette architecture en utilisant COSABuilder [51] est donnée par la figure suivante :

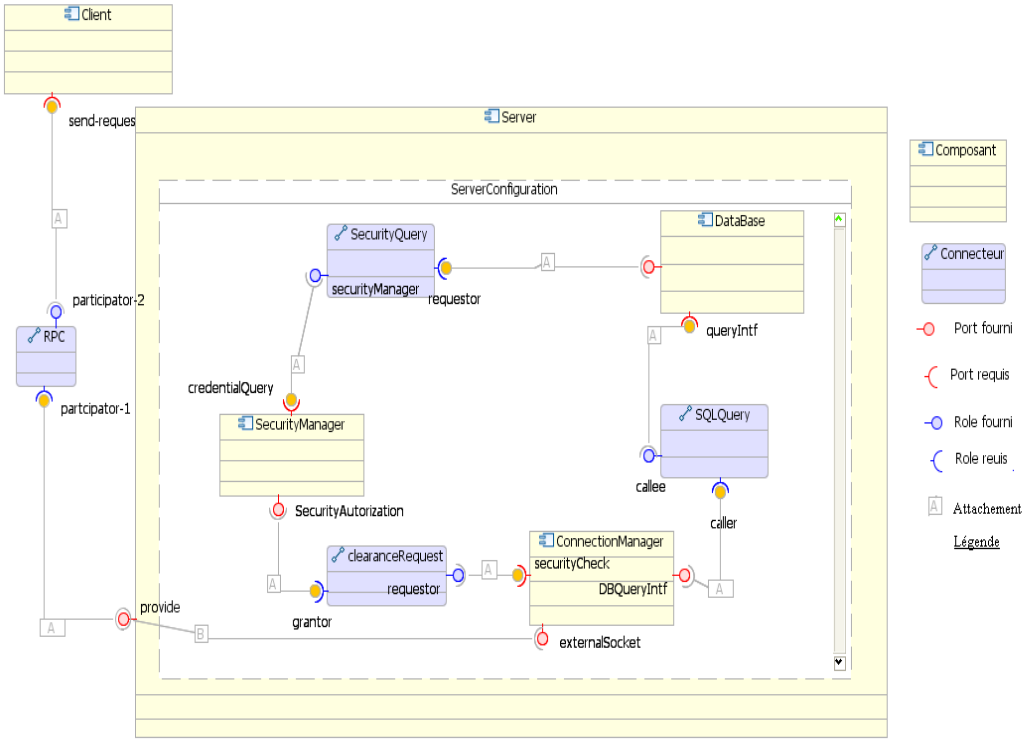

Figure 5.6 – L'architecture Client/Serveur en COSA

- Le composant **Client** possède un port requis *send-request* et un service requis *demand-data*.
- Le composant **Server** est un composite. Il est décrit par un port de type fourni *provide* et un service de type fourni *send-data*. Ce composant **Server** est composé d'une configuration de :
  - Trois composants : ConnectionManager, SecurityManager, Database :

    1. Le composant **ConnectionManager** possède deux ports fournis *externelSocket* et *BDQueryIntf* et un port requis *securityCheck*.
    2. Le composant **SecurityManager** posséde un port fourni *securityAuthorization* et un port requis *credentialQuery*.
    3. Le composant **Database** posséde un port requis *queryIntf* et un port fourni *securityManagement*.

  - Trois connecteurs : SQLQuery, ClearanceRequest et SecurityQuery :

1. Le connecteur **SQLQuery** possède un rôle fourni *callee* et un rôle requis *caller*.
2. Le connecteur **ClearanceRequest** possède un rôle requis *grantor* et un rôle fourni *requestor*.
3. Le connecteur **SecurityQuery** possède un rôle fourni *securityManager* et un rôle requis *requestor*.

- Le connecteur **RPC** agit comme un médiateur entre les deux composants, il possède un rôle requis *participator-1* et un rôle fourni *participator-2* et son type de service est Conversion.

Après avoir illustrer les principaux concepts de l'ADL COSA, nous avons présenté un exemple d'une architecture Client/Serveur que nous avons décrit en COSA. Nous nous servirons de cet exemple tout au long de ce chapitre pour illustrer les propriétés sémantiques ainsi que le modèle d'évolution SAEV.

5.4 Illustration des propriétés sémantiques au travers de l'ADL COSA

Nous rappelons que nous avons proposé les propriétés sémantiques d'*Exclusivité*/*partage*, de *Dépendance*/ *indépendance*, de *Prédominance* /*Non prédominance* de *Cardinalité* et *Cardinalité inverse* pour enrichir la sémantique des connecteurs par plus d'informations sur le degré de corrélation entre les composants d'une architecture. Ces propriétés sémantiques seront ainsi exploitées par le modèle SAEV pour déterminer et propager automatiquement autant que possible les impacts d'une évolution au sein d'une architecture. Nous avons montré dans le chapitre 4 que les propriétés sémantiques peuvent être spécifiées au niveau *Architectural* (cf. section 4.3), comme elles peuvent être spécifiées au niveau *Méta* (cf. section 4.4). Nous illustrons ces deux cas dans le cadre de l'ADL COSA.

5.4.1 Propriétés sémantiques au niveau Architectural

Lors de la description des propriétés sémantiques dans le chapitre précédent, nous n'avons considéré aucune structure particulière de connecteur, faisant ainsi abstraction des détails de la structure du concept connecteur. Il s'agit dans cette section de montrer comment ces propriétés seront prise en compte dans l'ADL COSA.

Les propriétés sémantique proposées sont introduites dans le concept *Connecteur* de COSA. Nous reflétons cette extension dans le méta modèle de COSA (figure 5.1) en modifiant la méta classe **Connecteur** par l'ajout des propriétés sémantiques comme attributs descriptifs du concept Connecteur.

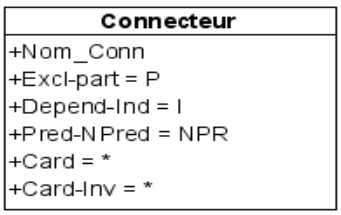

Figure 5.7 – Valeurs par défaut des connecteurs COSA

A chaque propriété d'un connecteur et donc à chaque attribut représentant une propriété est associée une valeur par défaut. Nous considérons que par défaut un connecteur COSA ne véhicule aucune contraintes entre les élément qu'il relie. Ainsi, un connecteur COSA est par défaut :

- Partagé (p) ;
- Indépendant (I) ;
- Non prédominant (NPR) ;
- A une cardinalité et cardinalité inverse variable [*].

Etant donné que le concept de connecteur de COSA est spécialisé en deux sous classes, *ConnecteurUtilisateur* et *ConnecteurConstruction*, ces dernières héritent de la spécification des propriétés sémantiques. Cependant ces propriétés ont des spécificités liées à chacun de ces types de connecteur.

5.4.1.1 Propriétés sémantiques des connecteurs Utilisateur

Un Connecteur utilisateur définit les mêmes valeurs par défaut que celle définies au niveau du concept *Connecteur*. A la spécification d'un connecteur utilisateur au niveau Architectural les propriétés sémantiques peuvent être redéfinies par le concepteur. A titre d'exemple le connecteur *RPC* entre le composant *Client* et le composant *Serveur* peut définir la propriété de Partagé source-Exclusif cible **(Ps-Ec)** à la place de la propriété de *partage* par défaut. Ceci implique qu'à un *Serveur* peuvent être attachés plusieurs *Clients*, mais un *Client* ne peut être relié qu'à un seul *Serveur*.

5.4.1.2 Propriétés sémantiques des connecteurs de Construction

a. Propriétés sémantiques au niveau des Attachements

Nous jugeons qu'il est nécessaire de préciser les propriétés sémantiques au niveau des attachements pour les raisons suivantes :

- La structure des connecteurs en COSA est telle qu'un connecteur peut relier à la fois plusieurs ports de composants à plusieurs autres ports d'autres composants (figure 5.8) ;
- Le concepteur peut vouloir exprimer des corrélations différentes entre les ports reliés deux à deux par le même connecteur. Par exemple, dans la figure 5.8 nous avons besoins d'exprimer des corrélations entre les ports (*Pf1* et *Pr1*) qui sont différentes de celles entre les ports (*Pf2* et *Pr1*).

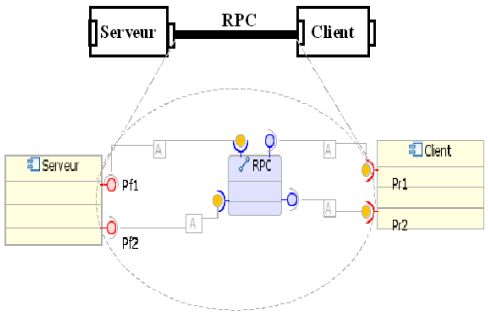

Figure 5.8 – Spécificités des connecteurs COSA

Plusieurs cas peuvent être distingués :

1. Un connecteur relie un *port* composant *fourni* à un seul *port* composant *requis*. Dans ce cas les propriétés associées au connecteur expriment bien la corrélation qui existe entre ces deux ports composant. Ces mêmes propriétés sont alors spécifiées aussi au niveau des *attachements* qui relient ces ports composant au connecteur concerné.

2. Un connecteur relie plusieurs *ports* composant fournis à plusieurs autres *ports* requis, deux cas peuvent alors se présenter :

   - Si les propriétés à exprimer entre tous les ports composants fournis et tous les ports composant requis sont les mêmes, ces propriétés sont spécifiées au niveau du connecteur et au niveau des attachements.
   - Dans le cas où nous avons besoin d'exprimer plusieurs propriétés différentes entre les ports composants reliés par un même connecteur, les propriétés sémantiques seront exprimées uniquement au niveau des attachements, auxquels ces ports composants sont reliées. Le connecteur ne pourra alors définir que les valeurs des propriétés par défaut.

Ainsi, les restrictions suivantes doivent être ajoutées aux contraintes liées à la définition des propriétés sémantiques d'un connecteur au niveau architectural :

- Si un connecteur définit un ensemble de propriétés (différentes des propriétés par défauts), ces propriétés doivent être reportées sur l'ensemble des attachements reliés à ce connecteur.

- Si des attachements reliés à un connecteur définissent des propriétés sémantiques différentes, alors le connecteur auxquels sont reliés ces attachements ne peut définir des propriétés sémantiques en dehors de ses propriétés par défauts.

En tenant compte de ces particularités liées aux connecteurs de COSA, une propriété sémantique d'un attachement peut être définie comme suit :

$P_{\textbf{Attachement}}$: $\textbf{E}_{\textbf{source}}$, $\textbf{E}_{\textbf{cible}}$ tel que $\textbf{E}_{\textbf{source}}$: représente un *port fourni* d'une configuration ou d'un composant ; $\textbf{E}_{\textbf{cible}}$: représente un *port requis* d'une configuration ou d'un composant.

Le lien entre les deux ports composants concernés par les propriétés associées à un attachement donné est identifiable au niveau du connecteur auquel est associé cet attachement.

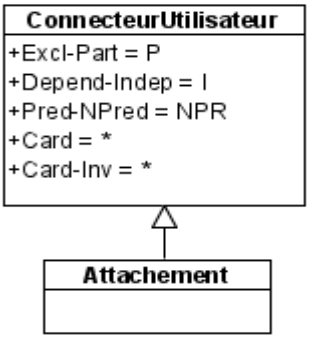

Figure 5.9 – Valeurs par défaut des attachements COSA

Les valeurs par défaut d'un attachement sont les mêmes valeurs par défaut des connecteurs. Ainsi un attachement est par défaut :

- Partagé (p) ;
- Indépendant (I) ;
- Non prédominant (NPR) ;
- A une cardinalité et cardinalité inverse variable [*].

b. Propriétés sémantiques au niveau des Bindings

Les propriétés associées à un binding expriment le degré de corrélation entre un port (respectivement un rôle) d'un composant composite ou d'une configuration (respectivement d'un connecteur composite) et un port (respectivement rôle) de l'un de ses éléments internes.

Pour exprimer, par exemple, que la suppression d'un port requis d'une configuration implique toujours la suppression du port requis du composant interne auquel il est attaché une propriété de *dépendance forte* peut être alors associée à ce binding.

Nous définissons les propriétés associées à un binding comme suit :

$P_{Binding} : E_{source}, E_{cible}$ tel que **E_{source}** : représente un port requis d'une configuration ou un port fourni d'un composant **E_{cible}** : représente respectivement un port requis d'un composant, un port fourni d'une configuration.

En tenant compte de la définition des bindings en COSA, les valeurs des propriétés sémantiques sont redéfinies au niveau des bindings comme suit :

- Partagé (P) ;
- Indépendant (Df) ;
- Non prédominant (PRf) ;
- A une cardinalité et cardinalité inverse variable [*].

Ainsi, par exemple un *port requis* d'une configuration peut être relié à plusieurs *ports requis* de ses composants internes (via la propriété Excl-part à **P** et card à *****). Un *port requis* d'un composant interne peut être relié à plusieurs ports requis de la configuration auquel il appartient (Excl-part à **P** Card-inv à *****).

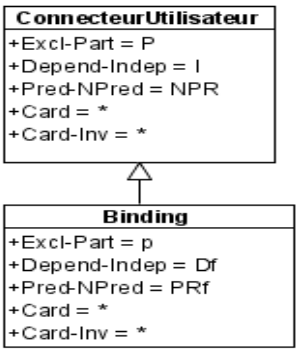

Figure 5.10 – Valeurs par défaut des bindings COSA

La suppression d'un port requis d'une configuration implique la suppression du port de son composant interne auquel il est attaché si ce dernier n'est pas relié à d'autres ports (**Df**). La suppression d'un *port requis* d'un composant interne implique la suppression du *port requis* de la configuration à laquelle il appartient si ce dernier n'est pas relié à d'autres ports (**PRf**).

c. Propriétés sémantiques au niveau des Uses (liens d'utilisation)

Les liens d'utilisation permettent d'associer un *service fourni* ou *requis* d'un composant respectivement d'un connecteur aux *ports* (respectivement *rôles*) *fournis* ou *requis* qui permettent sa réalisation. Un service peut utiliser plusieurs ports (respectivement rôles). Les propriétés sémantiques au niveau d'un lien *Use* permettront alors d'exprimer le degré de corrélation entre les deux extrémités (*port*, *role* et *service*) de ce lien. A titre d'exemple, pour exprimer qu'un *port* est indispensable pour la réalisation d'un *service*, une propriété de *dépendance forte* doit être associée au lien d'utilisation entre ce *service* et ce *port*.

Ainsi, une propriété associée à un Use peut être définie par :

P_{Use} : E_{source}, E_{cible} tel que E_{source} : représente un service fourni (respectivement requis) d'une configuration ou d'un composant E_{cible} : représente un port fourni (respectivement requis) d'une configuration ou d'un composant ou un rôle fourni (respectivement requis) d'un connecteur.

Les valeurs par défaut des propriétés sémantiques sont redéfinies au niveau des connecteurs Use comme suit :

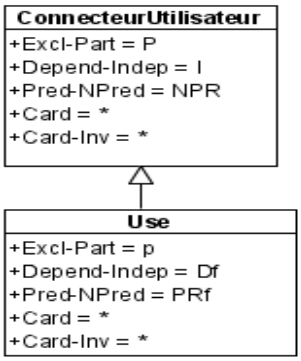

Figure 5.11 – Valeurs par défaut des connecteurs Use COSA

- Partagé (p) ;
- Indépendant (Df) ;
- Non prédominant (PRf) ;
- A une cardinalité et cardinalité inverse variable [*].

Ainsi, un *service* peut utiliser plusieurs *ports* ou *rôles* (Excl-Part à P et cardinalité à *). Un *port* ou un *rôle* peut être utilisé par plusieurs *services* (Excl-Part à P et la cardinalité inverse à *). La suppression de tous les services qui utilisent un port implique la suppression de ce port (Depend-Indep à Df). La suppression de tous les ports qu'un service utilise implique la suppression de ce service (Pred-NPred à PRf).

Nous avons montré dans cette section la prise en compte des propriétés sémantiques proposées au niveau de du concept connecteur de COSA. Nous avons montré que ces propriétés peuvent être associées aux connecteurs *utilisateurs* comme elles peuvent être associées aux connecteurs de *construction* (Attachements, aux Bindings et aux liens Use). Dans la section suivante, nous illustrons la prise en compte de ces propriétés sémantiques sur l'exemple du Client/Serveur présenté dans la section 5.3.

5.4.1.3 Application des propriétés sémantiques à l'exemple d'illustration

Nous présentons dans cette section la spécification des propriétés sémantiques dans l'architecture Client/Serveur présenté dans la section 5.3. Nous illustrons uniquement les propriétés sémantiques des connecteurs utilisateurs, nous ne illustrons pas les propriétés sémantiques des connecteurs de construction.

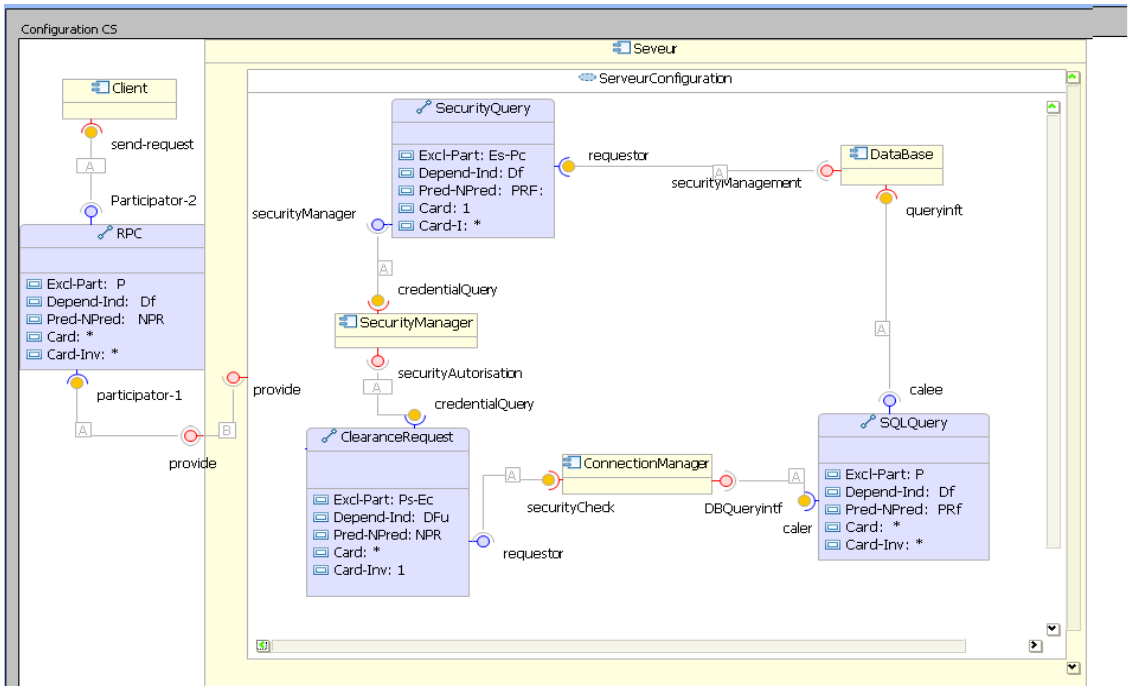

Figure 5.12 – L'architecture Client/Serveur en COSA

Nous avons illustré dans l'exemple les propriétés sémantiques associées aux différents connecteurs de l'architecture logicielle Client/Serveur. Nous illustrons uniquement les propriétés associées aux connecteurs utilisateurs. A titre d'exemple, le connecteur *RPC* entre le composant *Client* et le composant *Serveur* :

- est Partagé source-Exclusif cible (**Ps-Ec**) : ceci implique qu'à un *serveur* est attachés plusieurs *clients*, mais un *Client* ne peut être relié qu'à un seul *serveur*.
- définit la propriété de dépendance forte unique (**Dfu**) : ceci indique que le port requis **send-request** du composant *Client* est dépendant de port fourni *provide* du composant *Serveur* auquel il est attaché via le connecteur *RPC*. La suppression de ce port du composant *Serveur* impliquera la suppression du port du composant *Client*.
- définit la propriété de non prédominance (**NPR**), donc la suppression du port requis du composant *Client* n'implique pas la suppression du port fourni du composant serveur.
- définit la cardinalité de [**1..10**], ceci implique que 10 *Clients* au maximum peuvent être attachés à un composant *Serveur*.
- définit la cardinalité inverse de *1*, ceci implique qu'un *Client* ne peut être attaché qu'à un seul Composant *Serveur*.

