N^o d'ordre: 72

THÈSE

soutenue

devant l'Université de Bretagne Sud

pour l'obtention du grade de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD

Mention :
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

par

Chouki TIBERMACINE

Équipe d'accueil : Laboratoire VALORIA

Titre de la thèse :

Contractualisation de l'évolution architecturale de logiciels à base de composants : une approche pour la préservation de la qualité

Présentée le 20 octobre 2006 devant la commission d'examen composée de :

Professeur	Mourad	OUSSALAH	LINA, Université de Nantes	Président
Professeur	Laurence	DUCHIEN	LIFL,Université de Lille I	Rapporteur
Professeur	Marianne	HUCHARD	LIRMM,Université de Montpellier II	Rapporteur
Professeur	Flavio	OQUENDO	VALORIA,Université de Bretagne Sud	Examinateur
MCF HDR	Salah	SADOU	VALORIA,Université de Bretagne Sud	Directeur
MCF	Régis	FLEURQUIN	VALORIA,Université de Bretagne Sud	Encadrant

Remerciements

Je tiens tout d'abord à adresser mes vifs remerciements à mesdames Laurence Duchien et Marianne Huchard d'avoir rapporté avec un grand soin ce mémoire. Mes remerciement vont aussi à Mourad Oussalah qui m'a fait l'honneur de présider le jury de soutenance.

Je tiens à exprimer mes sincères remerciements à Flavio Oquendo pour sa participation au jury et pour sa profonde examination de ce travail.

Je remercie chaleureusement mon directeur de thèse Salah Sadou. Sans lui, sans son immense gentillesse et sans son grand soutien, ce travail ne serait pas abouti. Je remercie Régis Fleurquin pour ses précieux conseils et son encadrement exemplaire.

Je souhaite remercier Franck Poirier et Pierre-François Marteau, directeurs successifs du VALORIA, ainsi que Sylviane Boisadan, secrétaire du VALORIA, de m'avoir formidablement accueilli au sein du laboratoire. Je les remercie pour tous les moyens mis à ma disposition.

J'exprime mes remerciements les plus chaleureux à Laure Sadou pour la relecture de mon mémoire et des différents documents que j'ai eu à rédiger.

Je remercie très amicalement Didier Hoareau et Eugen Popovici, compagnons de bureau, ainsi que Reda Kadri et Bart George, compagnons de l'équipe SE, pour leurs commentaires constructifs.

En dernier, mais pas le moindre, je remercie ma femme, mes parents, ainsi que tout le reste de ma famille et de mes amis pour leur soutien.

Résumé

De toutes les étapes qui constituent le processus de maintenance, l'étape de compréhension d'une application avant son évolution, et l'étape de vérification de sa progression et de sa non régression après évolution sont de loin les plus coûteuses. Dans cette thèse, je présente une approche qui diminue les coûts associés à ces deux étapes, pour des applications conçues à l'aide de composants. Dans cette approche, les liens unissant les attributs qualité d'une application et les décisions architecturales sont documentés formellement. La définition de ces liens se fait à chaque étape du cycle de vie de l'application. J'ai développé un environnement d'assistance à l'évolution, qui exploite cette documentation, pour d'une part, garantir sa mise à jour et d'autre part, notifier au développeur les effets possibles sur les exigences qualité des changements architecturaux proposés. Cette documentation forme ainsi un contrat dit d'évolution. Il s'agit d'un accord entre l'architecte de l'application et le responsable de son évolution. Le premier doit expliciter ses décisions de conception, sous la forme d'un ensemble de contraintes, afin qu'elles puissent être respectées lors du développement et de l'évolution. Le second doit respecter ce contrat afin de bénéficier d'une garantie de préservation de la qualité initialement requise. Ceci permet, principalement, de réduire les coûts associés aux tests de non-régression sur l'aspect non-fonctionnel.

Cette même documentation est utilisée pour tracer toutes les décisions architecturales tout au long du processus de développement d'un logiciel. Il est possible, ainsi, de vérifier, à une étape donnée du processus, la non-altération de toutes les décisions prises dans les étapes en amont. J'ai proposé un outil permettant de transformer vers une représentation unique, puis évaluer, les contraintes définies lors des différentes étapes. Cette traçabilité des décisions a pour objectif de rendre persistants les attributs qualité qui sont à l'origine de ces décisions, à travers toutes les étapes du cycle de vie d'un logiciel.

Abstract

Among all activities in the maintenance process, application understanding before evolution, and checking its progression and non-regression after evolution are by far the most expensive. In this thesis, I present an approach which aims at reducing costs associated to these two activities for component-based applications. In this approach, links unifying quality attributes of the application to architectural decisions are formally documented. The definition of these links is made at every phase in the life cycle. I developed an environment for evolution assistance, which uses this documentation. It ensures on the one hand that this documentation is updated, and on the other it notifies application evolvers the possible effects of her/his changes on quality requirements. This documentation forms thus a contract, said an evolution contract. It represents an agreement between the application architect and its evolver. The first should make explicit her/his design decisions, in the form of a set of constraints that should be respected during development and evolution. The second should respect this contract in order to gain the guaranty of preserving initial quality requirements. This allows mainly to reduce costs associated to regression testing at the non-functional aspect.

This documentation is used for tracing all architecture decisions throughout a software development process. It is possible thus to check in a given phase of this process, the non-alteration of decisions made in upstream phases. I proposed a tool which allows to transform to a unique representation, and then evaluate, architecture constraints defined during different phases. This traceability of decisions has as a goal to make persistent quality attributes, which are origins of these decisions, throughout the phases of a software life cycle.

Table des matières

Table des figures

Liste des tableaux

Première partie

Introduction

Chapitre 1

Problématique de l'évolution du logiciel

Sommaire

1.1 Prologue

La première loi de Lehman, issue de constatations sur le terrain, stipule qu'un logiciel doit évoluer faute de quoi il devient progressivement inutile [88]. Bien qu'ancienne cette loi n'a jamais été démentie. La maintenance coûte même de plus en plus cher. Estimée dans les années 80 et 90 à environ 50 à 60 % [90, 100] des coûts associés aux logiciels, de récentes études l'évaluent désormais entre 80 et 90 % [40, 139]. La réactivité toujours plus grande, exigée des applications informatiques de plus en plus complexes, supports de processus métiers évoluant eux-mêmes de plus en plus vite, explique cette tendance. La maintenance est donc, plus que jamais, une activité aussi incontournable que coûteuse. De toutes les étapes qui constituent le processus de maintenance, l'étape de compréhension d'une application avant son évolution, et l'étape de vérification de sa non régression après évolution sont celles qui consomment le plus de ressources. L'étape de compréhension de l'architecture, à elle seule, compte par exemple pour plus de 50 % du temps de maintenance [12]. Ce coût élevé est dû principalement aux raisons suivantes :

– des documentations incomplètes et rarement mises à jour : en effet l'évolution est souvent effectuée dans l'urgence pour ne pas accroître le budget initialement fixé et également pour satisfaire le plus rapidement possible les demandes des clients,
– des documentations ambiguës : la cause principale de ceci est la réalisation de l'évolution par des personnes tierces qui n'ont pas forcément participé au développement du logiciel et à la préparation de sa documentation,
– des documentations non prises en compte de manière automatique par les outils actuels de rétro-ingénierie encore limités à la génération de vues trop proches du code.

Si les tests de non régression peuvent, quant à eux, être automatisés en partie sur leur composante fonctionnelle avec des outils du commerce, ce n'est généralement pas le cas de leur composante non fonctionnelle. Mais, même dans l'hypothèse d'une automatisation suffisante, la vérification reste tardive et génère d'inévitables et coûteux allers et retours entre la phase de test et celle de développement.