Nous montrons dans la section 5.5.2, comment ces propriétés sémantiques spécifiées au niveau de cette architecture Client/serveur seront considérées par SAEV pour la détermination et la propagation des impacts à l'évolution d'une application Client/Serveur.

Dans la section suivante nous illustrons la prise en compte des propriétés sémantiques au niveau méta

de l'ADL COSA.

5.4.2 Les propriétés sémantiques au niveau Méta

Les propriétés sémantiques telles qu'elles ont été présentées jusqu'ici sont définies à la spécification d'une architecture au niveau architectural et interviendront à l'évolution de toute application construite à partir de cette architecture.

Nous avons montré dans la section 4.4 du chapitre 4, que les propriétés sémantiques peuvent également être spécifiées au niveau méta et interviendront à l'évolution d'une architecture au niveau architectural et/ou au niveau application. Cela permet une uniformité dans la spécification des propriétés sémantiques à tous les niveaux d'abstraction. Pour pouvoir spécifier ces propriétés au niveau méta nous avons proposé de réifier les liens structuraux entre les concepts comme étant des *connecteurs Niveau-Méta* (nous rappelons qu'un connecteur Niveau-Méta est un connecteur utilisé pour relier deux concepts d'un ADL au niveau Méta).

Nous appliquons cette réification à l'ADL COSA. Ainsi, chaque lien défini au niveau du méta modèle COSA (lien de composition et d'association) sera réifié comme étant un connecteur Niveau-Méta. Nous obtenons ainsi deux types de connecteurs de Niveau-Méta en COSA :

- Connecteur de composition : utiliser par exemple pour relier les concepts composant et port, connecteur et rôle, configuration et composant etc.
- Connecteur d'association : utiliser pour relier par exemple un port à un attachement, un port à un binding, etc.

A chacun de ces types de connecteur Niveau-Méta sont associés des propriétés sémantiques par défaut. Nous les présentons dans le tableau suivant.

Connecteur Niveau-Méta	**Propriétés**	**Connecteur COSA du niveau méta**
Connecteur de composition	Exclusivité : **Pc-Es**	**Composition :Configuration-Port configuration** **Source** : Configuration **Cible** : Port-configuration
	Dépendance : **DFu**	**Composition :Configuration-Service configuration** **Source** : Configuration **Cible** : Service configuration
	Prédominance : **PRf**	**Composition : Composant-Port composant** **Source** : Composant **Cible** : Port composant
	Card : ***** ;	**Composition : Composant-Service composant** **Source** : Composant **Cible** : Service composant
	Card-inv : **1**	Composition : **Connecteur-Rôle connecteur** **Source** : Connecteur **Cible** : Role connecteur
		Composition Connecteur-Service connecteur **Source** : Connecteur **Cible** : service connecteur

... suite page suivante...

Connecteur Niveau-Méta	Propriétés	Connecteur COSA du niveau méta
		Composition : Configuration-Composant **Source** : Configuration **Cible** : Composant
		Composition : Configuration-Connecteur **Source** : Configuration **Cible** : Connecteur
Connecteur d'association	Exclusivité :**Es-Pc**	**Association : Port - attachement** **Source** : Port **Cible** : Attachement
	Dépendance : **D**Fu Prédominance :**PRf**	**Association : Rôle-Attachement** **Source** : Rôle **cible** : Attachement
	Card : ***** Card-Inv : *****	**Association : Port-Binding** **Source** : Port **cible** : Binding

Table 5.1: Connecteurs Niveau-Méta de COSA

Nous avons défini dans le tableau précédent les connecteurs Niveau-Méta que nous définissons pour COSA, ainsi que les propriétés que nous leurs associons par défaut. Chaque connecteur est désigné par un nom qui représente une concaténation du type du connecteur et des noms des concepts que le connecteur relie. Par exemple le connecteur de composition entre le concept *Composant* et le concept *Port* est noté : **Composition : Composant-Port**.

Pour chaque connecteur nous avons défini ses valeurs par défaut. A titre d'exemple, nous avons défini le connecteur *Composition : Composant-port-composant* comme étant *Partagé source- Exclusif cible* (**PS-Ec**), de dépendance forte unique (**Dfu**), de prédominance faible (**Pf**), de cardinalité variable et de cardinalité inverse égale à **1**. Ces valeurs sont définies au niveau méta ainsi, elles s'interprètent au niveau architectural comme suit :

- à un *composant* correspond un ou plusieurs *ports* (via la propriété Partagé source et cardinalité variable *) ;
- un *port* appartient à un et un seul *composant* (via la propriété d'Exclusivité cible et cardinalité inverse à 1) ;
- la suppression du *composant* implique automatiquement la suppression de ses *ports* (via la propriété de Dfu) ;
- la suppression d'un *port* n'implique pas automatiquement la suppression d'un *composant*, mais la suppression de tous les ports du composant implique la suppression de ce *composant* (via la propriété PRf) ;

Les valeurs définies dans le tableau précédent sont des valeurs par défaut. Un concepteur peut redéfinir ces valeurs à la spécification d'une architecture au niveau architectural. Par exemple, la propriété de prédominance entre un *composant* et un de ses *ports* peut être modifiée de la *prédominance faible* à la *prédominance forte*. Ainsi, ce port devient indispensable à l'existence de ce composant et sa suppression impliquerait directement la suppression du composant auquel il appartient.

Avec la même analogie qu'avec le niveau Architectural et le niveau Application. Les connecteurs Niveau-Méta et leurs propriétés sémantiques par défaut doivent être introduits à un niveau au **dessus du niveau méta**. Ainsi, les valeurs des propriétés sémantiques seront spécifiées au niveau méta pour s'appliquer aux niveaux inférieurs (Architectural et Application). Nous nous sommes limités au cours de ce travail à trois niveaux d'abstraction de description architecturale, par conséquent, nous ne montrons pas comment ces propriétés sont introduites dans le niveau au dessus du niveau méta.

5.4.3 Bilan

Nous avons illustré dans cette section l'extension du concept connecteur COSA par le propriétés sémantiques proposées. Nous avons montré en section 5.4.1 que les propriétés sémantiques sont intégrées au niveau Méta de COSA dans le concept Connecteur, leurs valeurs sont spécifiées au niveau Architectural à la spécification d'une architecture et elles interviendront à l'évolution de toute application construite à partir de cette architecture. Nous avons montré en section 5.4.2 aussi que ces propriétés peuvent être spécifiées au niveau Méta. Nous avons proposé pour cela des connecteurs Niveau-Méta auxquels nous avons associé ces propriétés sémantiques.

Dans les sections suivantes, nous illustrons l'utilisation du modèle SAEV sur des architectures décrites en COSA enrichi par les propriétés sémantiques définies.

5.5 Illustration de SAEV au travers de l'ADL COSA

Nous illustrons dans cette section l'application du modèle d'évolution SAEV sur des exemples d'architectures décrites via l'ADL COSA, en tenant compte des propriétés sémantiques définies précédemment. Nous montrons, une évolution au niveau *Architectural* et une évolution au niveau *Application*. Nous nous appuyons sur l'exemple didactique du Client /Serveur défini dans la section 5.3 et enrichi par les propriétés sémantiques dans la section 5.4.1.3. Nous considérons dans un premier temps uniquement le cas ou les *propriétés sémantiques définies au niveau Architectural et qui s'appliquent à l'évolution d'une Application*. Nous ne considérons pas les connecteurs Niveau-Méta de COSA ainsi que leurs propriétés sémantiques.

Nous rappelons que dans le modèle SAEV, l'évolution doit être d'abord spécifiée en utilisant les concepts proposés par le modèle avant de procéder à l'exécution de cette évolution. Spécifier une évolution dans SAEV au niveau architectural ou au niveau application revient à préciser :

- L'ensemble des éléments architecturaux pouvant évoluer $\longrightarrow$ spécifier le concept **ElémentArchitectural** de SAEV.
- L'ensemble des stratégies d'évolutions associées à ces éléments architecturaux $\longrightarrow$ spécifier les concepts **StrategieEvolution**, **Invariant** et **RegleEvolution**.

5.5.1 Exemple d'une évolution au niveau Architectural

Considérant l'architecture Client/Serveur de la section 5.4 :

L'évolution à répercuter sur cette architecture Client/Serveur consiste à rendre l'accès **libre** à la base de données du composant **Server**. L'authentification n'est plus nécessaire pour accéder à la base de données. Cette évolution se reflète concrètement sur l'architecture par la suppression du composant

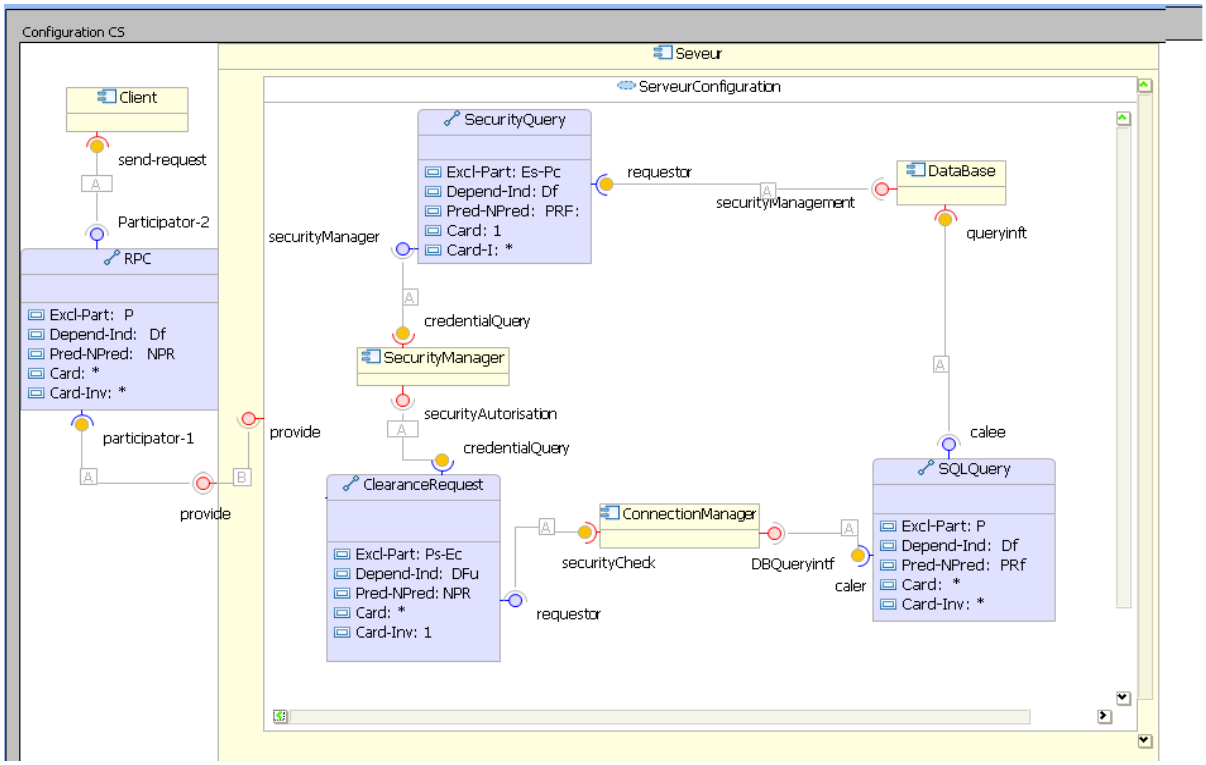

Figure 5.13 – Architecture Client/Serveur

securityManager et par les propagations que cette suppression engendre. Nous détaillons dans ce qui suit comment cette évolution sera spécifiée et exécutée via le modèle SAEV.

5.5.1.1 Spécification de l' évolution de l'architecture Client/Serveur

a. Les éléments architecturaux : les éléments de l'architecture à faire évoluer sont décrits dans les tableaux suivants. La spécification des connecteurs entre autres des indique leurs propriétés sémantiques.

ConnectionManager : **Composant**	Database : **Composant**	securityManager :**Composant**
Port-requis : securityCheck	***Port-requis*** : securityManagement	***Port-requis*** : credentialQuery securityAuthorization
Port-fourni : externalSoket BDQueryIntf	***Service-requis*** : queryIntf	***Port-fourni*** : ...
Service-requis : ...	***Service-requis*** : ...	***Service-requis*** : ...
Service-fourni : ...	***Service-fourni*** : ...	***Service-fourni*** : ...

Serveur : **Composant**	ClearanceRequest : **Connecteur**	SecurityQuery :**Connecteur**
Port-requis : securityCheck	***Role-requis*** : Grantor	***Role-requis*** : requestor
port-fourni : externalSoket	***Role-fourni*** : request	***Role-fourni*** : securityManager
Service-requis : ...	***Service-requis*** : ...	***Service-requis*** : ...
Service-fourni : ...	***Service-fourni*** : ...	***Service-fourni*** : ...
	Source : securityAuthorization	***Source*** : securityManagement
	Cible : securityCheck	***Cible*** : credentialQuery
	Exclusivite : Ps-Ec	***Exclusivité*** : Es-Pc
	Dependance : DFu	***Dependance*** : DFu
	Predominance : NPR	***Predominance*** : PRf
	Card : *	***Card : 1***
	Card : 1	***Card :*** *

b. Stratégies d'évolution : les stratégies d'évolution associées aux éléments architecturaux et permettant de faire évoluer l'architecture sont données par le tableau suivant. Pour chaque stratégie nous avons apporté des exemples de règles d'évolution qui nous servirons dans l'exemple d'illustration :

Stratégies d'évolution	**Configuration**	**Composant**	**Connecteur**	**Interface**
Stratégies d'ajout : Règles				
Stratégie de suppression : Règles		**SSComp** : R1,	**SSCon** : R4,R3,	**SSInt** : R2,
Stratégie de modification : Règles	**SMConf** : R6, R7	**SMComp** : R5,		
Stratégie de substitution : Règles				

c. Les règles d'évolution : nous présentons uniquement la liste des règles que nous jugeons nécessaires pour illustrer notre exemple. A titre d'exemple nous présentons la règle *R1* qui décrit la suppression d'un composant d'une configuration. La règle *R1* indique qu'avant de supprimer un composant, il faut modifier la configuration à laquelle il appartient, en le supprimant de la liste des composants de cette configuration, supprimer l'interface du composant à supprimer puis déclencher la suppression proprement dit de ce composant.

R1 : Règle de Suppression Composant	**R2 : Règle de Suppression interface-Composant**
Evenement : Supprimer-composant(C :Composant, Cf : :Configuration	**Evenement** : Supprimer-interface-composant(I : Interface-composant, C : Composant, Cf :Configuration)
Condition : C ∈ Composants(Cf)	**Condition** :I ∈ Ports-composant(C) ou I ∈ Services-composant(C)
Action : Modifier-configuration(Cf, C) **Pour** tout I ∈ Interface-composant(C) Supprimer-interface-composant (I, C,Cf) **Finpour** *C.Execute-supprimer-composant()*	**Action :** Modifier-composant(C,I) **Si** type (I)= Port **alors** **Pour** tout t ∈ Attachements(Cf) tel que I = source(t) ou I = cible (t) alors Supprimer-attachement(t,I,C) **Finpour** *I.Execute-supprimer-port-composant(C)* Si type(I)= Service alors *I.Execute-supprimer-service-composant(C)*

R3 : Règle de suppression Attachement	**R4 : Règle de suppression connecteur**
Evenement : Supprimer-attachement(t :Attachement, I :Interface, C :composant)	**Evenement** : Supprimer-connecteur(N :Connecteur, Cf :Configuration)
Condition :I = source(t) ou I = cible (t)	**Condition** : N ∈ connecteurs(Cf)
Action : Modifier-configuration(configuration(C), t) *t.Execute-supprimer-attachement(configuration(C))*	**Actions** : Modifier-configuration(N,Cf) **Pour** tout I interface(N) Supprimer-interface-connecteur(I,N) **FinPour** *N.Executer-supprimer-connecteur()*

R5 : Règle modification Composant	**R6 : Règle de Modification Configuration**
Evenement : Modifier-composant(C :Composant, I :Port)	**Evenement** : Modifier-configuration (Cf :Configuration, C :Composant)
Condition : I ∈ Ports(C)	**Condition** : C ∈ Composants(Cf)
Action : Ports-composant(C) := Ports-composant (C)- I ;	**Action** : Composants(Cf) :=Composants(Cf)-C

d. Les invariants : les invariants associés aux éléments architecturaux de notre exemple sont illustrés par le tableau suivant :

Elément Architectural	**Invariants**
I1 : Invariants Configuration	-Une configuration est composée au moins d'une interface fournie ; -Un composant dans une configuration doit être relié au minimum à un connecteur ; -Un connecteur dans une configuration doit être relié au minimum à deux composants -Un composant ne peut être relié directement à un autre composant ; -Un connecteur ne peut être relié directement à un autre connecteur ;
I2 : Invariants Composant	- Un composant est constitué au moins d'une interface fournie ;
I4 : Invariants Connecteur	- Un connecteur est constitué au moins d'une interface fournie et une interface requise ; - Un connecteur est composé d'une glu ; - Un connecteur ne peut définir à la fois la propriétés d'exclusivité (**E,Ps-Ec)** et la propriété de dépendance faible (**Df**) ou forte multiple (**DFM**) - Un connecteur ne peut définir à la fois la propriétés d'exclusivité (**E,Es-Pc)** et la propriété de prédominance faible (**PRf**) **;**
I5 : Invariants Interface-connecteur	- Une interface connecteur doit appartenir au moins à un connecteur

Nous notons que les contraintes liées à la définition des propriétés sont intégrées comme invariants du concept Connecteur.

5.5.1.2 Exécution d'une évolution

Après la spécification de l'évolution le concepteur sélectionne l'élément architectural à faire évoluer et l'opération à appliquer sur cet élément. Il sélectionne dans notre cas l'élément **securityManager** et l'opération de **suppression**.

Cette sélection sera interceptée par le gestionnaire d'évolution (GE) comme étant l'événement d'évolution : *Supprimer-composant(securityManager : Composant, Client-serveur : Configuration)*. Il réagit

ensuite au vu des stratégies et des règles d'évolution définies, en exécutant les étapes suivantes :

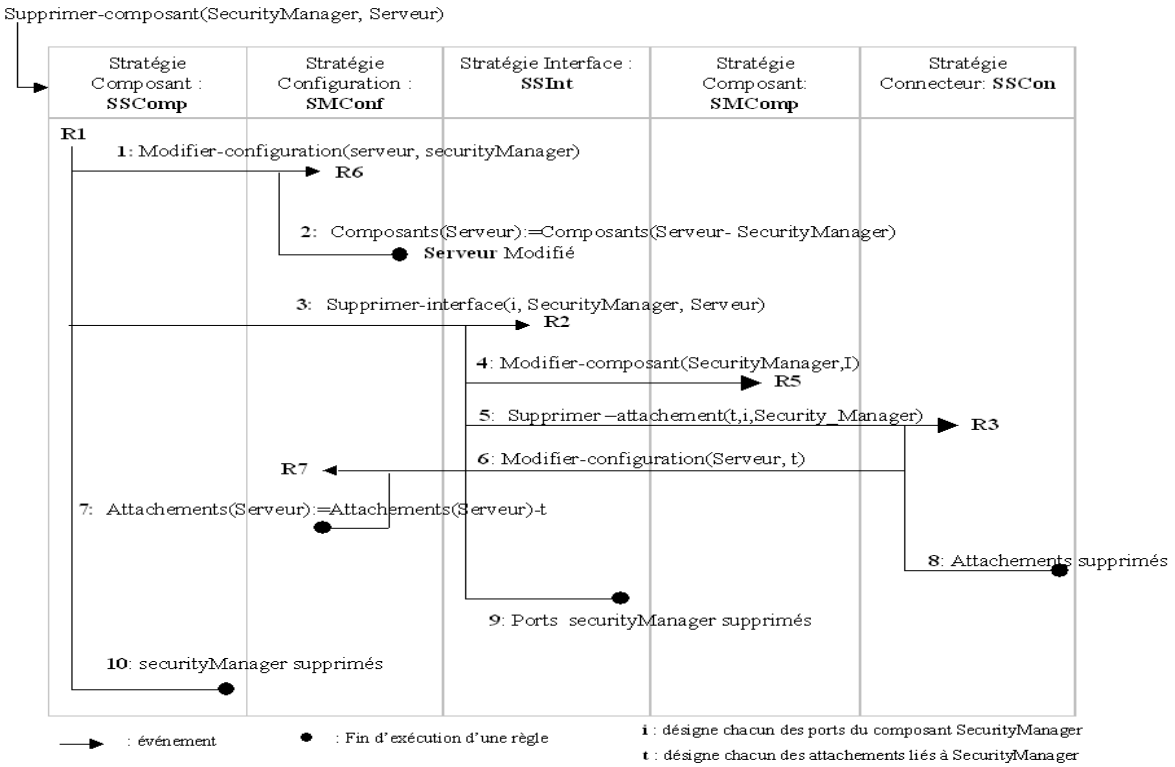

Figure 5.14 – Enchainement de l'exécution des règles d'évolution

a) **L'identification de la stratégie et de la règle d'évolution à exécuter** :

- Le *GE* identifie la stratégie d'évolution correspondante à l'élément et l'opération invoqués au travers l'événement reçu. Il sélectionne la stratégie *SSComp*.
- Dans la stratégie *SSCom*, il sélectionne la ou les règles d'évolution ayant leurs événements qui correspondent à l'évènement reçu et dont leurs conditions sont satisfaites. Dans notre exemple, il sélectionne la règle de suppression d'un composant *R1*.

b) Exécution d'une règle d'évolution :

Le *GE* déclenche l'exécution de la règle *R1* action par action. Nous montrons dans ce qui suit uniquement l'enchaînement des sélections des stratégies et des règles d'évolution.

1 : La première action de *R1* correspond à un nouvel évènement " modifier-configuration " qui sera intercepté par le GE.

2 : Le GE sélectionne la stratégie *SMConf* pour modifier la configuration du composant *Serveur* (supprimer le composant *securityManager* de la liste des composants du *Serveur*), via la règle *R6*.

3 : La deuxième action de *R1* correspond à un événement. La stratégie *SSInt* est sélectionnée pour supprimer l'interface du composant *securityManager*, via la règle *R2*.

4 : A l'exécution de la première action de **R2**, la stratégie *SMComp* est sélectionnée pour enlever l'interface à supprimer de la liste des interfaces du composant *SecurityManager* via la règles **R5**.

5 : La deuxième action de **R2** déclenche la suppression des attachements liés à l'interface du composant *securityManager*, via la règle **R3** de la stratégie *SSCon*.

6 : La première action de **R3** déclenche la modification de la configuration du composant Serveur pour enlever les attachements à supprimer de la liste des attachements de la configuration via la règle **R7** de la stratégie *SMConf* (la configuration du composant *Serveur* est modifiée).

8 : La dernière action de **R3** déclenche la suppression proprement dite des *attachements*.

9 : La dernière action de **R2** provoque la suppression proprement dite de l'interface du composant *securityManager*.

10 : La dernière action de **R1** supprime proprement dit le composant SecurityManager

L'exécution de ces règles induit donc à la suppression du composant **SecurityManager**, de ses interfaces et des attachements liés à ses interfaces.

c) Vérification de la cohérence :

Après l'exécution de cette évolution, le gestionnaire d'évolution lance la vérification de la cohérence. Les deux incohérences suivantes seront détectées, via l'invariant associés au concept connecteur :

- Le connecteur *ClearanceRequest* est lié à un seul composant ;
- Le connecteur *SecurityQuery* est lié à un seul composant.

Le concepteur doit alors corriger ces incohérences en déclenchant d'autres règles qui, soit supprimeront ces deux connecteurs, soit les relieront vers d'autres composants.

5.5.2 Exemple d'une évolution au niveau Application

Considérant une application Client/Serveur, instance de l'architecture précédente, avant évolution :

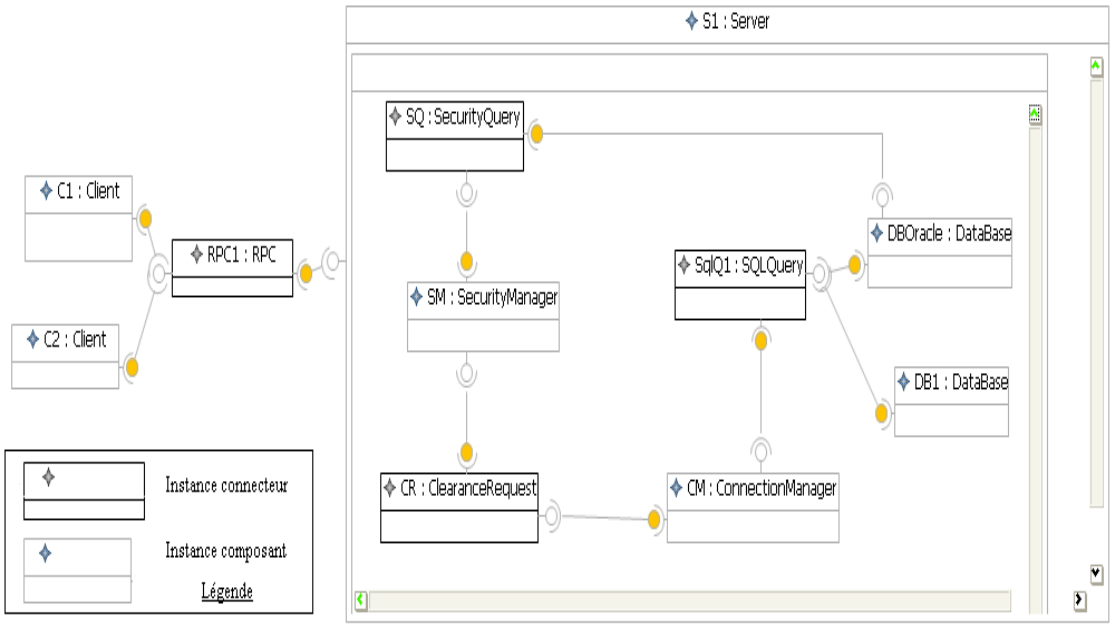

Figure 5.15 – Une application Client/serveur

Nous nous intéressons à l'évolution du Seveur *S1* instance du composant *Serveur*. L'évolution à répercuter sur cette application consiste à supprimer le composant *SM* instance du composant *Security-Manager* qui appartient au composant Serveur *S1*.

Selon le même principe présenté dans la section précédente, il faut en premier procéder à la spécification de l'évolution puis à son exécution.

5.5.2.1 Spécification de l'évolution de l'application Client/Serveur

Spécifier une évolution revient à spécifier les éléments architecturaux à faire évoluer, leurs stratégies d'évolution, leurs règles d'évolution et leurs invariants. Pour illustrer l'exemple d'évolution de l'application Client/Serveur, nous présentons dans ce qui suit, les éléments architecturaux qui interviendront dans cette évolution ainsi que des exemples de stratégies et de règles d'évolution.

a. Les éléments architecturaux

Nous présentons ici la liste des éléments architecturaux qui interviendront dans cette évolution (nous faisant abstraction des détails de description de ces éléments architecturaux).

Instance	Type instance
C1	Composant Client
C2	Composant Client
RPC1	Connecteur RPC
SQ	Connecteur SecurityQuery
SqlQ1	Connecteur SQLQuery
CR	Connecteur ClearanceRequest
SM	Composant SecurityManager
CM	Composant ConnectionManager
DB1	Composant DataBase
DBOracle	Composant DataBase

b. Stratégies d'évolution

Stratégies d'évolution	Configuration Server	Composant SecM	Connecteur SecQ	Interface SecM
Stratégie d'ajout : **Règles**				
Stratégie de suppression : **Règles**		**SSACompSecM** : RA1	**SSAConSecQ** : RA4,RA5	**SSAISecM** : RA2, RA4
Stratégie de modification : **Règles**	**SMAConfSer** RA6			
Stratégie de substitution : **Règles**				

c. Les règles d'évolution

Nous présentons dans les tableaux suivants, des exemples de règles d'évolution que nous utiliserons pour illustrer l'évolution d'une application. Pour faciliter la présentation des règles, nous avons adopté les abréviations suivantes :

SecM : le composant type SecurityManager ;

SecQ : le connecteur type SecurityQuery ;

ClR : le connecteur type ClearenceRequest.

RA1 : Règle de suppression Composant SecM	RA2 : Règle de suppression Interface Composant SecM
Evenement : Supprimer- composant-SecM(C : Composant-SecM, Cf :Configuration-Sever)	**Evenement** : Supprimer-interface-composant-SecM(P : Port-SecM C : SecM, Cf : Configuration-Server)
Condition : C ∈ Composants-SecM(Cf)	**Condition** :p ∈ ports (C) et ∃ N : ConnecteurU tel que source(N)= type(P)
Action : Modifier-configuration-Server(Cf,C) **Pour** tout I ∈ interfaces- composant-SecM(C) Supprimer-interface-composant-SecM (I, C,Cf) **Finpour** *C.Execute-supprimer-composant-SecM(Cf)*	**Action :** **Pour** tout n : N tel que I = source(n) ou I = cible(n) Supprimer-connecteurU-N(n,I,Cf) **FinPour** *I.Execute-supprimer- port-composant-SecM(C,Cf)*

RA3 : Règle suppression-connecteur-SecQ	**RA4 : Règle suppression interface-connecteur-SecQ**
Evenement : Supprimer-connecteur-SecQ(n : Connecteur-SecQ, I : port, Cf : Configuration-Server)	**Evenement** : Supprimer-interface-connecteur-SecQ (r : Role-SecQ, n : connecteur-SecQ)
Condition : I = source(n),	**Condition** :r ∈ roles(n)
Action :	**Action :**
Modifier-configuration-server(Cf, n,I) **Pour** tout j ∈ interface-connecteur-SecQ(n) Supprimer-interface-connecteur-SecQ(j,n) **Finpour** **Si dependence(n)**= **DFU** alors Supprimer-interface-composant-composant-type(cible(n)) (cible(n),composant(cible(n)) *n.Execute-supprimer-connecteur-SecQ()*	**Pour** tout t ∈ attachements(n) tel que r = source(t) ou r = cible(t) Supprimer-attachement-SecQ(t, r, n) **FinPour** *r.Execute-supprimer -role-SecQ(n)*

RA5 : Règle suppression Attachements SecQ	**RA6 : Règle modification Configuration Server**
Evénement :Supprimer-Attachement(t :attachement-SecQ I : interface-connecteur-SecQ, N :connecteur-SecQ)	**Evénement** : Modifier-configuration-Server(Cf : configuration-server, C : composant-SecM)
Condition : I ∈ source(t) ou I ∈cible(t)	**Condition** : C ∈ Composants-SecM(Cf)
Action : Modifier-configuration-server(configuration(N), t) *t.Execute-supprimer-attachement-SecQ(configuration(N))*	**Action** : Composants-SecM(Cf) :=Composants-SecM(Cf)-C

RA7 : Règle modification Configuration Server	**RA8 : Règle suppression Connecteur ClR**
Evenement :Modifier-configuration-server(Cf : Configuration -Sever : configuration, N : Connecteur-ClR	**Evenement** : Supprimer-connecteur-ClR(n :Connecteur-ClR, I : Port, Cf : Configuration-server)
Condition : N ∈ Connecteurs-ClR(Cf)	**Condition** : I = cible(n)
Action : Connecteurs-ClR(Cf) := Connecteurs-ClR(Cf) - n	**Action** : Modifier-configuration-server(Cf, n,I) **Pour** tout j ∈ interface-connecteur-ClR(n) Supprimer-interface-connecteur-ClR(j,n) Supprimer-interface-connecteur-ClR(j,n) **Finpour** Si **Predominance**(n)= **PRF** alors Supprimer-interface-composant-composant-type(source(N)) (source(n), composant(source (n))) *n.Execute-supprimer-connecteur-SecQ()*

5.5.2.2 Exécution d'une évolution

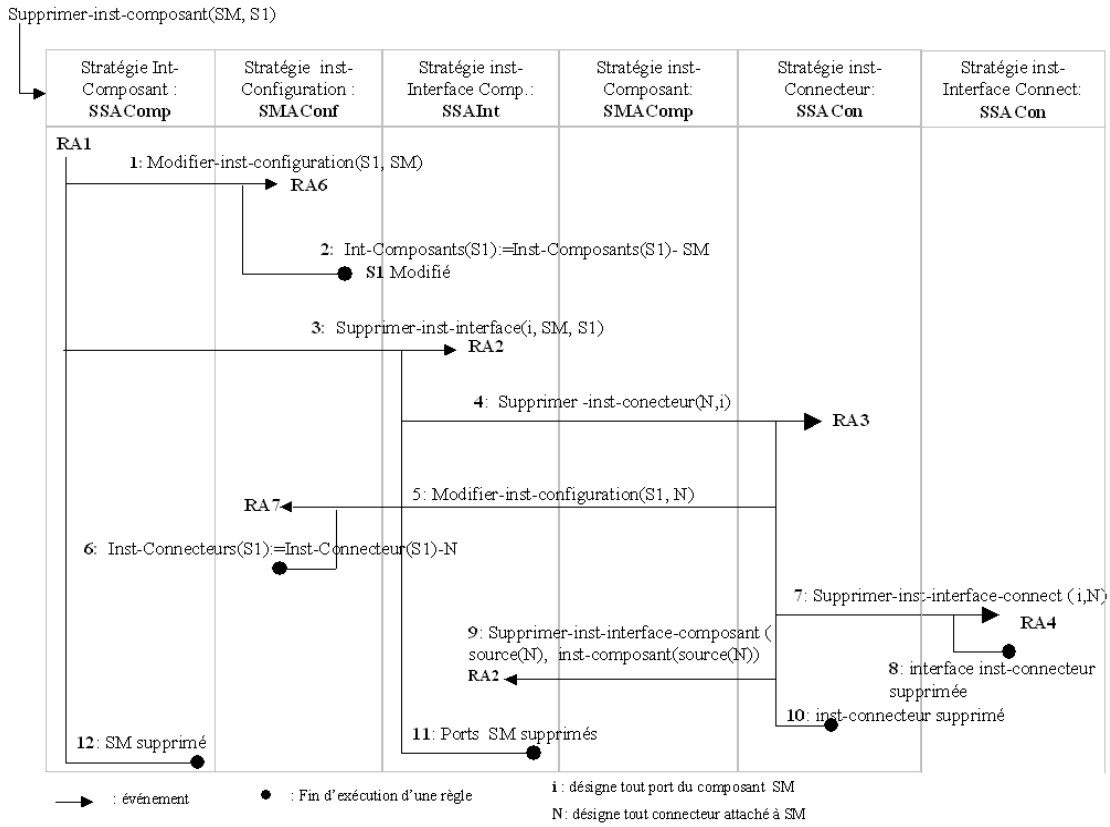

Figure 5.16 – Enchaînement de l'exécution des règles d'évolution

Le concepteur sélectionne l'élément à faire évoluer ainsi que l'opération à appliquer sur cet élément. Il sélectionne l'instance *SM* et l'opération de Suppression. Cette action sera interceptée par le gestionnaire d'évolution (GE) comme étant l'évènement d'évolution et réagit à cet évènement en exécutant les étapes suivantes :

a) Identification de la stratégie et de la règle à exécuter :

- Une fois l'événement de suppression est intercepté par le GE, il sélectionne la stratégie *SSAComp,* associé à l'instance d'un composant.
- Dans cette stratégie il sélectionne la ou les règles d'évolution correspondante à l'événement. La règle *RA1* sera alors sélectionnée.

b) Exécution des règles d'évolution :

La règle *RA1* sera exécutée action par action :

1 : La première action de *RA1* correspond à un événement qui sera intercepté par le GE et qui effectuera les sélections et les exécutions suivantes :

2 : La stratégie *SMAConfServer* est sélectionnée pour modifier la configuration du composant *S1* ;

3 : La stratégie *SSAIntSecM* est sélectionnée pour supprimer les interfaces de composant *SM* (SA, CQ) via la règle *RA2*.

4 : A l'exécution de la première action de *RA2*, la stratégie *SSAConSecQ* est sélectionnée pour supprimer les connecteurs attachés à l'interface de l'instance SM (les connecteurs *SQ*, *CR*), via les règles *RA3*, et *RA8* (pour plus de lisibilité de la figure nous avons illustré uniquement l'exécution de la règle *RA3*). Ces deux actions permettent de vérifier d'abord les propriétés sémantiques du connecteur à supprimer.

5-6 : La première action de *RA3*, déclenche la règle *RA7* pour modifier la configuration du composant *S1* par la suppression du connecteur *SQ* de la liste des connecteurs de la configuration de *S1*.