1.2 Illustration de la problématique

Tout au long de ce mémoire, un exemple simple représentant un système de contrôle d'accès à un musée (nommé MACS) sera utilisé[1]. La Figure 1.1 fournit un aperçu de son architecture. Le système reçoit en entrée les données nécessaires pour l'authentification des utilisateurs (le composant Authentificateur). Après identification, les données sont envoyées au composant de contrôle d'accès (CtlAcces). Ce dernier consulte une base de données d'authentification (le composant DataStoreAuth) pour vérifier si l'utilisateur[2] est autorisé à entrer dans la zone du musée ou non. Ensuite, il ajoute certaines données (heure d'entrée, la galerie visitée, etc.) aux données reçues et envoie le tout vers le composant archiveur. Le composant ServiceAdmin permet l'administration de la base de données d'authentification. Le composant archiveur fournit une interface pour rendre persistantes localement les archives (journaux). Cette interface est utilisée par le composant DataStoreArchiv qui fournit une interface pour un composant de fouille de données (ServiceFouilleDonnees). Ce composant implémente une interface qui permet de superviser localement le musée et de consulter les journaux (archives). Les deux composants ServiceAdmin et ServiceFouilleDonnees exportent leurs interfaces via le même composant (AdminDonnees). Après l'archivage local, les données sont transmises (par le composant EmetteurServeur) via le réseau pour le serveur central de l'entreprise responsable de la sécurité dans le musée pour un archivage central.

1.2.1 Décisions architecturales et leurs origines

L'architecture décrite dans la Figure 1.1 a été conçue en prenant en compte les exigences qualité définies dans la spécification des besoins non-fonctionnels. Quelques unes de ces exigences et leurs implémentations sur l'architecture sont présentées ci-dessous.

[1] Ce système n'est pas un logiciel réel à base de composants. A mon avis, il est suffisamment simple à présenter et suffisamment complet afin d'illustrer la problématique et l'utiliser également comme support pour les exemples fournis dans la suite du mémoire.

[2] Les utilisateurs du musée sont les visiteurs, les organisateurs d'expositions, le personnel administratif et le personnel de service.

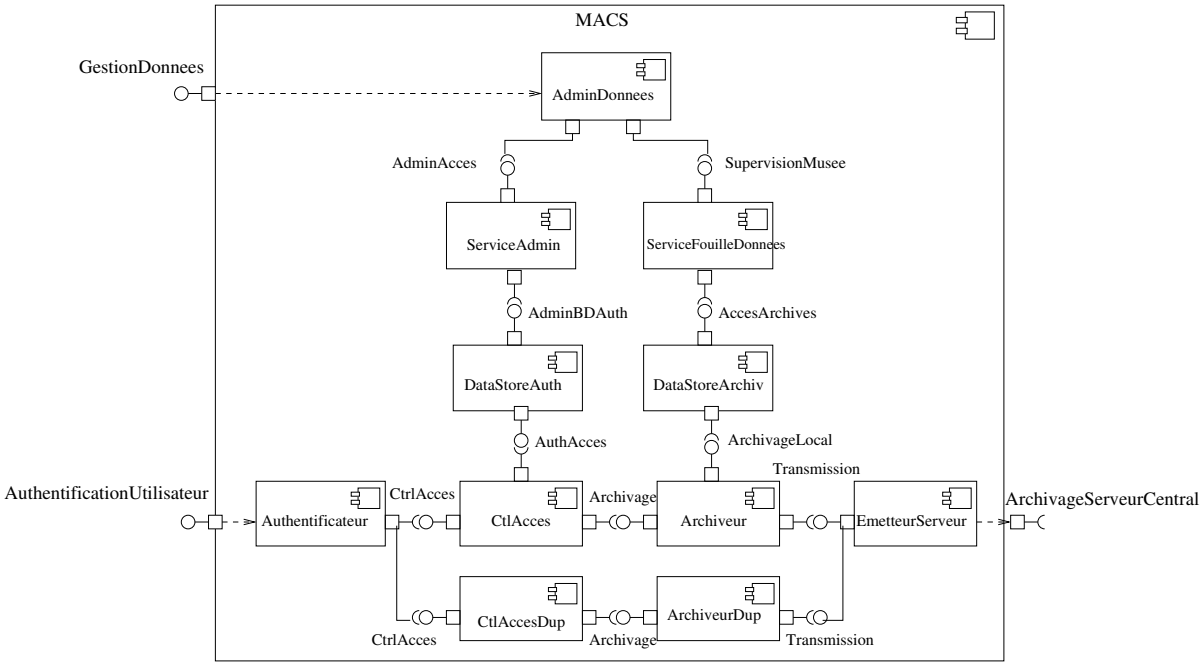

FIG. 1.1 – Une architecture simplifiée d'un système de contrôle d'accès à un musée

1. "*Le système doit être maintenu facilement.*" (cet attribut qualité de maintenabilité est nommé AQ1.) Cet exigence qualité est garantie par le patron architectural système en couches [141], que nous pouvons apercevoir en découpant l'architecture du système verticalement. (cette décision architecturale est notée AD1.)

2. "*Le système doit être portable sur plusieurs environnements différents. Il peut servir différentes applications de supervision de musées ou d'administration des données de contrôle d'accès.*" (Cette propriété de portabilité est notée AQ2.) Afin d'atteindre ce niveau de portabilité, un composant façade -en analogie avec les objets façade [50]- a été conçu comme composant frontal pour l'accès aux sous-composants de MACS. Dans la Figure 1.1, ceci est garanti par le composant AdminDonnees. Toutes les communications des applications clientes pour accéder aux services de gestion des données transitent par ce composant. (cette décision architecturale est nommée AD2.)

3. "*La fonctionnalité de contrôle d'accès doit être disponible dans tous les cas pour le personnel de service.*" (cette propriété de disponibilité est notée AQ3) Dans le bas de la Figure 1.1, la séquence des composants CtlAcces et Archiveur est dupliquée. Ce schéma de redondance (nommé AD3) permet de rendre le système tolérant aux pannes et donc remplit l'exigence de disponibilité. Si l'un des deux composants (CtlAcces ou Archiveur) tombe en panne, la séquence ci-dessous (CtlAccesDup et ArchiveurDup) prend en charge le traitement. Dans ce mode dégradé de fonctionnement, le composant CtlAccesDup autorise l'accès au seul personnel de service. Les journaux sont maintenus dans l'état du

composant ArchiveurDup et ne sont pas persistés dans le composant DataStoreArchiv. Ensuite, les données sont transmises au serveur central.

La séquence de ces composants dupliqués est organisée comme un *pipeline* [141]. Ce style est une spécialisation du style architectural *pipe & filter*. (cette décision est nommée AD4.) Dans un pipeline, chaque filtre (composant) possède une et une seule référence vers le composant en aval par rapport à lui en considérant un flux de données qui transite d'un composant à un autre. Ceci garantit un certain niveau de maintenabilité (couplage minimal, AQ1) requis pour une telle solution d'urgence. Ce pattern garantit également un certain niveau de performance défini dans la spécification des exigences non-fonctionnelles et qui n'est pas détaillé ici. Ce dernier attribut qualité est nommé AQ4.

4. Les développeurs ont introduit un composant d'abstraction de données (ServiceFouille-Donnees), qui abstrait les détails de la base de données sous-jacente. Cette décision architecturale est nommée AD5. En effet, cette pratique traditionnelle garantit le premier attribut AQ1.