7-8 : La deuxième action de *RA3* déclenche la règle *RA4* pour la suppression de l'interface du connecteur à supprimer.

9 : La troisième action déclenche la suppression de la source du connecteur *SQ* (le port SMg).

10 : La dernière action déclenche la suppression proprement dite du connecteur *SQ*.

11 : La deuxième action de *RA2* déclenche la suppression proprement dite des ports du composant *SM*, le port *SA* et le port *CQ*.

12 : La dernière action de **RA1** déclenche la suppression proprement dite du composant *SM*.

La suppression du composant *SM* déclenche la suppression de son interface, des connecteurs attachés à ce composant et des ports des composants avec lesquels il interagit. En, effet grâce aux propriétés de **dépendance** et de **prédominance** définies sur les connecteurs *CR* et *SQ*, la suppression du composant *SM* a induit à la suppression des ports des composants avec lesquels il interagit (ces ports deviennent inutiles à la suppression du composant *SM*).

Nous avons montré au travers des sections précédentes l'application du modèle SAEV sur une architecture et une application Client/serveur en tenant compte des propriétés sémantiques définies pour les connecteurs au niveau architectural.

Dans la section suivante nous montrons une illustration de la prise en compte des propriétés sémantiques au niveau Méta.

5.5.3 Prise en compte des propriétés sémantiques au niveau Méta : illustration

Nous avons montré dans la section 4.4 du chapitre précédent la possibilité de réifier les liens entre les concepts architecturaux de COSA et de les considérer comme des entités de premières classe que nous avons appelé *connecteurs Niveau-Méta*. Nous avons montré que nous pouvons aussi associer les mêmes propriétés sémantiques à ces connecteurs Niveau-Méta. Ainsi, la propagation des impacts peut être améliorée au niveau *Application* et au niveau *Architectural*. Par exemple :

- **Au niveau Architectural** : dans l'architecture Client/Serveur définie en section 5.5.1, la règle d'évolution **R1** spécifie que la suppression d'un composant implique la suppression de son interface.

Cependant si le lien de composition entre le concept *composant* et le concept *interface* est réifié autant *connecteur Niveau-Méta*, il suffit alors d'associer la propriété de *dépendance unique* à ce connecteur qui indiquera que la suppression d'un composant implique la suppression de l'interface de son composant. Si nous voulons ensuite exprimer le fait que la suppression d'un composant n'implique pas la suppression de ses interfaces, alors, il suffit de modifier la sémantique associée à ce connecteur de composition entre un composant et son interface.

- **Au niveau Application** : les propriétés sémantiques des connecteurs Niveau-Méta peuvent être redéfinies au niveau Architectural et intervenir alors à l'évolution de toute application créée à partir de ce niveau Architectural. Par exemple, si un *port requis* est indispensable pour un *composant*, la suppression de ce *port* implique la suppression du *composant* lui-même. Ceci peut être exprimé par exemple par une propriété de *prédominance* associée au lien de composition entre ce *port* et ce composant.

En se basant ainsi sur les propriétés sémantiques au niveau des *connecteurs utilisateur*, des *connecteurs de construction* et/ou des *connecteurs Niveau-Méta*, l'ensemble des propagations sera alors véhiculé par les *connecteurs*. Le concepteur n'est pas alors contraint à spécifier les impacts d'une évolution au niveau des règles d'évolution, mais ces derniers seront automatiquement véhiculés via les propriétés sémantiques des connecteurs.

Rcn2 : Règle de suppression connecteur	**Rcn3 : Règle de suppression interface connecteur**
Evenement : Supprimer-connecteur(N : connecteur)	**Evenement** : Supprimer-interface-connecteur(I :interface-connecteur)
Condition : N $\exists$	**Condition** : I $\exists$
Action : Pour tout l : connect-Niveau-Méta tel que N = source(l) ou N = cible(l) alors supprimer-lien (l,N) finpour *C.Execute-supprimer-connecteur()*	**Action** : Pour tout l : connect-Niveau-Méta tel que I = source(l) source(l) ou I = cible(l) alors supprimer-lien (l,I) finpour *I.Execute-supprimer-interface-connecteur()*

A titre d'illustration la règle Rcn2 définie au niveau Architectural décrit la suppression d'un *connecteur*. Cette règle indique qu'à la suppression d'un connecteur, tous les liens de type connecteurs Niveau-Méta, auxquels il est attaché seront supprimés. Avant la suppression d'un connecteur de Niveau-Méta auquel participe le connecteur, on vérifie ses propriétés sémantiques. Si à titre d' exemple, il s'agit d'un connecteur de composition qui relie ce connecteur à son interface et si ce connecteur défini la propriété de *dépendance forte unique* la suppression de ce connecteur implique la suppression de son interface. La suppression de l'interface du connecteur induira à la suppression des connecteurs de type Niveau-Méta auxquels elle est reliée et de la même façon, il y aura la vérification des propriétés de ces connecteurs Niveau-Méta, ainsi de suite jusqu'a ce que toutes les propagations soient effectuées.

5.6 Conclusion

Nous avons consacré ce chapitre à l'illustration des propriétés sémantiques ainsi que l'utilisation du modèle SAEV au travers de l'ADL COSA. Nous avons présenté au préalable une brève description de l'ADL COSA ainsi que l'exemple de l'architecture Client/Serveur sur lequel nous nous sommes basés pour illustrer nos propositions. Nous avons présenté ensuite l'illustration des propriétés sémantiques dans le cadre de COSA. Nous avons montré que ces propriétés sémantiques pouvaient être associées à

un connecteur utilisateur comme elles pouvaient être associées aux connecteurs de construction (attachements, bindings ou liens d'utilisation). Nous avons montré ensuite la réification des liens structuraux entre les concepts du méta modèle COSA comme étant des connecteurs particuliers que nous avons appelé connecteurs Niveau-Méta. Nous avons montré que les propriétés sémantiques pouvaient être associées aussi à ces connecteurs Niveau-Méta.

Dans la deuxième partie de ce chapitre nous avons illustré l'utilisation du modèle SAEV au travers d'une architecture et d'une application Client/Serveur décrite en COSA, en tenant compte des propriétés sémantiques. Nous avons montré ensuite l'apport de considérer des propriétés sémantiques proposées au niveau Méta.

Après avoir présenté la description conceptuelle du modèle d'évolution SAEV, nous proposons dans le chapitre suivant une démarche possible de son implémentation.

CHAPITRE 6
Expérimentations

6.1 Introduction

Nous avons présenté dans le chapitre 3, SAEV, un modèle d'évolution pour les architectures logicielles à base de composants. Nous avons illustré ensuite dans le chapitre 5 une utilisation du modèle sur une architecture Client/Serveur décrite en utilisant l'ADL COSA. Nous avons mis l'accent jusque là sur la description conceptuelle du modèle SAEV. Le but de ce chapitre est d'illustrer une approche possible d'implémentation du modèle SAEV. L'idée principale retenue pour cette approche est d'exploiter les fondements de la théorie de transformation de graphes et ses supports pour cette implémentation. L'avantage est alors de réutiliser une approche éprouvée dont plusieurs outils et supports sont disponibles.

Nous présentons dans la première section de ce chapitre le rapprochement que nous avons établi entre l'évolution architecturale telle qu'abordée par SAEV et la transformation de graphes en soulignant les avantages de la théorie de transformation de graphes pour le modèle SAEV. Nous présentons ensuite l'outil AGG (Attributed Graph Grammar) [83], un outil de transformation de graphes que nous adoptons comme support pour l'implémentation du modèle SAEV. Nous présentons premièrement une brève description de l'approche algébrique [18, 26] sur laquelle repose AGG. Nous montrons ensuite les concepts supplémentaires adoptés dans AGG et que nous jugeons nécessaires à réutiliser dans le cadre de SAEV. Dans la troisième section, nous illustrons comment nous projetons les concepts de SAEV sur AGG. Le but de cette projection est d'identifier les concepts d'AGG qui peuvent être substitués ou qui peuvent intervenir dans l'implémentation des concepts de SAEV. Dans la quatrième section nous montrons les choix techniques adoptés pour le prototypage de SAEV en tenant compte des choix d'implémentation de l'outil AGG, avant de faire le point sur ce travail d'expérimentation.

6.2 Evolution architecturale dans SAEV et transformation de graphes : similitudes

L'idée que nous exploitons pour l'implémentation de SAEV est de ramener la description d'une architecture logicielle à une représentation sous forme d'un graphe, pour ensuite exploiter les concepts et les outils de la transformation de graphes comme support à l'implémentation du modèle SAEV, notamment de son processus d'évolution et de ses propagations.

Les graphes sont des structures fréquemment utilisées dans le domaine de l'ingénierie du logiciel pour la représentation de tout artefact logiciel (code source, bases de données, modèles de conception, etc.). L'avantage qu'offre les graphes est leur capacité à illustrer la description de tout artefact logiciel, quelle que soit la diversification de ses entités manipulées, par une représentation manipulant deux types d'entités : les *nœuds* et les *arcs*. Les nœuds sont utilisés pour représenter généralement les entités de première classe de l'artefact à modéliser et les arcs pour modéliser les relations entres ces entités.

Plusieurs techniques et approches sont alors proposées permettant de manipuler et d'exploiter les graphes. Parmi ces approches, on trouve les approches de transformation de graphes. Le principe de la transformation de graphes est de partir d'un graphe de départ et, en appliquant un ensemble de règles de transformation, d'aboutir à un graphe d'arrivé (le graphe transformé). L'évolution architecturale telle qu'elle est considérée dans SAEV peut être ainsi assimilée à une transformation de graphes en considérant la représentation d'une architecture logicielle sous forme d'un graphe. Nous nous intéressons dans cette expérimentation à la transformation de graphes basée sur une approche algébrique [18, 26]. Une telle approche nous offre des avantages au niveau de la spécification des concepts et du processus d'évolution de SAEV :

- Au niveau des **concepts** : réutiliser les concepts de la théorie de la transformation de graphes pour spécifier les concepts de SAEV, notamment les *règles d'évolution* et les *invariants*, nous offrira l'avantage de s'appuyer sur des concepts d'une théorie formelle et éprouvée.
- Au niveau **processus** : le processus d'évolution dans SAEV peut être assimilé au processus de transformation de graphes. Nous pouvons ainsi réutiliser les concepts qui implémentent le processus de la transformation de graphes pour spécifier le processus d'évolution de SAEV. Il s'agit également de tirer profit des mécanismes de vérification de la cohérence fondé sur cette théorie de transformation de graphes.

Ainsi pour l'implémentation de SAEV, nous nous appuierons sur un outil qui implémente la transformation de graphes. Nous avons opté pour l'outil AGG (Attributed Graph Grammar) [83, 86]. Des similitudes entre le modèle SAEV et AGG nous ont orientées vers ce choix. Nous les reprendrons dans la section 6.4.

6.3 AGG (Attributed Graph Grammar) : un outil de transformation de graphes

AGG (Attributed Graph Grammar) est un outil de transformation de graphes au moyen d'une *approche algébrique*. Nous présentons quelques fondements basiques de cette approche algébrique pour la transformation de graphes avant de présenter l'outil AGG.

6.3.1 Approche algébrique : définitions et notions de base

L'approche algébrique pour les grammaires de graphes a été proposée par H. Ehrig, M. Pfender et H.J.Schneider dans le but de généraliser les grammaires de Chomsky aux graphes [27]. L'idée principale est de généraliser le principe de la concaténation de termes à des constructions de graphes. Ce qui permet ainsi d'exprimer la réécriture de graphes par des constructions de graphes. Nous apportons dans ce qui suit les définitions des concepts de base manipulés dans cette approche algébrique, ainsi que ses approches pour la transformation de graphes :

6.3.1.1 Concepts de base :

La transformations de graphes opère sur des graphes via un ensemble d'opérations que nous présentons dans ce qui suit :

a. Graphe : un *graphe* est un ensemble fini de *nœuds* reliés par des *arcs*.

b. Graphe dirigé annoté : un graphe G *dirigé* et dont les *arcs* et les nœuds sont annotés, est un graphe dont les arcs sont orientés (à chaque arc on peut associer un nœud source et un nœud cible) et dont les nœuds et les arcs sont labelisés (à chaque arc et à chaque nœud est associé un label).

c. Sous-graphe un graphe H est *sous-graphe* de G, si l'ensemble des *nœuds* de H est inclu ou égal à l'ensemble des *nœuds* de G, l'ensemble des *arcs* de H est inclue ou égale à l'ensemble des *arcs* de G, chaque *arc* a le même *nœud* source, même *nœud* cible et même label dans H et dans G et chaque *nœud* a le même label dans G et dans H.

L'approche algébrique pour la transformation de graphes s'appuie sur des opérations dont les plus importantes dans le cadre de cette expérimentation : le *morphisme*, la *production* et la *dérivation*. Ces opérations nous permettrons de comprendre comment est effectuée la transformation de graphes.

d. Morphisme de graphes : *un morphisme de graphes* est une famille de relations entre les ensembles porteurs du graphe *source* vers les ensembles porteurs de graphe *cible* préservant la structure des graphes ainsi que les labels [18]. Un morphisme de graphes g est défini comme suit :
$g : G \rightarrow L$ est une paire de fonctions définies ainsi :

$$g_V : V_G \rightarrow V_L$$

$$g_E : E_G \rightarrow E_L$$

Où g_V est la fonction de correspondance entre les nœuds, et g_E la fonction de correspondance entre les arcs.

e. Production : une production est composée de deux graphes, un graphe de gauche et un graphe de droite. La production définie ainsi une correspondance partielle entre les éléments du graphe de gauche et du graphe de droite. L'application d'une production à un graphe, permet de déterminer les éléments (nœuds et arcs) de ce graphe qui seront préservés, supprimés ou crées via cette production.

Pour pouvoir appliquer une production P à un graphe G, il faut d'abord identifier dans le graphe G les occurrences correspondantes à la partie gauche de la production P. Ces occurrences sont appelées *Match* (correspondance). Un *match* est donc un morphisme entre la partie gauche de la production P et le graphe G.

f. Dérivation directe : la dérivation décrit l'application d'une production à un graphe. La notation $G \overset{p,m}{\Longrightarrow} H$ dénote alors une *dérivation directe* telle que l'application de la production P suivant le match m à G permet d'obtenir le graphe dérivé H. Appliquer une production P à un graphe G revient à supprimer tout élément de G qui concorde avec un élément de la partie gauche de P et qui n'a pas un élément correspondant dans sa partie droite et à ajouter au graphe G, tout élément de figurant dans la partie droite qui n'a pas de correspondant dans la partie gauche. Un exemple de dérivation directe est illustré dans la figure suivante :

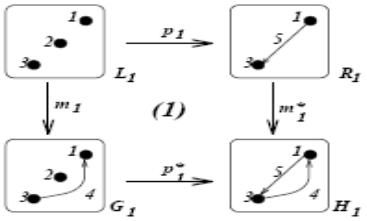

Figure 6.1 – Exemple d'une dérivation

La production P_1 part de trois nœuds et produit deux nœuds reliés par un arc. En appliquant cette production au graphe G_1 suivant le match $m1$, le nœud n°2 sera supprimé du graphe G et l'arc n°5 sera ajouté.

6.3.1.2 Approches de transformation de graphes

Deux approches sont proposées pour appliquer une transformation de graphes : l'approche *Double PushOut* et l'approche *Single PushOut* [18].

a. Approche Double PushOut (DPO) : dans cette approche une production est définie par une paire de morphismes de graphes à partir d'un graphe d'*interface* K :$L \cap R$ et la dérivation directe consiste en deux diagrammes (1) et (2) illustrés dans la figure suivante :

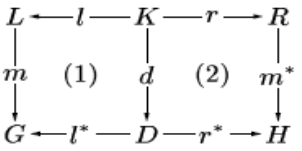

Figure 6.2 – Approche Double PushOut

Pour appliquer une production dans l'approche Double PushOut, le *match* m doit satisfaire la condition d'application (*gluing condition*). La **Gluing condition** est composée de deux parties la *Dangling condition* et l'*Identification condition*. La *Dangling condition* impose que si une production P spécifie la suppression d'un nœud, alors cette production doit aussi spécifier la suppression de tous les arcs adjacents à ce nœud.

La condition d'identification indique que chaque élément de G qui doit être supprimé par l'application de la production P, doit avoir une seule image dans L.

b. Approche Single PushOut (SPO) : l'approche Single PushOut décrit l'application directe d'une production P selon un match m. Cette approche n'impose pas la vérification de la condition d'application (gluing condition).

Malgré tout, de part la définition même d'un graphe, un arc ne peut exister sans nœuds à ses extrémités. Une stratégie de suppression d'un tel arc est alors adoptée, même si cette suppression n'est pas définie dans la production.

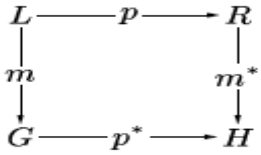

Figure 6.3 – Approche SPO

Nous avons apporté dans cette section les définitions et notions de base relatives à l'approche algébrique pour la transformation de graphes. Nous montrons dans la section suivante comment ces concepts et notions sont adoptés dans l'outil AGG que nous exploitons pour implémenter le modèle SAEV.

6.3.2 Brève présentation de l'outil AGG

Nous présentons brièvement dans la section suivante les concepts de base d'AGG que nous considérons intéressants à exploiter dans le cadre de l'implémentation de SAEV, ainsi que son mécanisme pour la transformation de graphes.

6.3.2.1 Concepts de base d'AGG :

a. Graphe : un graphe dans AGG peut être dirigé, attribué typé et labellisé :

- *Dirigé* : chaque arc qui relie deux nœuds peut être doté d'une direction.
- *Attribué* : les nœuds et les arcs peuvent être dotés d'attributs (couple de variable-valeur destiné à représenté une valeur nommée).
- *Typé et labellisé* : les arcs et les nœuds sont caractérisés au moyen d'un type.

b. Graphe de types : dans un graphe il n'y a pas de moyen pour exprimer des contraintes sur les connexions entre les arcs et les nœuds, tel que par exemple un nœud portant un label x ne peut être source d'un arc de label y. De telles contraintes peuvent être exprimées sur un *graphe de types*. Le lien entre un graphe et un graphe de types peut être comparable au lien entre un méta modèle et un modèle. Ainsi, un graphe doit être toujours conforme à son graphe de types.

Ce principe de distinguer ces deux niveaux lors de la transformation de graphes est donc conforme au principe de SAEV qui distingue la description de l'architecture à faire évoluer (le niveau *Architectural*, respectivement le niveau *Application*) de la description du niveau supérieur qui a permis la spécification de cette architecture (le niveau *Meta*, respectivement le niveau *Architectural*).

De plus le graphe de types permet de définir différents types de nœuds et différents types d'arcs ainsi des contraintes pour relier ces différents nœuds. Le graphe de types permet ainsi de minimiser la définition des contraintes sur tout graphe crée à partir de ce graphe de types.

Nous illustrons dans les deux figures suivantes le *graphe de types* représentant les concepts de l'ADL COSA (niveau Méta) qui nous a servi dans le chapitre 5 pour l'illustration, et le graphe représentant l'exemple d'illustration de l'architecture Client/Serveur (cf. section 5.3 du chapitre 5). Ces graphes de types et graphes ont été spécifiés au travers de l'interface graphique d'AGG.

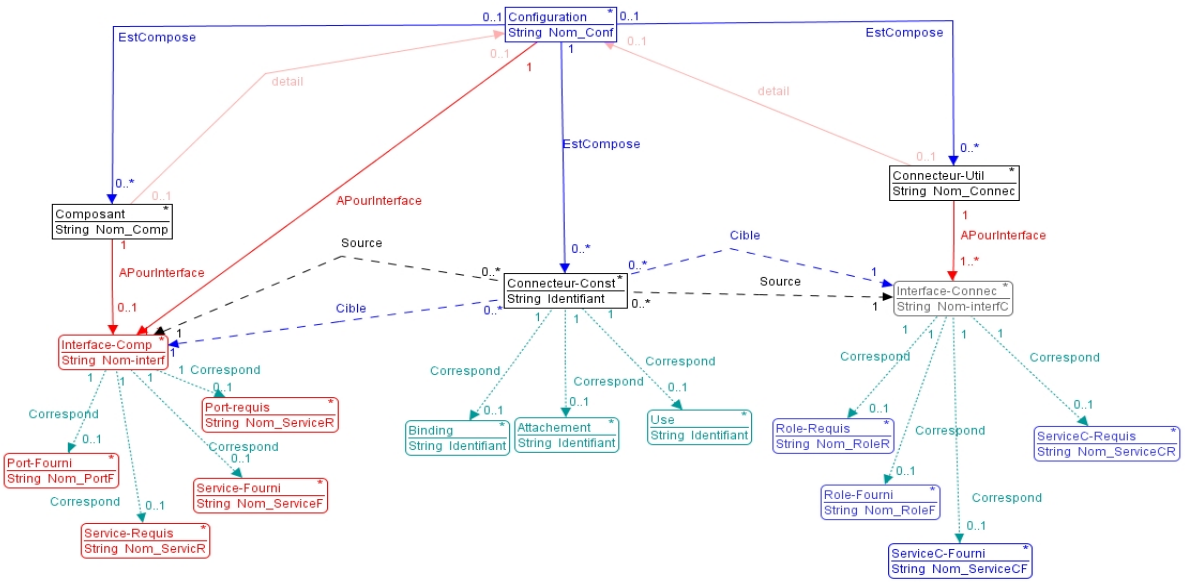

Figure 6.4 – Graphe de Types de l'ADL COSA

Dans cette représentation chaque concept réifié dans COSA est décrit sous forme d'un nœud, et chaque lien entre ces concepts est modélisé sous forme d'un arc.

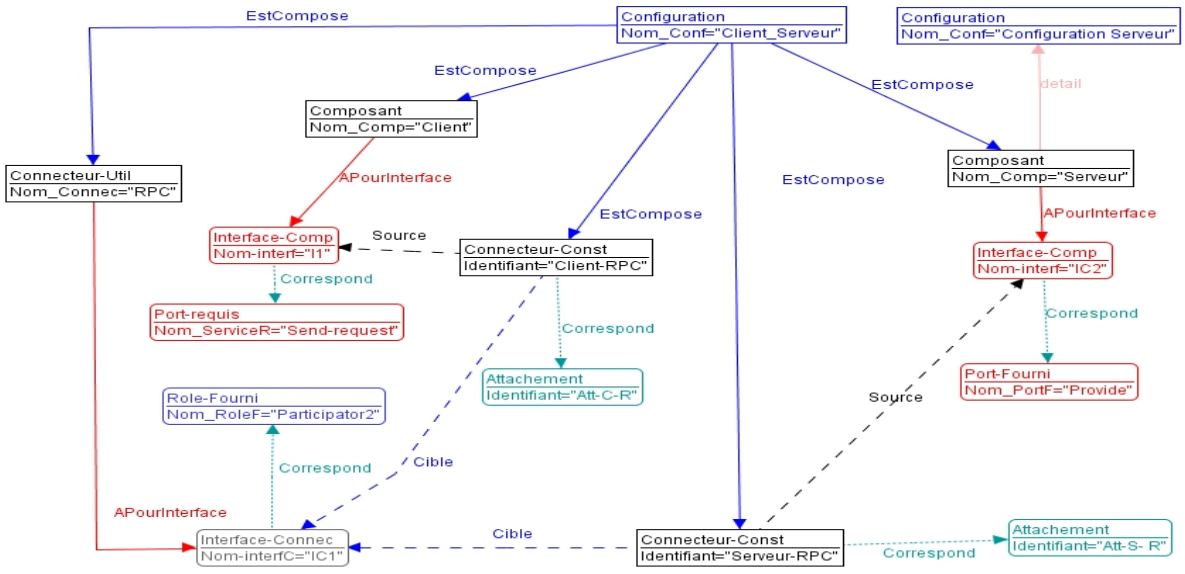

Figure 6.5 – Architecture Client/Serveur

La figure illustre une architecture Client/Serveur conforme au graphe de types de la figure précédente. Ainsi chaque élément à faire évoluer est représenté sous forme d'un nœud. Les liens entre ces éléments sont représentés sous forme d'arcs.

c. Règle de transformation : le mécanisme de transformation dans AGG est exprimé au moyen de *Règles de transformation* composées de deux morceaux de graphes. Un premier morceau de graphe destiné à effectuer la mise en concordance avec le graphe de départ appelé graphe de *gauche* et un autre morceau de graphe qui décrit la modification se nomme graphe de *droite.* La figure suivante illustre un exemple de règle de suppression d'un composant *Client* dans une architecture *Client/Serveur*.

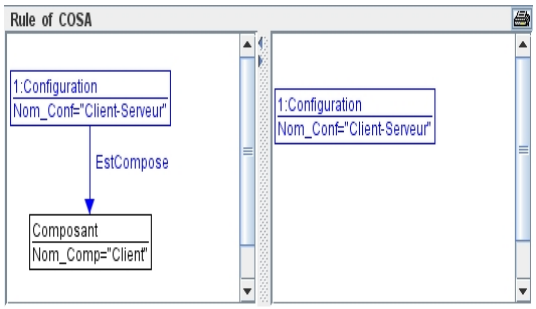

Figure 6.6 – Règle de suppression d'un composant Client

La mise en correspondance entre les éléments du graphe de *gauche* et ceux du graphe de *droite* est illustrée au moyen d'une numérotation. Ceci permet d'identifier à quel élément du graphe de *droite* correspond un élément du graphe de *gauche* pour la transformation. Une règle de transformation correspond à la notion de *production* définie précédemment.

d. Condition d'Application Négative : une Condition d'Application Négative (NAC : Negative Application Condition), est destinée à vérifier l'absence du morceau de graphe qu'elle définie, dans le graphe de *départ*. Elle exprime en effet, une condition qui ne doit pas être satisfaite dans le graphe de *départ* pour pouvoir exécuter la Règle de transformation à laquelle est associée. Les NACs permettent d'enrichir les règles de transformation.

c. Contraintes de consistance Graphique (Graphical Consistency Constraints) : une contrainte de consistance (CCG) sur un graphe G est composée de deux parties, la partie gauche est appelée *prémisse* et la partie droite est appelée *conclusion*. La partie gauche et la partie droite étant des sous-graphes de G [86]. Une contrainte peut avoir plusieurs conclusions.

Définir une contrainte de consistance c entre P et C revient à dire que tout prémisse P dans G possède une image C par c dans G.

Dans la figure suivante nous illustrons une contrainte qui impose qu'un connecteur de construction de COSA doit toujours avoir pour *source* une *interface composant* et pour *cible* une *interface connecteur*.

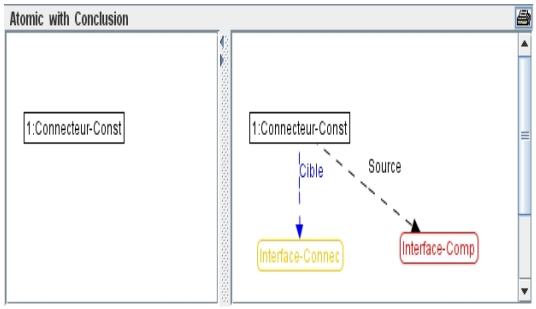

Figure 6.7 – Exemple de contrainte

Nous pouvons constater que ces contraintes peuvent nous servir à la spécification des invariants du modèle SAEV.

f. Conditions de cohérence (Consistency Conditions)

Les conditions de cohérence sont des expressions booléennes construites sur la base des GCC. Par exemple si on considère deux contraintes $C1$ et $C2$ d'un graphe, une contrainte de cohérence peut être ($!C1$ & $C2$), qui se lit non $C1$ et $C2$.

6.3.2.2 Processus de transformation de graphes dans AGG

Le processus de transformation dans AGG s'effectue en partant d'un graphe initial, le graphe de *départ*, pour produire un graphe transformé appelé graphe *modifié*, en appliquant une ou plusieurs *règles de transformation* qui sont aussi exprimées par deux morceaux de graphes. Lorsqu'une règle de transformation est appliquée à un graphe de *départ*, une ou plusieurs mise en concordance sont effectuées entre le graphe de *gauche* de la règle de cette transformation et le graphe de *départ*.

Pour le processus de transformation, AGG adopte une approche *Single PushOut*. Ainsi, une règle de transformation qui spécifie la suppression d'un nœud et non pas la suppression de ses arcs adjacents est applicable. La stratégie de suppression de tels arcs est adoptée même si elle n'est pas spécifiée dans la règle de transformation. Pour l'identification, soit elle est définie par l'utilisateur au moyen d'une numérotation soit elle est définie par défaut dans AGG de manière à favoriser la suppression des nœuds et des arcs qui ne sont pas identifiés en cas de conflit. La transformation de graphe dans AGG s'appuie sur les notions définies dans les sections précédentes *match*, *dérivation*.

Nous avons présenté dans cette section les concepts d'AGG que nous jugeons importants à considérer dans l'implémentation de SAEV. Dans ce qui suit, nous montrons comment les concepts de SAEV peuvent être projetés vers ces concepts d'AGG.

6.4 SAEV : surcouche d'AGG

Pour exploiter l'outil AGG comme support d'implémentation pour le modèle SAEV, nous identifions d'abord les concepts d'AGG qui peuvent être intégrés ou substitués à des concepts du modèle SAEV, ainsi que le processus d'AGG qui peut être réutilisé dans le cadre de SAEV. Il s'agit en effet de définir les projections possibles entre les concepts de ces deux modèles.

6.4.1 Projection des concepts de SAEV

Dans cette section, nous reprenons les concepts de SAEV. Pour chaque concept nous identifions le concept d'AGG, quand il existe, vers lequel il peut être projeté. Autrement ce concept devra être crée.

6.4.1.1 Concept Element Architectural

Le concept d'*Elément Architectural* réifie dans SAEV l'élément architectural à faire évoluer. Dans les principes évoqués dans SAEV pour faire évoluer une architecture logicielle décrite à un niveau d'abstraction, il faut identifier les concepts du niveau d'abstraction supérieur qui a permis la spécification de cette architecture.

Cette philosophie de SAEV est adoptée dans AGG sur deux niveaux d'abstraction, en distinguant le concept du *graphe de types* de celui du *graphe*.

- Pour faire évoluer une architecture au niveau Architectural, le *graphe de types* sera utilisé pour exprimer les concepts de l'ADL (c'est à dire du niveau Méta) qui a servi à la spécification de l'architecture, et le *graphe* pour modéliser l'architecture à faire évoluer.
- Pour faire évoluer une application, le *graphe de types* décrira alors une architecture du niveau Architectural, et l'application sera décrite par un *graphe* conforme à ce *graphe de types*.

Le concept de Elément architectural est projeté vers soit un *nœud* d'un *graphe* ou *nœud* d'un *graphe de types*, selon le niveau d'abstraction de l'architecture à faire évoluer.

6.4.1.2 Concept Stratégie Evolution

Ce concept est proposé par SAEV pour encapsuler les règles d'évolution de tout élément architectural. Ce concept n'a pas de concept correspondant dans AGG. Il sera ainsi crée.

6.4.1.3 Concept Règle Evolution

L'évolution de l'architecture logicielle est opérée par SAEV, au travers des *règles d'évolution*. Une règle d'évolution permet en effet, de décrire l'opération d'évolution à appliquer sur un élément architectural donné, les conditions nécessaires pour le faire ainsi que les éventuels impacts de cette opération sur les autres éléments architecturaux (cf. 3.3.2.4 section du chapitre3). La partie action d'une règle correspond soit à un *Evénement* soit à une *Opération élémentaire*. C'est l'exécution de l'opération élémentaire qui provoque l'évolution proprement dite d'un élément architectural.

Le rôle de l'*Opération élémentaire* est ainsi comparable à celui des *Règles de transformation* d'AGG décrivant les transformations sur un graphe (une évolution dans le modèle de SAEV). Nous proposons d'implémenter chaque *Opération élémentaire* à l'aide des règles de transformation au sens d'AGG. Ainsi chaque Opération d'évolution sera spécifiée par deux morceaux de graphes. Le concepteur peut aussi utiliser les NACs pour enrichir ces règles de transformation.

AGG considère aussi la notion d'évènement et d'interception d'évènements. Nous proposons ainsi de projeter le concept Evenement de SAEV vers celui d'AGG (GraEvent), pour pouvoir réutiliser le mécanisme d'interception des évènements d'AGG.

Le concept de Condition sera directement crée au niveau de SAEV.

6.4.1.4 Concept Invariant

Ce concept modélise dans SAEV toute contrainte qui doit être respectée par l'élément architectural tout au long de son cycle de vie. AGG offre des concepts pour exprimer des *contraintes* (cf. section 6.3.2.1) sur le graphe à transformer. Nous réutiliserons le concept de contraintes d'AGG pour spécifier les invariants et pour bénéficier du mécanismes de vérification de la cohérence offert par AGG.

Les différentes correspondances que nous avons établies entre les concepts de SAEV et ceux dans AGG sont exprimées par le tableau suivant :

Concepts de SAEV	***Concepts d'AGG***
Element Architectural	Nœud du graphe et/ou du graphe de types
Evenement	Evenement d'AGG
Opération élémentaire	Règle de transformation
Invariants	Contraintes AGG

Table 6.1 – Projection des concepts de SAEV sur AGG

6.4.2 Processus de SAEV et AGG

Le processus opératoire de SAEV est décrit dans la section 3.3.3 du chapitre 3. Nous illustrons dans cette section l'intégration des fonctionnalités d'AGG dans le processus de SAEV. En comparant le processus opératoire de SAEV et celui d'AGG, les fonctionnalités d'AGG peuvent intervenir dans 3 étapes : à l'interception des événements, à l' étape de l'*exécution d'une règle d'évolution* et à l'étape de la *vérification de la cohérence*.

- *A l'interception des évènements* : en se basant sur le concept d'évènement manipulé dans AGG, nous pouvons réutiliser le mécanisme d'interception des événements d'AGG au niveau Gestionnaire d'évolution pour l'interception des évènements d'évolution.
- *A l'exécution d'une règle d'évolution* : l'exécution d'une règle d'évolution dans SAEV se fait action par action. Si une action correspond à un *Événement*, une autre règle sera déclenchée pour propager les impacts. Si une action correspond à une *Opération élémentaire*, cette dernière sera empilée jusqu'à ce toutes les règles de propagations engendrées par la règle au cours d'exécution soit exécutées.

  Une fois que toutes les règles de propagation sont exécutées, le gestionnaire d'évolution lance l'exécution de ces opérations d'évolution empilées. Cette exécution correspond en effet à lancer la transformation dans AGG qui se chargera alors d'effectuer les transformations nécessaires sur le graphe en fonction des règles de transformation décrivant les opérations élémentaires.
- *A la vérification de la cohérence* : l'outil AGG dispose d'une fonctionnalité de vérification de la cohérence qui se base sur les contraintes spécifiées. Lancer la vérification de la cohérence revient donc à lancer la vérification de la cohérence au sens de AGG, en réutilisant les classes nécessaires.

Nous avons présenté dans cette section les projections que nous établissons entre les concepts et processus de SAEV et ceux d'AGG. Ces projections montrent comment peuvent être spécifiés les concepts de SAEV en se basant sur ceux d'AGG. Nous avons montré l'apport de cette projection notamment pour la spécification des *invariants* et des *opérations d'évolution*, ainsi que pour le réutilisation du mécanisme

de *vérification de la cohérence*. Nous apportons dans la section suivante quelques étape de l'implémentation de SAEV sur AGG.

6.5 Choix techniques d'implémentation

Le travail d'implémentation de SAEV sur AGG est réalisé dans le cadre du stage du Master recherche de Xavier Segnard [74]. Nous apportons dans ce qui suit une vue sur l'architecture de l'outil AGG, et le modèle d'implémentation de SAEV sur AGG.

6.5.1 Une vue sur l'architecture de l'outil AGG

L'outil AGG comporte trois parties dissociées en couches logicielles :

- Au plus bas niveau, on trouve l'AGG-Engine. C'est le moteur de transformations AGG. Cette boite à outils peut fonctionner de manière autonome et indépendante en vue d'effectuer des transformations. Elle est directement utilisable par programmation. Elle permet entre autres :
  - la construction et la manipulation d'une grammaire en mémoire ;
  - la transformation du graphe de départ de cette grammaire.
- Au niveau intermédiaire, on trouve l'API qui utilise l'AGG-Engine et expose l'ensemble des fonctions accessibles aux utilisateurs qui veulent utiliser l'outil AGG par programmation ;
- Au niveau supérieur, il y a l'AGG GUI qui exploite l'API en vue d'offrir une interface conviviale à l'utilisateur. Cet outil de visualisation et de manipulation facilite également l'utilisation et la compréhension des deux autres couches.

  l'AGG-Engine et l'API sont réutilisés dans le cadre de cette expérimentation. AGG est écrit en Java 5.0, langage adopté pour cette expérimentation.

Nous présentons dans la section suivante le modèle d'implémentation d'AGG sur SAEV.

6.5.2 Le modèle d'implémentation de SAEV

Le diagramme de classes que nous présentons ici complète le diagramme de classes du modèle SAEV (cf. section 3.3 du chapitre 3). Il permet ainsi de montrer les liens entre les classes du modèle SAEV et les plus importantes des classes réutilisées d'AGG.