1.2.2 Constats lors du développement

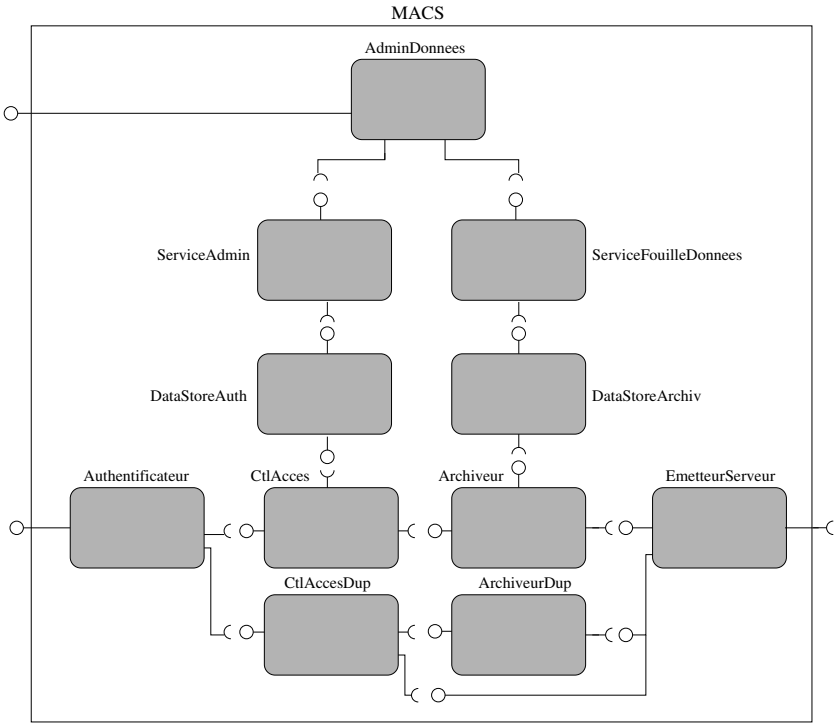

FIG. 1.2 – Une implémentation de MACS en CCM

Disposant de la conception architecturale illustrée dans la Figure 1.1, nous avons décidé d'implémenter le système dans une technologie de composants, par exemple CCM (CORBA

Component Model [119]). Nous supposons qu'il s'agit d'un autre groupe de développeurs qui s'occupe de l'implémentation (des développeurs qui n'ont pas assisté à la conception architecturale du système). Nous avons extrait l'architecture du système après implémentation et le résultat est exposé sur la Figure 1.2. En comparant les deux figures (Figure 1.1 et Figure 1.2), nous avons remarqué l'existence d'un nouveau connecteur entre le composant CtlAccesDup et le composant EmetteurServeur. Après investigation, il s'est avéré que les développeurs qui ont implémenté le système envoient certaines données directement pour l'archivage sur le serveur central. L'archivage local n'était pas utile pour ce type de données.

Bien évidemment, le système ainsi implémenté ne respecte pas le style architectural pipeline décidé pour les deux composants dupliqués (AD4). Ceci affectera par conséquent l'attribut qualité AQ1 (maintenabilité). Notons que l'absence d'informations explicites sur la décision architecturale AD4, durant l'implémentation, a mené les développeurs à un modèle qui ne respecte pas les exigences de qualité initialement établies.

1.2.2.1 Problématique 1 :

"*A une étape donnée dans le processus de développement d'un logiciel, l'absence d'informations sur les décisions architecturales, faites dans les étapes en amont, peut entraîner la perte de ces décisions dans le modèle développé. Par conséquent, les exigences qualité, qui sont implémentées par ces décisions, peuvent ne plus être respectées.*"

Ceci reste valable avec la supposition que les développeurs intervenant à une étape donnée ne sont pas forcément ceux qui interviendront dans les étapes en aval ; ou bien avec les mêmes développeurs, il y a un éloignement temporel entre les différentes étapes dans le processus de développement.

1.2.3 Un scénario d'évolution et ses conséquences

Supposons maintenant qu'après mise en route du système, l'équipe de maintenance reçoit une requête de modification sur le système qu'elle doit implémenter en urgence. Il est demandé à ce qu'on ajoute un nouveau composant représentant un service de notification : (implémenté par le composant NotificationMAJ_BDD). Ce composant notifie les applications clientes, qui ont souscrit à ce service, quand des mises à jour sont faites sur les bases de données de journalisation. Ce nouveau composant exporte une interface *publish/subscribe* via le port qui fournit l'interface GestionDonnees. Ce composant implémente le patron d'interaction *publish on-demand* et utilise directement le composant DataStoreArchiv (voir la Figure 1.3). Ce changement fait perdre au système les avantages du patron façade garanti par le composant Admin-Donnees (AD2) et par conséquent l'attribut qualité de portabilité (AQ2).

Le manque de connaissance pendant l'évolution sur les raisons qui ont conduit les architectes initiaux de prendre de telles décisions peut facilement entraîner l'altération de certaines décisions architecturales et par conséquent, les attributs qualité correspondants. Ce simple exemple illustre comment nous pouvons perdre de telles propriétés. Ceci est souvent détecté (en partie) durant les tests de non-régression. Il est alors nécessaire d'apporter des modifications sur l'architecture une autre fois, et éventuellement itérer à plusieurs reprises.

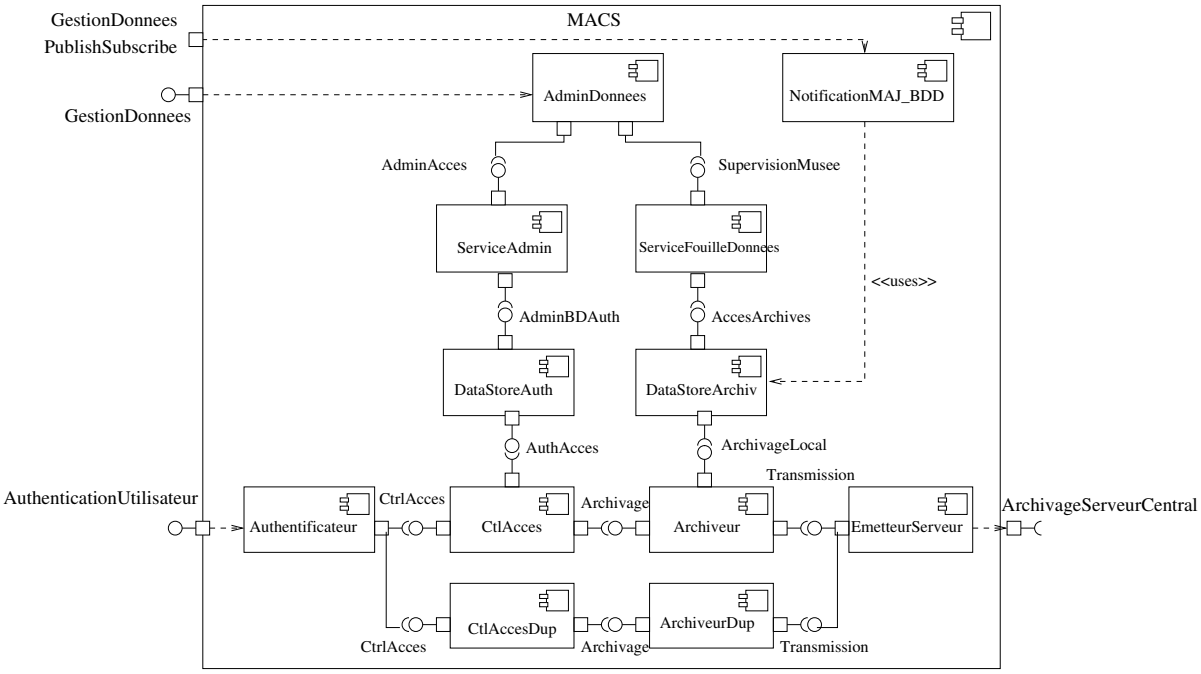

FIG. 1.3 – Une évolution du système MACS

1.2.3.1 Problématique 2 :

"*Lors de l'évolution d'un logiciel, l'absence d'informations sur les décisions architecturales, faites tout au long du processus de développement, peut entraîner la perte de ces décisions dans le modèle obtenu après évolution. Par conséquent, les exigences qualité, qui sont implémentées par ces décisions, peuvent ne plus être respectées.*"

De la même manière que pour la sous-section précédente, nous supposons que l'évolution est faite par de tierces personnes ou longtemps après le développement de la dernière version.

1.3 Problématique dans l'industrie et la littérature

La problématique sur laquelle j'ai travaillé a été constatée par un partenaire industriel avec qui nous avons monté un projet récemment. Elle a été énoncée également par un certain nombre de travaux de recherche dans la littérature.

1.3.1 Problématique dans l'industrie

Les mêmes remarques que celles présentées précédemment ont été faites par une entreprise qui collabore avec l'équipe SE du laboratoire VALORIA. Cette PME (Petite et Moyenne Entre-

prise), appelée Alkante[3], développe des applications Web diverses. Durant les quatre dernières années, cette entreprise a développé et fait évoluer une infrastructure Web pour les systèmes d'information géographique, appelée Alkanet. Ce système a été développé en PHP, Perl et JAVA et est organisé sous forme de modules interconnectés.