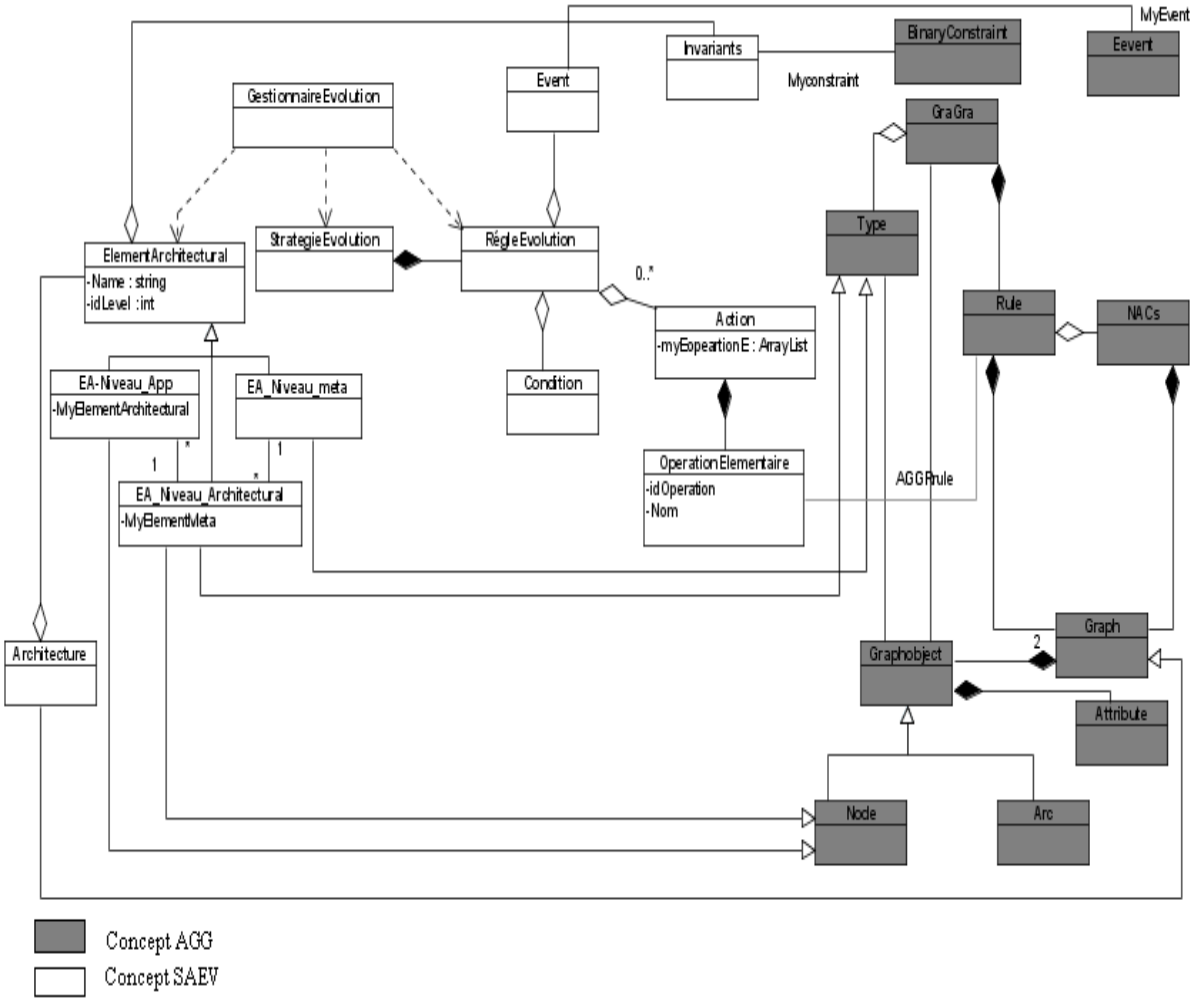

Figure 6.8 – Le diagramme de classes du modèle de SAEV sur AGG

Le diagramme de classes montre l'interaction entre les classes d'AGG et les classes de SAEV, suivant les projections que nous avons présenté dans le tableau 6.1.

Nous décrivons à titre illustratif le concept *Element Architectural*. Dans le méta modèle que nous avons présenté dans la section 3.3.1 du chapitre 3, nous n'avons pas illustré la distinction entre les éléments architecturaux des différents niveaux d'abstraction. Il est intéressant de conserver cette information sur le niveau d'abstraction d'un élément architectural et sur le niveau d'abstraction supérieur qui a permis sa spécification. Ainsi, il est plus facile d'identifier les éléments architecturaux du niveau inférieur correspondants à un élément architectural d'un niveau donné. Cette correspondance est nécessaire pour propager notamment les impacts de l'évolution d'un élément d'un niveau vers les éléments de son niveau inférieur.

Nous spécialisons ainsi, le concept Elément Architectural, en distinguant les trois types d'éléments architecturaux considérés par SAEV : Elément Architectural_Niveau_Meta, Element Architectural _Niveau_Architectural, Element Architectural _Niveau_Application. Ainsi :

- Element Architectural_Niveau_Meta sera projeté vers un *nœud* du *graphe de types*.
- Element Architectural_Niveau_Architectural sera projeté vers un *nœud* du *graphe* si l'évolution concerne le niveau architectural. Il sera projeté vers un *nœud* du *graphe types* si l'évolution concerne le niveau application.
- Element Architectural_Niveau_Application sera projeté vers un *nœud* du *graphe*.

Ceci est illustré dans le modèle d'implémentation par les liens d'héritage entre chacune de ces classes de SAEV et les classes correspondantes d'AGG.

6.6 Conclusion

Nous avons présenté dans ce chapitre une approche possible d'implémentation du modèle SAEV qui s'appuie sur un outil de transformation de graphes. En effet, nous avons souligné dans la section 6.2 que l'évolution architecturale telle que nous l'avons considéré dans le cadre de SAEV peut être assimilée à une transformation de graphes, en considérant la représentation d'une architecture logicielle sous forme d'un graphe. L'idée directrice est de réutiliser l'expérience gagnée dans le domaine de la transformation de graphes dans le cadre de l'évolution architecturale de SAEV. Nous nous sommes appuyés ainsi sur un outil de transformation de graphes comme support à l'implémentation de SAEV. L'avantage d'une telle approche est d'offrir d'un côté un moyen d'exprimer d'une façon formelle les concepts de SAEV, notamment les opérations d'évolutions et les invariants et d'un autre coté de s'appuyer sur l'implémentation de la transformation de graphe pour implémenter le modèle SAEV. Nous nous sommes appuyés dans cette expérimentation sur l'outil AGG (Attributed Graph Grammar). Un premier travail a consisté à étudier l'outil AGG ainsi que l'approche algébrique sur laquelle il repose. Nous avons établi ensuite une projection conceptuelle des concepts de SAEV vers ceux d'AGG. Nous avons montré aussi les fonctionnalités d'AGG qui peuvent être réutilisées dans le processus de SAEV. L'implémentation de la surcouche SAEV sur AGG se poursuit afin de finaliser et valider le prototype de SAEV.

Conclusion et perspectives

Bilan

Nous avons abordé dans cette thèse la problématique de l'évolution structurelle dans les architectures logicielles à base de composants. Nous avons proposé une approche de l'évolution par sa modélisation et sa spécification indépendamment de celle de l'architecture et sur différents niveaux d'abstraction.

L'évolution structurelle d'une architecture logicielle se reflète par les différents changements dans sa structure ou dans celles de ses éléments constitutifs. Ces changements sont engendrés par l'application des opérations d'évolution telles que l'ajout et la suppression sur ces éléments architecturaux.

L'évolution structurelle des architectures logicielles est peu abordée dans le domaine des architectures logicielles et par conséquent peu prise en compte par leurs langages de description (ADL). Comme nous l'avions préalablement souligné dans le chapitre 2, peu d'ADL proposent des supports pour l'évolution des architectures logicielles. La plupart de leurs propositions se limitent à des mécanismes tels que le sous-typage et la composition hiérarchique et elles n'abordent que l'évolution de certains éléments architecturaux tels que les composants types et les configurations. Dans ce qui suit nous analysons, nos contributions pour l'évolution structurelle des architectures logicielles à base de composants.

Notre première contribution est liée à l'état de l'art sur les approches d'évolution structurelle des architectures logicielles dans les ADL. Nous avons proposé une classification de l'évolution architecturale basée sur trois dimensions, afin de pouvoir étudier et comparer les approches d'évolution par rapport à un même référentiel :

- L'*objet de l'évolution* : l'évolution peut porter sur tout élément architectural réifié par l'ADL de l'architecture à faire évoluer,
- Le *type de l'évolution* : l'évolution peut être statique, si elle est réalisée durant la phase de spécification ou de conception de l'architecture logicielle. L'évolution est dite dynamique, si elle est réalisée durant l'exécution du système. Nous avons distingué l'évolution *dynamique anticipée* de l'évolution *dynamique non anticipée*. L'évolution est dite anticipée lorsque les besoins d'évolution sont identifiés et cernés dès le début. Nous parlons d'évolution dynamique non anticipée lorsque l'évolution est assurée pour des besoins qui apparaissent de façon imprévue durant le cycle de vie du système .
- Le *support de l'évolution* : décrit le moyen proposé pour *exprimer* l'évolution. Le support de l'évolution peut être implicite, prédéfini ou explicite. Le support est considéré comme implicite si aucune opération d'évolution ni mécanisme n'est proposé pour exprimer l'évolution. Il est dit prédéfini, si un ensemble d'opérations ou de mécanismes prédéfini est proposé pour exprimer l'évolution. Enfin, le support est dit explicite s'il offre le moyen d'exprimer toute évolution sur l'architecture. Dans le cas des supports d'évolution explicite et prédéfini, l'évolution exprimée peut être avec *rupture* ou *continue*. Elle est dite avec rupture si le lien entre l'architecture de

départ et l'architecture d'arrivée (ayant évolué) ne peut être établie. Dans le cas contraire on parle d'évolution continue.

Nous avons mis en évidence que toute approche d'évolution ou toute problématique peut être positionnée dans cette représentation triptyque en répondant pour chacune d'entre elles aux questions suivantes :

- Quel est l'objet de l'évolution : composant type, connecteur type configuration, interface, etc ?
- Quel type d'évolution propose t-elle : statique ou dynamique ?
- Quel type de support propose t-elle pour exprimer l'évolution : implicite, prédéfini ou explicite ?

Cette classification nous a permis de situer les aspects de l'évolution architecturale qui ont été largement traités, peu traités ou jamais traités.

Notre deuxième contribution consiste en la proposition d'un modèle d'évolution d'architecture logicielle baptisé SAEV (Software Architecture EVolution model). Nous avons mis en avant dans le modèle SAEV la nécessité d'abstraire l'évolution et de la considérer explicitement et indépendamment du comportement des éléments architecturaux et de leurs langages de description. Le modèle SAEV propose ainsi des concepts et des mécanismes pour permettre la *spécification* et la *gestion* de l'évolution de tout élément architectural d'une manière uniforme et sur les différents niveaux d'abstraction de description d'une architecture logicielle : le niveau Architectural et le niveau Application. SAEV se base principalement sur les concepts de *stratégie d'évolution* et d'*invariant*. Une stratégie d'évolution regroupe un ensemble de *règles d'évolution*. Une règle d'évolution décrit l'application d'une opération d'évolution sur élément architectural, en spécifiant les conditions nécessaires pour le faire, ainsi que les éventuels impacts qu'elle peut engendrer sur les autres éléments architecturaux. Ces règles d'évolution sont décrites à l'aide du formalisme ECA (Event/Condition/Action) [22] pour permettre à une architecture d'être réactive. Les invariants expriment les contraintes à respecter au cours de l'évolution pour guider l'évolution en sauvegardant la cohérence globale de l'architecture logicielle. SAEV propose aussi un *gestionnaire d'évolution* permettant de gérer l'évolution d'une architecture en fonction des événements reçus et des différentes règles d'évolutions spécifiées.

Nous avons souligné au travers de SAEV l'importance de considérer la propagation des impacts d'une évolution. En effet, l'évolution d'un élément architectural peut avoir des répercussions sur les autres éléments de l'architecture. Ces répercussions doivent être propagées pour sauvegarder la cohérence globale de l'architecture. Pour permettre la détermination et la propagation automatique, autant que possible, des impacts d'une évolution, SAEV doit disposer de plus d'informations sur le degré de corrélation entre les éléments architecturaux. Notre troisième contribution consiste ainsi à définir un ensemble de propriétés sémantiques qui expriment le degré de corrélation entre les éléments architecturaux : l'Exclusivité/Partage, la Dépendance/Indépendance, la Prédominance/Non prédominance et la Cardinalité/Cardinalité inverse. Ces propriétés sont associées au concept de Connecteur. Nous considérons en effet, que les connecteurs par leurs positions d'intermédiaires entre les éléments architecturaux représentent des supports idéaux pour ces propriétés sémantiques. Les propriétés sémantiques sont ainsi associées au concept connecteur, renseignées à la spécification d'une architecture et s'appliquent à l'évolution de toute application construite à partir de cette architecture.

Notre quatrième contribution est pour l'ADL COSA. COSA est un ADL hybride proposé au sein de notre équipe pour la description structurelle des architectures logicielles, en combinant les concepts de l'approche objet et ceux de l'approche à base de composants. Nous avons proposé d'enrichir le concept

du *Connecteur* de COSA par les propriétés sémantiques proposées tout en considérant les spécificités particulières des connecteurs de COSA. Nous avons illustré ensuite l'utilisation du modèle SAEV pour l'évolution d'architectures logicielles décrites en COSA au niveau architectural et au niveau application. Le modèle SAEV couplé ainsi à COSA, permet d'aboutir à une approche pour la spécification et l'évolution structurelle des architectures logicielles.

Perspectives

Le travail de cette thèse a permis la proposition d'un modèle pour l'évolution structurelle des architectures logicielles à base de composants. Nous avons capitalisé les concepts nécessaires pour la spécification et la gestion de l'évolution des architectures logicielles indépendamment de leurs spécifications proprement dites. Nous avons souligné l'importance d'enrichir les descriptions architecturales par plus d'informations sur le degré de corrélation entre ces éléments pour améliorer le processus d'évolution et de propagation des impacts d'une évolution. Cette thèse peut se prolonger en plusieurs perspectives de recherche à moyen et à long terme qui peuvent se décliner en six points :

Prise en compte des opérations d'évolution plus complexes

Nous nous sommes appuyés pour la description du modèle SAEV sur des opérations d'évolution minimales et basiques. D'autres opérations plus complexes peuvent être construites sur la base de ces opérations basiques, tels que la *fusion*, le *transfert*, le *versionnement* etc. Il s'agit ensuite de définir les règles d'évolution associées à ces opérations d'évolution et pour chaque élément architectural .

Spécification des propriétés sémantiques dans le cas des connecteurs non réifiés

Nous avons proposé, dans le chapitre 4, des propriétés sémantiques exprimant le degré de corrélation entre les éléments architecturaux que nous avons associé au concept de connecteur. Ces propriétés sont ainsi exploitées et utilisées uniquement dans le cadre des ADL réifiant le concept du connecteur. L'objectif est alors d'offrir le moyen d'exprimer ces propriétés sémantiques entre les éléments architecturaux même si le concept du connecteur n'est pas réifié. L'idée est soit de proposer un nouveau concept au niveau de SAEV qui permettrait de spécifier la corrélation entre les éléments architecturaux soit de les exprimer sous forme de règles d'évolution ou d'invariants particuliers.

Analyse et réutilisation des règles d'évolution

Une des améliorations possibles au niveau des règles d'évolution est de permettre l'analyse des règles d'évolution pour détecter des éventuelles incompatibilités et interblocage entre les règles d'évolution et pour offrir la possibilité d'activer et désactiver les règles d'évolution avant de lancer l'exécution de l'évolution. La spécification formelle des règles d'évolution peut nous aider à apporter cette solution.

Pour faciliter la spécification des règles d'évolution, le modèle d'évolution doit permettre au concepteur de réutiliser des spécifications existantes de règles d'évolution. En effet, il est possible de définir un mécanisme de *surdéfinition*, qui permettrait de créer des règles d'évolution en *surchargeant* soit l'événement, la partie condition ou la partie action d'une règle.

Prise en compte de l'aspect comportement des architectures logicielles

Nous avons noté dans le chapitre 1 que la spécification d'une architecture logicielle peut se focaliser sur l'aspect structurel ou sur l'aspect comportemental. Nous avons considéré uniquement l'aspect struc-

turel dans nos propositions. Nous considérons que l'intégration de l'aspect comportemental dans notre travail actuel ne sera que complémentaire pour le modèle d'évolution que nous proposons.

L'évolution dynamique non anticipée des architectures logicielles

Cette préoccupation constitue l'un des défis actuels de la communauté des architectures logicielles. Pour permettre l'évolution dynamique non anticipée des architectures logicielles, l'approche d'évolution que nous proposons doit être capable de refléter les différentes évolutions au niveau d'une architecture logicielle directement sur le système en cours d'exécution. Une des pistes à exploiter est de définir des passerelles entre les concepts de description architecturales et les concepts d'implémentation de ces architectures logicielles, se servir ensuite de cette passerelles pour définir les correspondances des évolutions architecturales au niveau du système.

Implémentation et Validation

Nous avons montré dans le chapitre 6 une projection des concepts de SAEV vers un outil de transformation de graphe AGG. Ce dernier ne gère que deux niveaux d'abstraction à la fois. Notre réflexion doit se poursuivre sur le moyen de manipuler les trois niveaux d'abstraction en même temps pour pouvoir gérer les impacts d'évolution entre ces différents niveaux d'abstraction.

Bibliographie

[1] Project ACCORD.
Etat de l'art sur les langages de description d'architecture (adls).
Dans *Rapport technique, INRIA*, France, 2002.

[2] R. ALLEN.
A formal Approach to Software Architecture Description.
Thèse de Doctorat, Carnegie Mellon University, Technical Report Number : CMU-CS-97-144, 1997.

[3] R. ALLEN, R. DOUENCE et D.GRALAN.
Specifying and analyzing dynamic software architectures.
Dans *Proceeding of 1998 conference on Fondamental Approches to Software Engineering*, Lisbon, Portugal, 1998.

[4] R. ALLEN et D. GARLAN.
The wright architectural specification language.
Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, 1996.

[5] R. ALLEN et D. GARLAN.
A formal basis for architectural connection.
ACM Transactions on Software Engineering and Methodology, 6(3):213–249, 1997.

[6] A. APRIL, J.M. DESHARNAIS et R. DUMKE.
A formalism of ontology to support a software maintenance knowledge-based system.
Dans *the Eighteenth International Conference on Software Engineering Knowledge Engineering Conference (SEKE06)*, pages 331–336, 2006.

[7] O. BARAIS.
Construire et Maîtriser l'Evolution d'une Architecture Logicielle à base de Composants.
Thèse de Doctorat, Université des Sciences et Technologies de Lille, 2005.

[8] O. BARAIS et L. DUCHIEN.
Safarchie : Maîtriser l'Évolution d'une architecture logicielle.
Langages, Modèles et Objets - Journées Composants - LMO 2004-JC 2004, L'objet, 10(2-3):103–116, 2004.

[9] F. BARBIER, C. CAUVET, M. OUSSALAH, D. RIEU et C. SOUVEYET.
Concepts clés et techniques de réutilisation dans l'ingénierie des systèmes d'information.
Dans *Ingénierie des composants dans les systèmes d'information M. Oussalah et D. Rieu (réda.)*. Hermes Science Publications, 2004.

[10] L. BASS, P. CLEMENTS et R. KAZMAN.
Software Architecture In Practice.
Addison-Wesley Publishing, 1998.

[11] T. BOLUSSET.
B-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B.
Thèse de Doctorat, Université de Savoie, 2004.

[12] T. BRAY, J. PAOLI, C.M. SPERBERG-MCQUEEN et E. MALER.
Extensible markup language (xml) 1.0 (fourth edition) w3c recommendation.
http://www.w3.org/TR/REC-xml/.