Suite à diverses modifications apportées à ce système logiciel, les développeurs de cette entreprise ont constaté un coût élevé dû à l'évolution de certains modules. Ce coût était lié à la perte de l'attribut qualité maintenabilité et portabilité dans ce système. En effet, opérant des changements sur certains modules spécifiques d'Alkanet, les développeurs se sont trouvés plus tard face à un système dont l'architecture n'est pas facilement maintenable. En plus, voulant utiliser certains serveurs du système avec d'autres applications clientes (portabilité), les changements à effectuer n'étaient pas négligeables.

1.3.2 Problématique dans la littérature

La problématique traitée dans cette thèse a été abordée de différentes manières dans la littérature :

- La 7ème loi de Lehman (*Declining System Quality* [88]), révisée pour les logiciels à base de composants dans [87], stipule que la qualité d'un système logiciel diminue à moins que des adaptations rigoureuses soient apportées à ce dernier pour prendre en compte les changements de son environnement opérationnel.
- David L. Parnas a parlé de vieillissement de logiciels (*Software Aging* [128]). Un logiciel vieillit et donc ne peut plus évoluer facilement. Ceci est dû en partie aux changements successifs effectués sur lui sans prise en compte des choix de conception faits initialement (évolution appelée par l'auteur, chirurgie ignorante -*Ignorant Surgery*-). Ce cumul de pertes de choix de conception fait, par conséquent, perdre au logiciel sa qualité et le rend difficile à maintenir et à faire évoluer.
- Dans [93], les auteurs parlent de dérive de logiciels (*Software Drift*). La dérive d'un logiciel est le processus qui cause la déviation des fonctionnalités et de toute la qualité d'un logiciel des spécifications établies avec le client. Les auteurs discutent de la dérive négative et la dérive positive. Comme son nom l'indique, la dérive négative est provoquée lorsque le logiciel ne répond pas aux exigences formulées dans la spécification. La dérive positive est une dérive avec de meilleures fonctionnalités et qualité.
- Lorin et Lindvall parlent d'architectures logicielles qui dégénèrent (*Software Architecture Degeneration*) [63]. Une dégénérescence de l'architecture veut dire que le système conçu devient plus difficile à maintenir et à faire évoluer. Ceci est dû principalement à la négligence de la bonne structuration du code et à sa bonne documentation.

Comme nous le constatons, différents travaux dans la littérature discutent de la perte de certaines propriétés d'un système lors de son évolution. Des efforts doivent être entrepris afin de résorber ce problème. La plupart de ces travaux, ainsi que d'autres travaux (Evolution de Mozilla [57] et de larges systèmes de télécommunication [49, 39]) discutent la perte de qualité et plus particulièrement l'attribut qualité de maintenabilité.

[3] Site Web de l'entreprise : www.alkante.com

1.4 Organisation du mémoire

Après ce chapitre introductif présentant la problématique de recherche traitée dans cette thèse, le reste du document est organisé comme suit :

Partie I : Toujours dans cette partie d'introduction, j'aborderai, dans le deuxième chapitre, le cadre dans lequel se place cette thèse. Ceci a pour objectif d'introduire les concepts de base utilisés dans ce travail.

Chapitre 2 : Dans ce chapitre, le contexte de cette thèse sera donc présenté. J'introduirai trois domaines, à l'intersection desquels se situe ce travail. Ces domaines sont l'évolution et la maintenance de logiciels, les architectures logicielles et les logiciels à base de composants, et enfin l'ingénierie dirigée par les modèles. Un certain nombre de définitions y seront données. Celles-ci forment les hypothèses de travail.

Partie II : Un état de l'art des travaux de recherche en relation avec cette thèse sera présenté dans cette thèse. Afin de résoudre les problématiques énoncées dans ce chapitre, la notion de contrat d'évolution est introduite. Ces contrats représentent une documentation des décisions architecturales prises dans le processus de développement, ainsi que leurs liens avec les propriétés non-fonctionnelles (attributs qualité). Cette documentation sert lors de l'évolution à assister le développeur afin que les changements effectués n'affectent pas les décisions architecturales prises. L'objectif est donc de contrôler l'évolution afin que les propriétés non-fonctionnelles correspondantes ne soient pas altérées. Les travaux connexes à cette thèse s'articulent donc autour de trois thèmes.

Chapitre 3 : Le premier chapitre concerne la documentation des décisions architecturales. Différentes approches de documentation de ces décisions seront présentées. Ces approches adressent la formalisation des contraintes architecturales, des styles architecturaux, et des patrons de conception. Il existe certaines autres approches informelles pour la documentation des décisions.

Chapitre 4 : Ce chapitre s'intéresse aux travaux sur la documentation des propriétés non-fonctionnelles et des attributs qualité. Deux catégories d'approches seront présentées. La première catégorie englobe les approches de documentation des propriétés non-fonctionnelles centrées produits. Elles visent à représenter ces propriétés au sein même des produits (composants, par exemple). Elles sont ensuite évaluées sur ce produit. La deuxième catégorie est composée des approches de documentation dites orientées processus. Elle visent la représentation des propriétés non-fonctionnelles à travers le processus de développement. Une évaluation n'est pas nécessaire par la suite, car le produit final est conforme à ces propriétés.

Chapitre 5 : Ce chapitre sera consacré aux diverses approches de contrôle de l'évolution. Certaines de ces approches s'intéressent à ce contrôle afin de maintenir la cohérence d'un logiciel. D'autres approches visent le maintien d'attributs qualité en général.

Je me positionnerai, dans ces chapitres, par rapport aux différents travaux existants. Je mettrai, surtout, en avant la contribution de cette thèse sur ces divers aspects.

Partie III : Cette partie détaille la contribution de cette thèse. Elle est composée de deux chapitres présentant mes travaux pour répondre aux deux problématiques introduites dans ce premier chapitre. Le troisième chapitre de cette thèse introduit les outils développés pour implémenter les approches proposées dans les deux chapitres qui le précèdent.

Chapitre 6 : Je consacrerai ce chapitre à la présentation du concept de contrat d'évolution comme un moyen de documenter les architectures logicielles afin de préserver les attributs qualité lors de l'évolution. Je discuterai de son utilisation dans l'automatisation de certaines vérifications afin d'assister l'évolution. Ceci contribuera à la résolution de la problématique 2 énoncée ci-dessus.

Chapitre 7 : Ce chapitre présentera une approche pour la traçabilité des décisions architecturales dans le processus de développement d'un logiciel à base de composants. Cette approche aidera à résoudre la problématique 1 citée ci-dessus.

Chapitre 8 : Dans ce chapitre, les outils prototypes que j'ai développés pour concrétiser mes propositions seront présentés. Je discuterai de leur organisation et de leur fonctionnement. Je déroulerai quelques exemples pour illustrer mes propos.

Partie IV : Cette dernière partie englobe trois chapitres qui terminent ce mémoire.

Chapitre 9 : Je présente dans ce chapitre les apports de cette thèse sur les plans conceptuel et de l'ingénierie. Les différentes solutions proposées tout au long de ce mémoire seront classifiées dans ces deux catégories.

Chapitre 10 : Ce chapitre vise à mettre en avant les limites des approches proposées, et fournir des pistes d'amélioration possibles. Ces dernières constituent un travail important que le temps imparti à cette thèse n'a pas permis d'approfondir. Elles constituent des ouvertures pour des travaux futurs.

Chapitre 11 : Le mémoire est terminé dans ce chapitre épilogue.