[13] J. BUFFENBARGER.
Systactic software merging.
Dans *software configuration management ICSE SCM-4 and SCM-5*, pages 153–172. spring-verlag, 1995.

[14] C. BURROWS et I. WESLEY.
Ovum evaluates: Configuration management.
Ovum, London, 1998.

[15] L. CARDELLI et P. WEGNER.
On understanding types and data abstraction and polymorphism.
ACM Computing Surveys, 17(4):471–521, 1985.

[16] S. CLARKE.
Composition of Object-Oriented Software Design Models.
Thèse de Doctorat, Université de Dublin, School of computer application, 2001.

[17] COLABNET.
Subversion.
http://www.subversion.tigris.org.

[18] A. CORRADINI, U. MONTANARI, F. ROSSI, H. EHRIG, R. HECKEL et L. LOWE.
Algebraic approaches to graph transformation. part i: Basic concepts and double pushout approach.
Dans *Handbook of Graph Grammars*, pages 163–245, 1997.

[19] T. COUPAYE, E. BRUNETON et J. STEFANI.
The fractal composition framework, proposed final draft of interface specification version 0.9.
The ObjectWeb Consortium, 2002.

[20] E.M. DASHOFY et A. van der HOEK.
Representing product family architectures in an extensible architecture description language.
Dans *Proceedings of the International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, 2001.

[21] E.M. DASHOFY, A. van der HOEK et R.N. TAYLOR.
A highly-extensible, xml-based architecture description language.
Dans *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[22] U. DAYAL, A. BUCHMMAN et D. MCCARTHY.
Rules are objects too: a knowledge model for an active object-oriented database systems.
Lecture Notes in Computer Science 334, pages 129–143, 1988.

[23] F. DEREMER et H. KRON.
Programming in the large vs programming.
IEEE Transactions on Software Engineering, 2(2):321–327, 1976.

[24] A. DIMOV et S. ILIEVA.
System level modeling of component based software systems.
Dans *International Conference on Computer Systems and Technologies - CompSysTech2004*.

[25] F. DUCLOS, J. ESTUBLIER et R. SANLAVILLE.
Architectures ouvertes pour l'adaptation des logiciels.
Génie Logiciel, (58), 2001.

[26] H. EHRIG, R. HECKEL, M. KORFF, M. LOWE, L. RIBEIRO et A. CORRADINI.
Algebraic approaches to graph transformation. part ii: Single pushout approach and comparison with double pushout approach.
Dans *Handbook of Graph Grammars*, pages 247–312, 1997.

[27] H. EHRIG, M. PFENDER et H. J. SCHNEINDER.
Graph grammars: an algebraic approach.
Dans *Proceedings of IEEE conerence on automata and switching Theory*, pages 167–180, 1973.

[28] Eclipse FONDATIONS.
http://www.eclipse.org.

[29] Institute for SOFTWARE RESEARCH.
Archstudio.
http://www.isr.uci.edu/projects/archstudio, 2005.

[30] D. GARLAN.
Software architecture and object-oriented systems.
Dans *Proceedings of the IPSJ Object-Oriented Symposium*, 2000.

[31] D. GARLAN, R. MONROE et D.WILE.
Acme : An architecture description interchange language.
Dans *Proceedings of CASCON'97*, Toronto, Canada, 1997.

[32] D. GARLAN, R. MONROE et D.WILE.
Acme : An architecture description of component-bases systems.
Fondations of Component-Based Systems, Leavens Gray et Sitaraman Murali (Réd.), pages 47–68, 2000.

[33] D. GARLAN et D. PERRY.
Introduction to the special issue on software architecture.
IEEE Transactions on software Engineering, 21(4), 1995.

[34] IEEE Architecture Working GROUP.
Ieee recommended practice for architectural description of software-intensive systems.
Institute of Electritical and Electronics Engineers, 2000.

[35] Object Management GROUP.
Uml 2.0 infrastructure specification, technical report ptc/03-09-15.
2003.

[36] G. HEINEMAN et W. COUNCILL.
Component-based software engineering- putting the pieces together.
Massachusetts, 2001. Addison-Wesley Publishing, Reading.

[37] C. HOARE.
Communicating sequential processes.
1995.

[38] A.V.D HOEK, M. RAKIC, R.ROSHANDEL et N. MEDVIDOVIC.
Taming architectural evolution.
Dans *Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9*, pages 1–10, Vienna, Austria, 2005.

[39] Software Engineering INSTITUTE.
How do you define software architecture?
http://www.sei.cmu.educ/architecture/definitions.html.

[40] ISR.
xarch.
http://www.isr.uci.edu/architecture/xarch.

[41] M. JACKMAN et C. PAVELIN.
Conceptual graphs.
Approaches to Knowledge Representation, Ringlind G.A and Duce D. A (réd.), 1989.

[42] T. KHAMMACI, A. SMEDA et M. OUSSALAH.
Coexistence of software architecture and object oriented modeling.
volume 3. World Scientific Publications Co.

[43] R. KHARE, M. GUNTERSDORFER, P. OREIZY, N. MEDVIDOVIC et R. N. TAYLOR.
xadl: Enabling architecture-centric tool integration with xml.
Dans *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, 2001.

[44] G. KICZALES.
Aspect-oriented programming.
ACM Computing Surveys (CSUR), 28(4), 1996.

[45] W. KIM, E. BERTINO et J.F. GARZA.
Composite objects revisited.
International Conference on Management of Data, pages 337–347, 1989.

[46] J. LOWY.
Programming .net components 1st edition.
O'Reilly,USA, 2003.

[47] D. LUCKHAM, M. AUGUSTIN, J. KENNY, J. VERA, D. BRYAN et W. MANN.
Specification and analysis of system architectures using rapide.
IEEE Transaction on Software Engineering, 21(4).

[48] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER.
Specification and analysis of system architectures using rapide.
Dans *Proceedings of the fifth European Software Engineering conference*, pages 336–355, Barcelona, Spain, 1995.

[49] J. MAGEE et J. KRAMER.
Dynamic structure in software architecture.
Dans *Proceedings of ACM SIGSOFT'96: Fourth Symposium Foundation of Software engineering*, pages 3–14, 1996.

[50] M. MAGNAN.
Réutilisation de composants: Les Exceptions dans les objets Composites.
Thèse de Doctorat, Université de Montpellier II, 1994.

[51] S. MAILLARD, A. SMEDA et M. OUSSALAH.
Cosa: An architectural description meta-model.
Dans *Proceeding of International Conference on Software and Data Technologies (ICSOFT 2007)*, volume 3. INSTICC Press, 2007.

[52] N. MEDVIDOVIC, D.S. ROSENBLUM et R.N. TAYLOR.
Type theory for software architectures.
Dans *Technical Report, UCI-ICS-98-14, University of California, Irvine*, 1998.

[53] N. MEDVIDOVIC, D.S. ROSENBLUM et R.N. TAYLOR.

A language and environment for architecture-based software development and evolution.
Dans *Proceeding of the 21st international conference on Software engineering, (ICSE'99)*, pages 44–53, 1999.

[54] N. MEDVIDOVIC, R. TAYLOR et E. WHITEHEAD.
Formal modeling of software architectures at multiple levels of abstraction.
Dans *Proceedings of the 1996 California Software Symposium*, pages 28–40, Los Angeles, CA, 1996.

[55] N. MEDVIDOVIC et R.N. TAYLOR.
A classification and comparison framework for software architecture description languages.
IEEE Transactions on Software Engineering, 26(1):70–93, 2000.

[56] N. MEHTA, N. MEDVIDOVIC et S. PHADKE.
Towards a taxonomy of software connectors.
Dans *the 22nd International Conference of Software Engineering (ICSE'00)*, pages 178 – 187, Limerick, Ireland, 2000.

[57] T. MENS, J. BUCKLEY, M. ZENGER et A. RASHID.
Towards a software evolution taxonomy.
Dans *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, 2003.

[58] B. MEYER.
Applying design by contract.
Computer, 25(10):40–51, 1992.

[59] E.C. NISTOR, J.R. ERENKRANTZ, S.A. HENDRICKSON et A.V. HOEK.
Archevol: versioning architectural-implementation relationships.
Dans *Proceedings of the 12th international workshop on Software configuration management*, pages 99 – 111, Lisbon, Portugal, 2005.

[60] P. OREIZY.
Decentralized software evolution.
Dans *International Work on the principles of Software Evolution (IWPSE1)*, Kyoto, Japan, 1998.

[61] A. ORSO, E.J. HARROLD et D.S. ROSENBLUM.
Component metadata for software engineering tasks.
Dans *Second International Workshop on Engineering Distributed Objects (EDO 200),Springer Verlag*, pages 126–140, Berlin, 2000.

[62] M. OUSSALAH, N. SADOU et D. TAMAZALIT.
Saev : a model to face evolution problem in software architecture.
Dans *In proceedings of the International ERCIM Workshop on Software Evolution*, pages 137–146, Lille, France, 2006.

[63] M. OUSSALAH, A. SMEDA et T. KHAMMACI.
An explicit definition of connectors for component-based software architecture.
Dans *Proceedings of IEEE International Conference on the Engineering of Computer Based Sysrems (ECBS 2004), Brno, Czech Republic*, 2004.

[64] M. OUSSALAH et *al.*.
Gènie objet, Analyse et conception de lévolution.
Hermes édition, 1999.

[65] M. OUSSALAH et *al.*.

Ingénierie des composants : Concepts, Techniques et Outils.
Vuibert, Paris, 2005.

[66] D.E. PERRY et A.L. WOLF.
Foundations for the study of software architecture.
Software Engineering Notes, ACM SIGSOFT, 17.

[67] T. PIGOSKI et A. APRIL.
Software maintenance.
Ironman version of the Guide to the Software Engineering Body of Knowledge, pages 1–15, 2005.

[68] R. ROSHANDEL, A. V.D.HOEK, M. MIKI-RAKIC et N. MEDVIDOVIC.
Mae-a system model and environment for managing architectural evolution.
ACM Transactions on Software Engineering and Methodology, 13(2).

[69] N. SADOU, D. TAMAZALIT et M. OUSSALAH.
Saev: une solution à l'évolution structurelle des architectures logicielles.
Revue de l'OBJET, numéro spécial intitulée "Evolution du logiciel", 13(1/2007):45–80.

[70] N. SADOU, D. TAMAZALIT et M. OUSSALAH.
How to manage uniformly software architecture at different abstraction levels.
Dans *proceedings of the 24th ACM International conference on Conceptual Modeling (ER2005)*, pages 16–30, Klagenfurt, Austria, 2005.

[71] N. SADOU, D. TAMAZALIT et M. OUSSALAH.
A unified approach for software architecture evolution at different abstraction levels.
Dans *International Workshop on Principles of Software Evolution September (IWPSE 2005) in association with 8th European Software Engineering Conference and 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 65–70, Lisbon, Portugal, 2005.

[72] N. SADOU, D. TAMAZALIT et M. OUSSALAH.
Component based software architectures: their evolution through the connectors.
GESTS International Transactions on Computer Science and Engineering, 39(1), 2007.

[73] R. SANLAVILLE.
Architecture Logicielle : Une Expérimentation Industrielle avec Dassault Systèmes.
Thèse de Doctorat, Université Joseph Fourier-Gronoble I, Grenoble, 2002.

[74] X. SEIGNARD.
Proposition et développement d'un gestionnaire d'évolution architecturale basé sur la transformation de graphes.
2007.

[75] M. SHAW, R. DELINE, D.V. KLEIN, T.L. ROSS, D.M. YOUNG et G. ZELESNIK.
Abstractions for software architecture and tools to support them, 1995.

[76] A. SMEDA.
Contribution à l'élaboration d'une métamodélisation de description d'architecture logicielle.
Thèse de Doctorat, Université de Nantes, 2006.

[77] A. SMEDA, T. KHAMMACI et M. OUSSALAH.
Operational mechanisms for components-object based software.
Dans *Proceedings of IEEE International conference on Information communication Technologies: From Theory to Applications (ICTTA 2004)*, Damas, Syria, 2004.

[78] A. SMEDA, M. OUSSALAH et T. KHAMMACI.

Software connectors reuse in component-based systems.
Dans *Proceedings of IEEE International Conference on information Ruse and Integration (IRI 2003)*, 2003.

[79] A. SMEDA, M. OUSSALAH et T. KHAMMACI.
A multi-paradigm approach to describe complex software system.
WSEAS Transactions on Computers, 3(4):936–941, 2004.

[80] A. SMEDA, M. OUSSALAH et T. KHAMMACI.
Mapping adls into uml 2.0 using a meta adl.
Dans *Proceeding of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05),IEEE CS. Press*, Pittsburg, Pennsylvania, 2005.

[81] J.M. SPIVEY.
the z notations: A reference manual.

[82] C. SZYPERSKI.
Component software: Beyond object-oriented programming.
Addison Wesley Professional.

[83] G. TAENTZER.
Agg : A tool environment for algebric graph transformation.
Dans *Proceedongs of the International Workshop on Applications of Graph Transformation with industrial Relevance (AGtive'99)*, pages 481–488, 1999.

[84] D. TAMAZALIT, N. SADOU, et M. OUSSALAH.
Connectors conveying software architecture evolution.
31st International Computer Software and Applications Conference COMPSAC, pages 391–396.

[85] D. TAMZALIT.
GENOME: un Modèle pour la Simulation d'Emergence de Structures d'Objets.
Thèse de Doctorat, Université de Nantes, 2000.

[86] AGG TEAM.
The agg 1.5.0 development environment.
Dans *The User Manual, 2006*.

[87] T. VILLA, T. KAM, R. BRAYTON et A. SAGIOVANNI-VINCENTELLI.
Synthesis of finite machines : Logic optimization.
TKluwer Academic Publishers, Philip.

[88] C.V. D WESTHUIZEN et A.V.D HOEK.
Understanding and propagating architectural changes.
Dans *Proceedings of the Working IFIP Conference on Software Architecture*, pages 95–109, 2002.

[89] D.S. WILE.
Using dynamic acme.
Dans *Proceedings of a working Conference on Complex and Dynamic Systems Architecture*, 1997.

Liste des tableaux

— *Corps du document* —

— *Annexes* —

Liste des figures

— Corps du document —

— *Annexes* —

Liste des exemples

Table des matières

— Corps du document —

— *Pages annexées* —

— *Annexes* —

Annexes

Annexe A

Exemples d'évolution architecturale

Nous présentons dans cette annexe des exemples de spécifications d'évolution statiques ou dynamiques ainsi que le détail des outils introduit dans le chapitre 2.

Exemple d'évolution dynamique dans Wright

Nous présentons un exemple d'une spécification de la propriété de tolérance aux fautes dans une architecture Client/Serveur [3]. Considérons un système composé de deux serveurs. Un serveur *primaire* qu'il est préférable d'utiliser et un serveur *secondaire* qui sert de serveur de secours, dans le cas où le premier tombe en panne.

Style Fault-Tolerant-Client-Server
  Component Client
    Port p = ($\overline{request}\rightarrow$ reply $\rightarrow$ p $\sqcap$ §
    Computation = [. . .]
  Component FlakyServer
    Computation = [. . .]
    Port p = § [] (request $\rightarrow \overline{reply}\rightarrow$ p $\sqcap$ $control.down\rightarrow$ (§ [] control.up $\rightarrow$ p))
    Computation = [. . .]
  Component SlowServer
    Port p = § [] (control.on $\rightarrow \mu$Loop.(request $\rightarrow \overline{reply}\rightarrow$ Loop [] control.off $\rightarrow$ p $\sqcap$ §))
    Computation = [. . .]
  Connector FTLink
    Role c = [. . .]
    Role s = [. . .]
    Glue = [. . .]
End Style

Table 2 – Spécification du style tolérance aux fautes

La situation à spécifier est la suivante : dès que le serveur *primaire* tombe en panne, la connexion est interrompue. Elle est ensuite rétablie vers le serveur *secondaire* jusqu'à la restauration du serveur *primaire* . Cette situation est décrite à l'aide du configurateur suivant :

Configuror DynamicClient-Server
Style Fault-Tolerant-Client-Server
new.C :Client
→ new.Primary : FlakyServer
→ new.Secondary : SlowServer
→ new.L : FTLink
→ attach.C.p.to.L.c
→attach.Primary.p.to.L.s WaitForDown
where
WaitForDown = (Primary.**control**.down → Secondary.**control**.on → L.control.changeOk →
Style Fault-Tolerant-Client-Server
→ detach.Primary.p.from.L.s
→ attach.Secondary.p.to.L.s
→ WaitForUp)
[] §
WaitForUp = (Primary.**control**.up → Secondary.**control**.off → L.**control**.changeOk →
Style Fault-Tolerant-Client-Server
detach.Secondary.p.from.L.s
→ attach.Primary.p.to.L.s
→ WaitForDown)
[] §

Table 3 – Spécification du configurateur tolérance aux fautes

Nous remarquons d'abord que dans la spécification de l'architecture des événements de contrôles sont ajoutés précédés de " *control* ", tel que *down* et *changeOK*. Dans le configurateur les instances de composants et de connecteurs intervenants dans le configuration, ainsi que les différents attachements initiaux sont d'abords déclarés. Le configurateur se met en attente d'un événement de contrôle exprimé par *WaitForDown*. Dans le cas où le serveur primaire tomberait en panne, exprimé par l'événement de contrôle *Primary.control.down*, la reconfiguration est opérée. Le serveur primaire est alors détaché, et c'est le serveur secondaire qui sera rattaché à sa place : *detach.Primary.p.from.L.s* et *attach.Secondary.p.to.L.s*. Au rétablissement du serveur primaire *Primary.control.up* la reconfiguration inverse est opérée.

Évolution dynamique dans DARWIN

L'exemple suivant montre l'instanciation *paresseuse* du composant *server*.

Component ClientServer {
inst client : Client ;
inst server : dyn Server ;
bind client.request – server.reply
}

Table 4 – Instanciation paresseuse dans Darwin

L'instance *client* est crée à l'instanciation du composant composite *ClientServeur*. L'instance *server* quant à elle est déclarée dynamique et ne sera créée que lorsque le *client* la sollicitera. L'instanciation

paresseuse permet donc de décrire la configuration potentielle à l'exécution, en déclarant des instances potentiellement nécessaires.

L'évolution architecturale dans C2SADEL

Evolution dynamique

Étant donné l'architecture Client/Serveur suivante :

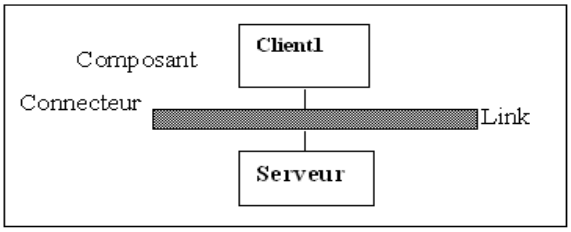

Figure 9 – Une architecture Client/serveur en C2SADEL

ArchStudio1 [60] offre une interface interactive pour l'écriture des spécifications d'évolution dynamiques. Pour ajouter par exemple le composant *Client2* à l'architecture précédente, la spécification sera écrite comme suit :

```
ClientServer.addComponent (Client2) ;
ClientServer.weld( Link, Client2) ;
```

La première ligne de la spécification décrit l'ajout du composant à l'architecture *ClientServer*, et la deuxième ligne indique la connexion de ce composant au connecteur *Link*.