Chapitre 2

Contexte : évolution, architecture et IDM

Sommaire

2.1 Introduction : intersection de trois domaines

Le travail de recherche présenté dans cette thèse couvre un certain nombre de domaines du génie logiciel. Il se situe à l'intersection de trois disciplines. La première concerne l'évolution et la maintenance du logiciel. Dans ce chapitre, un tour d'horizon et un positionnement dans

ce vaste domaine seront faits, et les définitions retenues seront exposées. Ce point sera suivi par la deuxième discipline, à savoir les architectures logicielles et les composants. En effet, comme énoncé précédemment, c'est le niveau architectural d'un logiciel à base de composants qui est étudié. Le travail présenté porte sur l'évolution et la maintenance de composants, de connecteurs et d'autres éléments de ce grain d'abstraction. Le troisième volet de ce chapitre sera consacré à l'Ingénierie Dirigée par les Modèles (IDM). Dans ce mémoire, deux aspects liés à cette discipline seront abordés. Le premier aspect concerne la méta-modélisation et le second, la transformation de modèles. Comme expliqué dans les chapitres 6 et 7, les problématiques de perte de décisions architecturales et d'attributs qualité sont résolus à l'aide de contrats. Ces contrats sont décrits à l'aide d'un langage se basant sur des méta-modèles. Par ailleurs, la préservation des décisions architecturales d'une étape à une autre se base sur des techniques de transformation de modèles.

2.2 Évolution et maintenance de logiciels

Les termes de maintenance et d'évolution sont souvent utilisés dans la littérature soit conjointement, l'un semblant compléter l'autre, soit indépendamment, l'un semblant pouvoir se substituer ou englober l'autre. Si le terme de maintenance paraît dégager un certain consensus, il n'en est rien du second. J'indique donc, dans un premier temps, ce que désignent ces deux termes en insistant tout particulièrement sur les divergences qu'ils suscitent. Je rappelle, ensuite, le contenu d'un processus de maintenance. Je me focalise, enfin, sur les enjeux et problèmes posés par les deux étapes de ce processus qui nous intéressent : la compréhension et la vérification.

2.2.1 Maintenance des logiciels

Parmi la vingtaine de normes et standards évoquant la maintenance des logiciels, trois sont particulièrement importants et informatifs : ISO 14764 (Software Maintenance), IEEE 1219 (Software Maintenance) et ISO 12207 (Information Technology - Software Life Cycle Processes). Ces trois textes ne diffèrent que peu sur le sens qu'ils prêtent au terme de maintenance. On peut donc se contenter de présenter le sens donné par l'un d'entre eux. Voici, la définition proposée par la norme ISO 12207.

La maintenance est le processus mis en oeuvre lorsque le logiciel subit des modifications relatives au code et à la documentation correspondante. Ces modifications peuvent être dues à un problème, ou encore à des besoins d'amélioration ou d'adaptation. L'objectif est de préserver l'intégrité du logiciel malgré cette modification. On considère en général que ce processus débute à la livraison de la première version d'un logiciel et prend fin avec son retrait.

Il est nécessaire de compléter cette définition donnée en insistant sur deux points trop souvent ignorés. En premier lieu, la maintenance des logiciels est, à la fois, préventive et curative. Le volet préventif cherche à réduire la probabilité de défaillance d'un logiciel ou la dégradation du service rendu. Cette maintenance préventive est, soit systématique lorsque réalisée selon un échéancier établi, soit conditionnelle lorsque subordonnée à un événement prédéterminé révélateur de l'état du logiciel. Les principales actions conduites dans ce cadre visent le plus

souvent, soit à accroître la robustesse d'une application, soit à renforcer son niveau de maintenabilité. Les techniques utilisées sont celles du *Restructuring* (on parle de *refactoring* pour les applications conçues à base d'objets). Bien qu'encore réduit à quelques pourcents des budgets actuels, ce type de maintenance prendra, avec l'élévation du niveau de maturité des entreprises, de plus en plus d'importance. En second lieu, la maintenance comprend des activités aussi bien techniques (tests, modification de code, etc.) que managériales (estimation de coûts, stratégie de gestion des demandes de modification, planification, transfert de compétences, etc.). Elle ne se résume donc pas à l'imaginaire commun qui la cantonne aux seules actions de reprise de code faisant suite à la découverte d'une anomalie de fonctionnement. Son champ d'action est bien plus vaste et fait par conséquent l'usage d'une multitude de techniques, outils, méthodes et procédures provenant de domaines variés.

Le seul point faisant débat reste la place des activités, dites préparatoires, conduites lors du développement initial pour préparer au mieux les maintenances à venir : choix de conception, évaluation d'architectures et de modèles de conception, mise en place de points de variation et de zones de paramétrage, planification des versions et de la logistique de maintenance, etc. En l'état, les textes normatifs, comme le montre la définition précédente, considèrent que ces activités ne relèvent pas de la maintenance.

2.2.2 Évolution des logiciels

Le terme *évolution* est dans la littérature un terme éminemment ambigu. Le seul point sur lequel tout le monde s'accorde et qui lui vaut un usage soutenu est qu'il renvoie une image très positive. Certains laboratoires et revues internationaux dédiés à la maintenance ont ainsi changé leur intitulé pour inclure ce terme. Ce simple mot semble à même de dépoussiérer une discipline qui n'a jamais été considérée comme étant d'un grand attrait. Il profite en cela d'une image favorable provenant de son usage dans les disciplines du vivant. Il évoque des aspects évolutionnistes et suggère l'existence d'une véritable théorie restant à découvrir pour les logiciels. Cependant, en parcourant la littérature, on peut distinguer trois écoles de pensée.

La première et la plus ancienne de ces écoles date de la fin des années 60. Les chefs de file de ce courant sont Lehman et Belady qui publièrent en 1976 une étude empirique, considérée comme fondatrice pour ce courant de pensée, mettant en évidence des phénomènes qui semblaient transcender les hommes, les organisations, les processus et les domaines applicatifs. Ces phénomènes ont été formulés sous la forme de lois qui furent, par la suite, révisées quelques 20 années plus tard [86]. Dans cette mouvance, qui depuis n'a cessé de se développer profitant de l'émergence de l'open-source et en particulier des projets Linux et Mozilla [134], le terme évolution désignait l'étude de la dynamique, au fil du temps, des propriétés [131] d'un logiciel et des organisations impliquées (taille du logiciel, efforts, nombre des changements, coûts, etc.). La filiation scientifique avec les théories évolutionnistes du monde du vivant fut même consommée lorsque des travaux commencèrent à rechercher de nouvelles lois en s'appuyant, non sur une analyse de constatations empiriques, mais sur des analogies avec les lois du vivant dont on cherchait, seulement ensuite, à vérifier la validité dans les faits. A l'époque, cette école positionnait clairement l'évolution comme une discipline nouvelle (Figure 2.1). En effet, l'étude d'une "dynamique" impose bien une nouvelle approche de recherche, il faut collecter des mesures à différents instants de la vie d'un logiciel puis ensuite tenter d'interpréter les

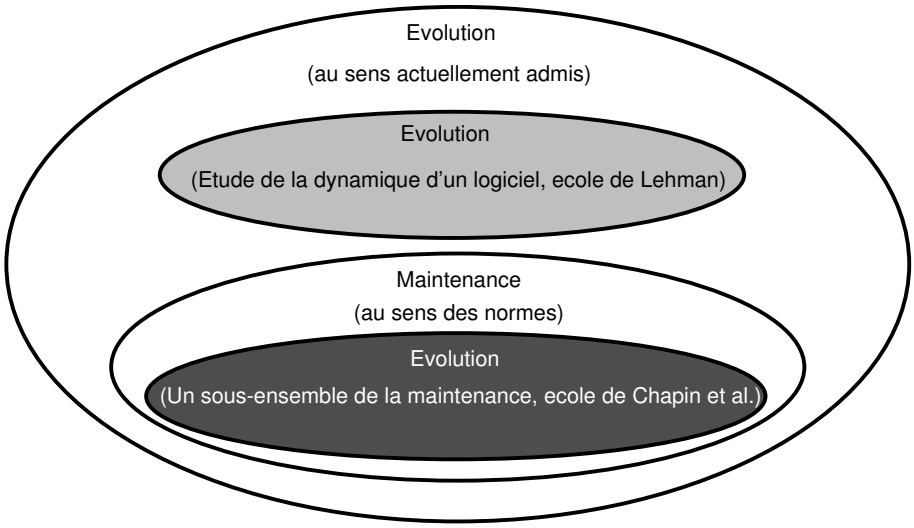

FIG. 2.1 – Écoles de définition du terme évolution

fonctions discrètes obtenues pour chaque variable mesurée. Le lecteur intéressé trouvera une synthèse des travaux entrepris dans le domaine dans [78].