Pour la suppression d'un composant d'une architecture, il faut procéder inversement. Déconnecter d'abord le composant du connecteur, puis déclencher sa suppression. La spécification suivante décrit la suppression du composant *Client1* attaché au connecteur *Link* :

```
ClientServer.unweld(Link,Client1) ;
ClientServer.removeCompoment (Client1) ;
```

Dans la littérature, ni l'origine de la création ou de la suppression des ces composants, ni les moyens de gestion de ces composants, une fois crées ne sont spécifiés.

Évolution statique dans ACME

Nous illustrons un exemple de la création d'un composant à partir d'un autre en utilisant le mécanisme du sous-typage.

Le composant type *UnixFilter* est défini comme une extension du composant type *FilterT*. Le composant *UnixFilter* définit un port *stderr* en plus de ce qui est défini par le composant *FilterT*.

```
Family PipFilterFam = {
  Component Type FilterT = {
      Ports { stdin ; stdout } ;
      Property { int } ;
   } ;
   Component Type UnixFilterT extends FilterT with
      Ports stderr ;
      Property implementationFile : String ;
   Connector = {...}
   Attachment = {...}
}
```

Principe de fonctionnement de TranSaT

Cette section présente le principe de l'évolution dans TranSaT. TransSAT (un modèle d'évolution d'architecture logicielle) a été défini pour étendre le modèle de type SafArchie, et permettre la construction incrémentale d'une architecture logicielle par ajout de nouvelles préoccupations. Inspirée des travaux autour de la séparation des préoccupations et des aspects [44, 16], ce canevas suggère d'établir d'abord la spécification de l'architecture du système autour de ses fonctionnalités métiers, et ensuite, d'établir la spécification des fonctionnalités techniques indépendamment, pour ensuite intégrer et regrouper les deux spécifications. Le canevas de conception TranSAT est construit comme un système de tissage d'aspects au niveau d'une description architecturale. Le canevas est centré autour des concepts du patron architectural et du tisseur. Le patron architecturale contient l'ensemble des informations relatives à une préoccupation : le plan, masque de jonction, règles de transformations. L'architecture globale de TranSAT est donnée dans la figure suivante :

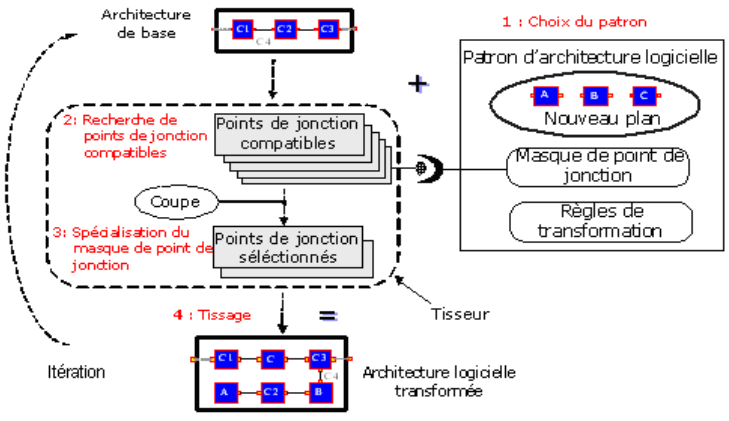

Figure 10 – Les trois dimensions de l'évolution architecturale

1. Patron architectural ;

   - Le plan : représente un assemblage de composants mettant en œuvre les services liés à une préoccupation ;

- Le masque de jonction : décrit des contraintes sur le lieu sur lequel le nouveau plan peut être intégré. Il est utilisé aussi pour décrire les modifications à apporter sur l'architecture pour intégrer une nouvelle fonctionnalité.
- Les règles de transformations : spécifient les modifications à apporter au niveau de l'architecture de base et sur le nouveau patron pour permettre le tissage. Ces règles sont décrites par un ensemble de primitives. Ces primitives sont scindées en deux, les primitives d'introduction et les primitives de reconfiguration.

2. Le tisseur : il est associé à un couple (patron d'architecture, plan de base). Il permet d'intégrer un nouveau plan d'architecture sur un plan de base. Il est d'un point de vue abstrait, un opérateur de transformation qui prend en entrée trois types d'informations : l'architecture logicielle à transformer, une spécification des lieux d'intégration sur cette architecture et le patron d'architecture.

Évolution statique dans Mae

Cette section présente la syntaxe utilisée dans Mae pour la spécification des composants types, obtenue de la combinaison des concepts architecturaux et des concepts de la *gestion des configurations.*

Syntaxe d'un **composant type**	Exemple
name	Name = *SIMULATIONAGENT*
revision	Revision=3
interface [,garde]*	Interface = *IGETSOURCES, IGETRULS, ISIMULATEBATTLE*
component [, guard]*	Component= *AGENT, KNOWLDGEBASE*
interfaceMapping *	InterfaceMapping= ...
behaviour*	behaviour=...
constraint*	constraints=...
ascendant*	ascendant=*SIMULATIONAGENT 2*
descendant*	descendant=*SIMULATEANDFIGHT 1*
subtype*	subtype=*beh* and *int, SIMULATIONAGENT 2*

L'exemple illustre la syntaxe de Mae pour la spécification d'un composant type. Le champ *revision* est utilisé pour indiquer le numéro de version du composant type. Le champ *ascendant* indique le composant type à partir duquel est crée ce composant type, en précisant via le champ *subtype* ce qu'il préserve de son prédécesseur (comportement, interface, etc.). Le champ *descendant* précise les composants types descendants de ce composant type.

Principe de fonctionnement de l'outil ArchEvol

Le but principal de ArchEvol [59] est de maintenir la cohérence entre la description architecturale d'un système et son implémentation (le code source) au fur et à mesure de leurs évolutions. ArchEvol propose un environnement conçu à partir de trois outils existants : Archstudio 3 [29], *Eclipse* (environnement de développement Java) [28] et Subversion [17] un système de gestion de configuration. Pour permettre à *Archstudio3* et *Eclipse* de coopérer, ArchEvol leur a défini des extensions, ainsi qu'une interface intermédiaire de communication. Ensuite, un mapping est défini entre les éléments gérés par *ArchStudio 3* et ceux d'Eclipse. Ainsi, pour chaque composant et chaque connecteur définis dans *ArchStudio 3*, est associé un component project dans *Eclipse*. Ce projet, en plus des packages et des classes

java, il contient un fichier de description des propriétés des composants et des connecteurs décrites sous formes de Méta données [61]. Les méta données doivent rester toujours en cohérence avec l'implémentation ainsi qu'avec la description architecturale.Le troisième outil Subversion réutilisé dans ArchEvol fournie un *WebDAV* dépôt permettant de stocker tous les projets et leurs contenus au fur et à mesure de leur évolution. Chaque composant/connecteur garde un pointeur vers son propre répertoire dans le dépôt de *Subversion*.

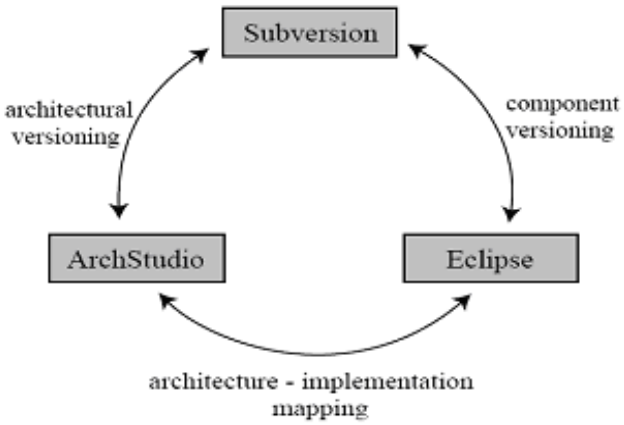

Figure 11 – Structure d'ArchEvol

Le processus d'utilisation d'ArchEvol est défini comme suit :

- Une copie de l'architecture à modifier est isolée en utilisant l'outil Subversion.
- L'architecte ouvre et modifie cette architecture avec ArchStudio 3.
- Via l'extension d'Eclipse, l'architecte récupère la liste des composants et connecteurs avec leurs respective URLs du code source. Ces dernières sont utilisées pour charger le code source comme un projet sous Eclipse ;
- La partie implémentation concernée est isolée et servira à la création d'un nouveau projet en se basant sur les méta données récupérées via Archstudio 3 ;
- Le développeur modifie le code source du composant/connecteur. Ces modifications vont servir ensuite pour mettre à jour les méta données de la description architecturale ;
- Une fois que la modification est validée sous Eclipse, l'architecture modifiée via ArchStudio 3 sera intégrée comme une nouvelle version à l'aide de Subversion.

Principe de l'algorithme de détermination de différence basé sur xADL2.0

```
elementListOne ← All elements in architecture one
elementListTwo ← All elements in architecture two
diff ← empty

/*
Inspect all elements in the first structure and compare them
to elements in the second structure to check for additions.
*/
for each element1 in elementListOne
{
   if an element in elementListTwo has an ID matching
      element1.getID()
   {
      element2 ← element with matching ID in elementListTwo
      if (!element1.isEquivalent(element2))
      {
         /*
         element1 and element2 have same ID but not
         identical internally so are different.
         */
         ...output warning & terminate...
      }
   }
   else
   {
      diff.addAdd(element1)
   }
}

/*
Check second list for removals.
*/
for each element2 in elementListTwo
{
   if no element in elementListOne has an ID matching
      element2.getID()
   {
      /*
      element2 is an obsolete element.
      */
      diff.addRemove(element2)
   }
}
```

Figure 12 – Algorithme de différence

L'algorithme *Diff* permet de déterminer les différences entres deux architectures en entrées, et transcrit ces différences dans un fichier XML *diff*. Ces différences sont décrites en terme d'*ajout*, de *suppression*, d'élément d'architectures. L'algorithme parcourt chaque élément (composant, connecteur, lien, composant type, etc.) de la première architecture : si un élément n'existe pas dans la deuxième architecture, une instruction d'ajout est alors ajoutée au fichier de *diff*, si un élément existe dans la deuxième architecture et pas dans la première, alors une instruction de *suppression* est ajoutée au fichier *diff*.

Annexe B

Tableau récapitulatif des approches d'évolution étudiées

Le tableau suivant récapitule le positionnement des différentes approches par rapports aux dimensions de l'évolution d'architectures logicielles.

ADL	***Objet de l'évolution***	***Type de l'évolution***	***Support de l'évolution***
Wright	**Composant type**	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
	Connecteur type		**Explicite** (Évolution avec rupture) - Définition des contraintes
	Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contraintes
		Dynamique	**Prédéfini** (Évolution anticipée et avec rupture) Modification configuration par : - Ajout/ suppression composant et/ou connecteur - Ajout/ suppression attachements (attach et detach)
	Composant	**Dynamique**	**Prédéfini** (Évolution anticipée et avec rupture) - Création/ suppression d'un composant
	Connecteur	**Dynamique**	**Prédéfini** (Évolution anticipée et avec rupture) - Création/ suppression d'un connecteur
Darwin	**Composant Type**	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
	Configuration	**Dynamique**	**Prédéfinie** (Évolution anticipée et avec rupture) Modification configuration par : - Instantiations dynamique et paresseuse d'un composant - Ajout d'un binding (bind)
C2SADEL	**Composant Type**	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique - Sous-typage de *nom*, d'*interface* de *comportement* et d'*implémentation*
			Explicite (Évolution avec rupture) - Définition des invariants
	Configuration	**Statique** **Statique**	**Prédéfini** (Évolution avec rupture) - Modification du mode de filtrage du connecteur (Évolution avec rupture) - Définition des invariants
		Dynamique	**Prédéfini** (Évolution non anticipée et avec rupture) Modification configuration par : - Ajout/suppression/remplacement composant

... suite page suivante...

ADL	*Objet de l'évolution*	*Type de l'évolution*	*Support de l'évolution*
			- Ajout/suppression/remplacement connecteur - Ajout/suppression d'attachement (*Weld*, *unweld*)
	Composant	**Dynamique**	**Prédéfini** (Évolution non anticipée et avec rupture) - Création /suppression composant
	Connecteur	**Dynamique**	**Prédéfini** (Évolution non anticipée et avec rupture) - Création /suppression d'un connecteur - Modification connecteur par : Ajout/ suppression d'interface Modification du mode de filtrage du connecteur
ACME	**Composant type**	**Statique**	**Prédéfini** (Evolution avec rupture) - Composition hiérarchique - Sous-typage
	Connecteur Type		**Explicite** (Evolution avec rupture) - Définition des contraintes
	Configuration	**Dynamique**	**Prédéfini** (Evolution anticipée et avec rupture) via des constructions syntaxiques tel que : - La *multiplicité* d'un composant ou d'un connecteur - *Open* pour indiquer qu'un élément est ouvert
UML2.0	**Composant type**	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique **Explicite** (Évolution avec rupture) - Définition des contraintes
	Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contraintes
SafArchie	**Composant Type**	**Statique**	**Prédéfini** (Évolution avec rupture) - Composition hiérarchique
	Interface type	**Statique**	**Explicite** (Évolution avec rupture) - Définition des contrats
	Composant primitif	**Statique**	**Prédéfini** (Evolution avec rupture) Modifier interface composant
	Configuration	**Statique**	**Explicite** (Évolution avec rupture) - Ajouter/supprimer liaison - Ajouter/supprimer composant - Ajouter/supprimer un composite
	Interface	**Statique**	**Explicite** (Evolution avec rupture) Modifier interface par - Ajout d'un port et /ou d'une opération - Suppression d'un port et/ou d'une opération - Déplacement d'une opération
	Composant type		**Prédéfini** (Évolution Continue) - La composition hiérarchique ; - Sous-typage hétérogène ;
	Connecteur type	**Statique**	

... suite page suivante...

ADL	*Objet de l'évolution*	*Type de l'évolution*	*Support de l'évolution*
Mae			- Création de versions ; - Création de composants/ connecteurs variants.
			Explicite (évolution continue) - Utilisation des gardes booléennes - Définition des contraintes
	Interface Type	**Statique**	**Prédéfini** (Évolution continue) - Création de la version d'une interface
	Configuration	**Statique**	**Prédéfini** (Évolution continue) Modifier de la configuration par : - Ajout, suppression ou remplacement d'une version d'un composant, d'un connecteur ou d'une interface par une autre
xADL2.0	**Composant type**	**Statique**	**Explicite** (Évolution continue) En utilisant les schémas de : - Versions - Variants
	Connecteur type		
	Interface Type	**Statique**	**Explicite** (Évolution continue) - schéma de versions
	Subarchitecture (configuration)		
	Composant	**Statique**	**Explicite** (continu) - schéma d'options
	Connecteur		
	Lien		

Table 5: Évolution architecturale dans les ADL

Evolution Structurelle dans les Architectures Logicielles à base de Composants

Nassima SADOU-HARIRECHE

Résumé

Le travail présenté dans cette thèse s'inscrit dans le cadre des architectures logicielles à base de composants. Une architecture logicielle offre une description d'un système à un niveau d'abstraction élevé en terme de composants et d'interactions entre ces composants. La problématique abordée relève de l'évolution structurelle dans les architectures logicielles à base de composants. L'évolution est une nécessité importante dans le monde du logiciel et des systèmes informatiques. Elle permet, dans le cadre des architectures logicielles, d'éviter que celles-ci ne restent figées et soient obsolètes par rapport aux besoins en perpétuels changements. Un autre objectif essentiel est la possibilité de pouvoir élargir les architectures logicielles et d'appliquer le passage à l'échelle, pour prendre en compte de nouveaux besoins ou des fonctionnalités plus complexes. Une architecture doit donc pouvoir être modifiée pour rester utilisable, réutilisable et disponible pour ses utilisateurs, et cela tout au long du cycle de vie du système. Notre contribution à cette problématique se scinde en trois axes :

Le premier axe consiste en la proposition d'un modèle d'évolution dénommé SAEV (Software Architecture EVolution model) permettant l'abstraction, la spécification et la gestion de l'évolution des architectures logicielles. SAEV se veut un modèle générique, uniforme et indépendant de tout langage de description d'architectures logicielles.

Le deuxième axe s'appuie sur deux constats : le premier constat est que les architectures logicielles ne véhiculent pas assez d'informations sur le degré de corrélation entre leurs éléments constitutifs, qui permettraient de déterminer et de propager automatiquement les impacts d'une évolution. Le deuxième constat est que les connecteurs, de par leur position d'intermédiaires entre les éléments architecturaux sont des supports idéaux pour véhiculer les changements entre ces éléments. C'est ainsi que nous proposons d'enrichir les connecteurs par des propriétés sémantiques qui exprimeraient alors la corrélation entre les éléments d'une architecture logicielle qu'ils relient.

Le troisième axe illustre la prise en compte des propriétés sémantiques proposées dans le cadre de l'ADL COSA (Component-Object based Software Architecture). COSA est un ADL hybride qui réifie les concepts communément admis par la majorité des langages de description d'architectures logicielles. Cet axe montre aussi l'application du modèle SAEV sur des architectures logicielles décrites en COSA et en tenant compte des propriétés sémantiques définies.

Mots-clés : Architecture logicielle, ADL, spécification de l'évolution, stratégie d'évolution, règle d'évolution, ECA, propagation de l'évolution, propriétés sémantiques des connecteurs.

Abstract

The work presented in this thesis concerns with the study of component-based software architecture. A software architecture provides a description of a software system at a high level of abstraction in term of components and interactions among these components. The studied problem related to the structural evolution in component-based software architecture. The evolution is a significant issue in the domain of software and information processing systems. It allows architectures to evolve in order to cope with changing requirements. Another essential objective is the possibility of improving the scalability of architectures in order to take into account the new needs with complex functionalities. Therefore, an architecture must accept modifications to remain usable, reusable and available for its users, and this throughout the whole life cycle of the system. Our contribution to these issues is divided into three axes:

The first axis consists of proposing an evolution model called SAEV (Software Architecture EVolution model) allowing the abstraction, the specification and the management of software architecture evolution. SAEV intended to be a generic model, uniform and independent of any architecture description languages.

The second axis is based on two issues: the first issue is that software architecture does not provide enough information concerning the degree of correlation among its components; this correlation would make it possible to determine and to propagate automatically the impacts of an evolution within an architecture. The second issue is that connectors, from their position as intermediaries between architectural elements, are ideal supports to propagate changes within these elements. Thus we propose to enhance connectors with semantic properties that would express the correlation between elements of a software architecture that they connect.

The third axis illustrates the use of semantic properties that are suggested within the framework of the ADL COSA (Component-Object based Software Structures). COSA is a hybrid ADL that supports the commonly admitted concepts by majority architecture description languages. This axis highlights also the application of the model SAEV for software architectures described in COSA taking into account the defined semantic properties.

Keywords: Software architecture, ADL, evolution specification, evolution strategy, evolution rules, ECA, evolution propagation , semantic properties of connectors.

Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.