Par la suite, ce terme a été repris de façon opportuniste pour donner un nouvel attrait à certaines catégories déjà anciennes et particulières de maintenance : par exemple l'ajout, le retrait ou la modification de propriétés fonctionnelles. L'évolution est décrite par cette école, comme un sous ensemble des activités de maintenance. Chapin et al. sont des représentants de cette école [23]. Il semble aujourd'hui que cette vision soit devenue minoritaire dans la littérature.

La dernière école est celle qui semble s'être imposée. Elle considère que l'évolution est un terme plus général et plus approprié pour décrire la problématique de la vie d'un logiciel après sa mise en service. Ce terme doit donc se substituer à celui de maintenance. Il étend ce dernier en incluant toutes les études sur la caractérisation des propriétés d'un logiciel, des processus et des organisations dans le temps. C'est cette dernière définition qui est adoptée.

2.2.3 Processus d'évolution

Il y deux façons d'évoquer le processus d'évolution : une vision macroscopique et une vision microscopique. La vision microscopique se préoccupe uniquement de la manière dont se déroule la production de la nouvelle version d'un logiciel depuis sa version actuelle. Elle décrit les activités verticales (analyse de la modification, compréhension du code, test, etc.) et supports (planification de la modification, gestion de la qualité, etc.) qui doivent être conduites pour ce faire. C'est la vision promue par les normes. A l'inverse, la vision macroscopique s'intéresse à des intervalles de temps bien plus grands. De grandes périodes de la vie d'un logiciel appelées phases, durant lesquelles le logiciel semble évoluer d'une manière bien particulière, émergent alors. Dans chacune de ces phases, la majorité des actions d'évolution (mi-

croscopiques) semblent revêtir des propriétés identiques propres à la phase dans laquelle elles se placent ; stratégiquement les demandes d'évolution sont analysées de la même façon et les processus microscopiques mis en oeuvre se ressemblent.

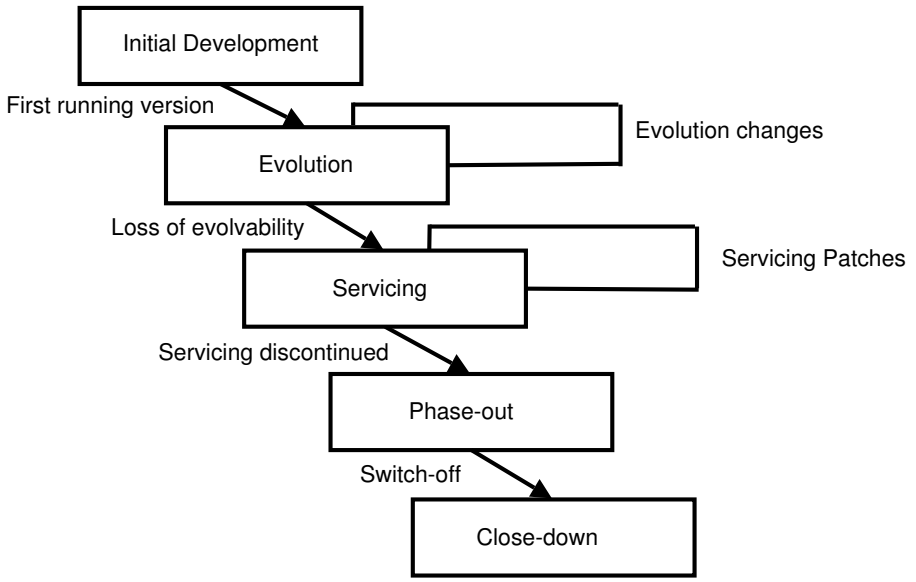

FIG. 2.2 – Macro-processus d'évolution de Rajlich

Le modèle de Rajlich [130] est le modèle le plus connu relevant de cette vision. Il affirme que dans sa vie un logiciel passe par 5 phases (Figure 2.2) : le développement initial, l'évolution, le service, la fin de vie et l'arrêt d'exploitation. Ces phases manifestent que plus on avance dans la vie d'un logiciel, plus celui-ci devient difficile à maintenir du fait, d'une part, de la dégradation de son architecture et d'autre part, d'une perte progressive de l'expertise le concernant. D'abord traitées rapidement et en toute confiance, les demandes de modification, même mineures, posent progressivement des problèmes tels qu'elles ne sont plus traitées (étapes de fin de vie). Ce modèle, dans son esprit, se place dans la droite ligne des travaux sur l'étude de la dynamique des logiciels.

Dans la mesure où l'approche proposée se focalise sur des aspects microscopiques, je ne m'intéresse ici qu'à cette vision du processus d'évolution. La norme ISO 12207 donne une description consensuelle sur le plan microscopique de l'ensemble des processus et activités à suivre. Des activités citées par ce texte, je n'évoquerai pas ici les activités dites de support, pour me limiter à celles directement liées au processus de modification.

La réception d'une demande de changement est le point de départ de toute action de maintenance (Figure 2.3). Cette demande émane, soit d'un client, soit d'un acteur interne à l'entreprise. Elle fait suite à la constatation d'une anomalie de fonctionnement (on parlera alors de cycle de maintenance corrective), au souhait de migrer l'application vers un nouvel environnement matériel ou logiciel (maintenance adaptative), au désir de modifier les aptitudes fonctionnelles ou non fonctionnelles de l'application (maintenance perfective) ou d'améliorer

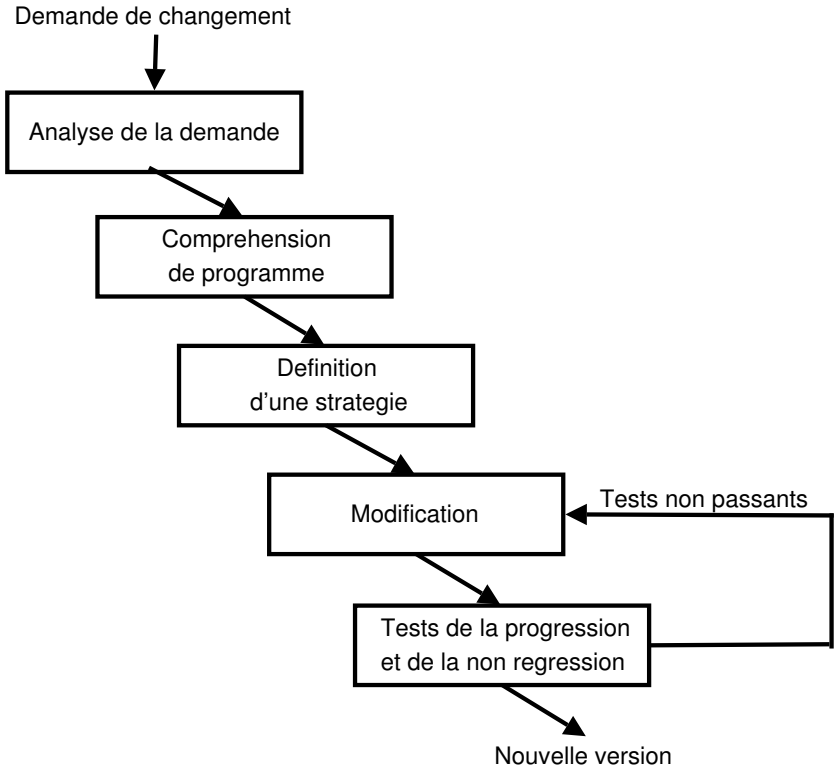

FIG. 2.3 – Micro-processus d'évolution

à titre préventif principalement sa maintenabilité et sa robustesse (maintenance préventive). Là encore, il convient de noter que plusieurs auteurs et normes ont proposé des classifications différentes. Pour une discussion sur ces divergences, on se reportera à [23]. Celle que j'ai donnée ici est, à ce jour, la plus couramment admise.

Cette demande est tout d'abord évaluée pour déterminer l'opportunité de sa prise en compte. Des considérations stratégiques, d'impacts et de coûts vont prévaloir dans cette étape. Des outils, tels que des modèles analytiques d'efforts et de coûts, peuvent être utilisés. Si la demande est acceptée, on commence par se familiariser avec l'architecture de l'application pour développer une connaissance suffisante de sa structure et de son comportement. On identifie ensuite différentes stratégies de modification. Ces stratégies vont être départagées en usant de critères, tels que le coût, les délais, le niveau de qualité garanti ou de compétence exigé, etc. Pour quantifier le coût des différentes stratégies, il est possible d'user d'outils évaluant l'impact des changements envisagés sur le reste du code. La stratégie retenue va, ensuite, être mise en œuvre par une modification effective du code et des documentations associées. On use ici d'outils, de méthodes et de procédures identiques à ceux utilisés lors du développement. Une fois la modification faite, il faut s'assurer que celle-ci n'a pas, d'une part altéré ce qui ne devait pas l'être (non régression) et d'autre part, qu'elle a bien ajouté les propriétés souhaitées

(progression). Une fois la vérification faite, la nouvelle version peut être archivée, diffusée et installée.

2.2.4 Étape de compréhension

Avant d'ajouter, de retirer ou de modifier des propriétés fonctionnelles et non fonctionnelles à une application, il est nécessaire d'acquérir un niveau de connaissance suffisant sur sa structure. Il s'agit, non seulement de reconstruire une image de ses réelles aptitudes fonctionnelles et non fonctionnelles (ce que fait dans sa version actuelle cette application), mais également de la manière dont ces aptitudes ont été obtenues (quelles ont été les décisions architecturales prises). L'étape de compréhension est d'autant plus simple que la documentation fournie avec le logiciel est de qualité. Cette documentation doit, en particulier, être complète, pertinente, à jour et non ambiguë. Dans le cas contraire, on peut éventuellement faire l'usage de techniques issues de la mouvance de la rétro-ingénierie (*Reverse Engineering*) et de la compréhension des programmes (Software Comprehension). Les techniques de rétro-ingénierie cherchent à identifier les composants d'un logiciel et leurs relations pour créer une représentation différente ou de plus haut niveau que celle dont on dispose. Et ceci typiquement pour générer, depuis un exécutable du code ou un modèle fonctionnel (diagramme de séquence UML), depuis du code des modèles de conception (par exemple diagramme de classes UML).

Les techniques de compréhension de programme cherchent à produire des modèles mentaux de la structure d'un logiciel à différents niveaux d'abstraction (du code jusqu'au modèle du domaine). Ces modèles sont construits automatiquement depuis le code et les documentations qui lui sont associées. On trouvera un état de l'art de ces dernières techniques dans [135]. Ces deux types de techniques sont passives, elles n'altèrent en rien un système. Elles sont, par contre, partie prenante de cycles de ré-ingénierie dans lesquels, sur la base des résultats qu'elles affichent, des modifications vont effectivement être entreprises.

Si les définitions sont en apparence très proches, les techniques de compréhension se distinguent cependant des techniques de *Reverse Engineering* par le fait qu'elles ne cherchent pas à reconstruire de l'information manquante ou oubliée dans un format usuel (diagrammes UML, graphes d'appels, etc.), mais à faire émerger, dans des formats mentalement parlant donc le plus souvent propriétaires, une information à très haute valeur ajoutée ; une information qui n'avait jamais été explicitée car ne pouvant le plus souvent pas être formulée dans les langages de modélisation utilisés par les informaticiens. Il faut noter toutefois que cette distinction n'est d'une part, pas si évidente selon les travaux et que d'autre part, certains considèrent la rétro-ingénierie comme étant un cas particulier de la compréhension de programme.

Il est curieux de constater que ces deux types de techniques, qui tentent pourtant de résoudre le même problème, sont portées par deux communautés de chercheurs relativement indépendantes. Cela semble lié au fait que la seconde a des préoccupations plus cognitives et didactiques que la première. Quoi qu'il en soit, toutes ces techniques se cantonnent pour le moment à dégager des visions (graphes des appels, diagrammes de classes, etc.) ou des abstractions (patrons de conception) très (voire trop) proches du code source. De telles vues ne permettent pas de dégager des traits architecturaux de plus haut niveau, préalable essentiel à la bonne compréhension d'une application. En supposant même que l'on dispose de techniques offrant les niveaux de visualisation adéquats, la reconstruction automatique du lien unissant le "pour-

quoi" (l'objectif recherché au travers d'un choix architectural) au "comment" (le choix architectural constaté) semble encore un voeu pieux. La documentation reste donc le seul outil fiable capable de maintenir à chaque étape du développement le lien entre une spécification et son implantation. Tout le problème étant, alors, de garantir non seulement la présence d'une telle documentation, mais également sa mise à jour lorsque nécessaire.

2.2.5 Étape de vérification

La vérification de la non régression et de la progression se fait traditionnellement a posteriori, une fois la modification entérinée en constatant in vivo ses effets, par exemple au travers de tests de non régression. Des outils du commerce permettent d'automatiser le jeu de ces tests. De plus, des algorithmes additionnels de sélection de tests ont été proposés pour limiter le nombre des tests à rejouer tout en maintenant le même niveau d'efficacité [47]. Ces travaux usent de techniques de comparaison des graphes de contrôle d'une application avant et après modification pour extraire de l'ensemble des tests le sous-ensemble de ceux qui sont potentiellement affectés par la modification de code réalisée. Par contre, on constate que la vérification de la progression et de la non régression sur la composante non fonctionnelle se prête généralement mal à une automatisation. En effet, elle se fait à l'aide de plusieurs logiciels ad hoc ou du commerce, indépendants et très spécialisés (analyseur de code pour les aspects qualité par exemple). A cette difficulté, on doit également ajouter le fait que cette approche de vérification a posteriori génère d'inévitables allers et retours entre la phase de test et celle de développement. Un test non passant va nécessiter une reprise du code qui sera suivie à nouveau d'une phase de test. Ces allers-retours s'avèrent d'autant plus coûteux que le nombre des tests à rejouer à chaque fois est important.

Il serait donc judicieux de promouvoir, en complément, une vérification a priori. Cette vérification peut être manuelle et prendre la forme de revues, ou de manière plus rigoureuse d'inspection des codes et des documentations. Si l'efficacité de ce type de vérification est prouvée, elle présente l'inconvénient d'être très coûteuse en temps et en hommes. Il serait donc pertinent de proposer un mécanisme approchant mais automatisable. Un tel mécanisme pourrait, par exemple, au moment où l'on exécute la modification sur le code ou sur la documentation, alerter des conséquences de celle-ci. Ces contrôles a priori peuvent permettre de réduire significativement le nombre des erreurs détectées (tardivement) lors des tests et en conséquence diminuer le nombre des allers-retours nécessaires à leur résolution.

Aujourd'hui, la non régression a posteriori est de loin la mieux maîtrisée car la plus simple. L'automatisation de la seconde, bien qu'utile et complémentaire, ne fait l'objet, à ma connaissance, d'aucun travail.

2.3 Architectures logicielles et composants

Dans cette thèse, l'étude sur l'évolution et la maintenance est limitée aux architectures de logiciels à base de composants. Des définitions des différents concepts liés aux architectures et aux composants seront présentées dans les sous-sections suivantes. Les hypothèses de travail y seront également introduites.

2.3.1 Définition d'une architecture logicielle

Malgré le fait qu'il n'y ait pas de consensus sur la définition du terme architecture logicielle, la définition retenue est celle d'ANSI/IEEE car elle paraît être la plus complète [1]. Cette définition précise qu'une architecture logicielle est *l'organisation fondamentale d'un système, incorporée dans ses composants, les relations entre ses composants et leur lien avec leur environnement, et les principes qui régissent sa conception et son évolution* [66].

En effet, une architecture logicielle est la structure gros grain d'un système logiciel. Elle permet de le représenter comme un ensemble d'unités qui interagissent. Pour l'ensemble de ces unités (individuellement et en globalité), elle met en avant les décisions conceptuelles faites durant leur développement ainsi que celles régissant leur évolution.

2.3.2 Rôles d'une architecture logicielle

Une architecture logicielle joue un rôle important dans différents aspects du développement logiciel [52] :

- **L'analyse :** Une architecture logicielle permet de raisonner sur la qualité du système qui va être implémenté (ses performances, sa maintenabilité, etc). En outre, elle permet de faire des vérifications de conformité par rapport à certaines contraintes de style et des vérifications de cohérence, de complétude et de correction.
- **La compréhension :** Une architecture logicielle permet de rendre explicite les décisions architecturales et les raisons de ces décisions. Ceci facilite considérablement la compréhension de l'organisation de systèmes assez larges et complexes (discutée dans la sous-section 2.2.4).
- **La réutilisation :** Une architecture logicielle permet également la réutilisation de descriptions. Elle est organisée parfois selon des patrons architecturaux, qui sont des modèles architecturaux récurrents et réutilisables.
- **L'évolution :** Comme précisé dans la définition ci-dessus, une architecture logicielle peut exposer les stratégies d'évolution du système. Ceci permet de mieux gérer la propagation des changements et d'évaluer les coûts associés à l'évolution.

2.3.3 Documentation d'une architecture logicielle

Une description d'architecture permet de modéliser un système logiciel, à un niveau élevé d'abstraction. Elle permet de modéliser ce système sous la forme :

- d'éléments représentant les unités de calcul et de sauvegarde des données dans ce système, appelés communément composants ou modules,
- d'éléments représentant les interactions entre les éléments précédents, appelés aussi connecteurs,
- de liens entre les deux types d'éléments précédents et leur organisation (assemblage ou composition). Cette organisation peut être :

[1] D'autres définitions sont disponibles sur le site :
http ://www.sei.cmu.edu/architecture/definitions.html (site consulté le 01 août 2006)

  - **hiérarchique :** un élément architectural donné (composant ou connecteur) peut être décrit à l'aide d'autres éléments (il peut en contenir d'autres),
  - ou **plate :** un élément architectural donné est une boîte noire. Il ne peut être décrit en fonction d'autres éléments.
- de contraintes représentant des choix de conception et des stratégies d'évolution.

Certains travaux [25] considèrent cette manière de décrire, et donc de documenter, une architecture logicielle, comme une vue spécifique de l'architecture. Il existe différentes vues d'une architecture. Chacune cible un utilisateur particulier de cette architecture (la personne responsable de l'implémentation, responsable de la maintenance, en charge des tests, etc.). Dans la littérature, ces vues ont été considérées par plusieurs auteurs. En 1992, Perry et Wolf [129] reconnaissent que plusieurs vues d'une architecture doivent être fournies comme dans le domaine des architectures de construction (plan de plomberie, plan d'électricité, plan des murs, etc.). En 1995, Kruchten propose les 4+1 vues d'une architecture [82] :

- Vue logique : Cette vue supporte la spécification des besoins fonctionnels d'un point de vue comportemental. Typiquement, cette vue est décrite à l'aide des diagrammes de classes et d'objets UML.
- Vue de processus : Dans cette vue, les objets sont projetés en processus. On s'intéresse dans ce cas à l'aspect non-fonctionnel lié à la distribution, la concurrence, la tolérance aux pannes et l'intégrité du système.
- Vue de développement : Cette vue montre l'organisation des modules et des librairies dans le processus de développement. Elle est décrite généralement avec des diagrammes de modules et de sous-systèmes.
- Vue physique : Cette vue projette les autres éléments aux noeuds de traitement et de communication. Elle s'intéresse donc aux attributs qualité de performance et de disponibilité.
- Vue "plus un" : Dans cette vue, les différentes autres vues sont projetées entre elles. Ceci permet de voir les interactions entre les vues et de modéliser certains aspects qui sont transversaux (générer d'autres vues). Par exemple, la vue d'exécution peut être modélisée en combinant les vues processus et physique.

En pratique, il n'existe pas de notation unique pour répondre à tous ces besoins de documentation des vues. Certains auteurs voient qu'UML va fournir dans le futur les moyens de décrire toutes ces vues. A l'opposé, d'autres auteurs croient que chacune de ces vues doit être traitée indépendamment des autres.

Quelques années plus tard, cette classification de vues a été enrichie et un standard a vu le jour ; il s'agit de la recommandation pratique ANSI/IEEE 1471-2000. Ce standard prescrit une manière basée sur les vues pour documenter les architectures. La Figure 2.4 représente un extrait du modèle conceptuel pour les descriptions d'architecture fourni dans ce standard. Dans ce modèle, un système possède une architecture décrite par une description architecturale. Celle-ci identifie un ou plusieurs participants dans le cycle de vie du système (*stakeholders*, comme, les développeurs, les clients, les gestionnaires de projets). Chaque participant a plusieurs préoccupations, identifiées par la description architecturale, comme la performance, la fiabilité, la sécurité, la maintenance et la distribution. Une description architecturale est organisée sous la forme d'une ou plusieurs vues. Dans cette recommandation, une vue stipule l'expression de l'architecture selon un point de vue. Ce dernier précise le langage, les méthodes de

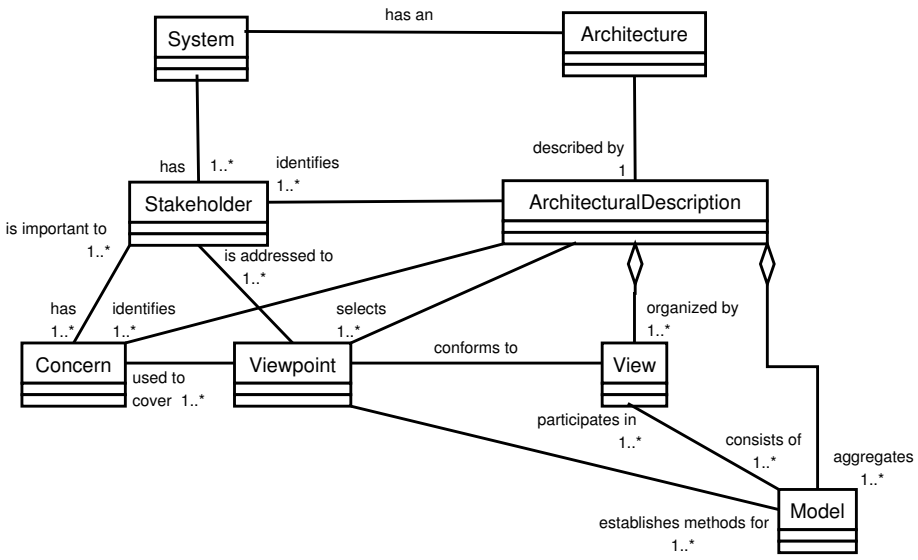

FIG. 2.4 – Extrait du modèle conceptuel du standard IEEE 1471-2000

modélisation et les techniques d'analyse utilisées pour décrire une vue. Ces langages et techniques permettent d'obtenir des résultats en relation avec les préoccupations adressées par ce point de vue. Une vue peut être composée de plusieurs modèles architecturaux. Chacun est développé en utilisant les méthodes établies par le point de vue qui lui est associé. Dans ce standard, il n'existe pas d'ensemble prédéfini de vues. En effet, ce sont les préoccupations des participants qui guident le choix du point de vue. Chaque préoccupation est donc adressée par une vue architecturale. Un point de vue est défini par les éléments suivants :

- le nom du point de vue,
- les participants concernés par ce point de vue,
- leurs préoccupations,
- le langage du point de vue, les techniques de modélisation ou les méthodes d'analyse utilisées
- et éventuellement la source du point de vue (par exemple, une citation littéraire)

Un raffinement de ce standard a été proposé par Clements et al [25]. Ces travaux fournissent des exemples de préoccupations et de participants ainsi que les points de vues répondant à ces préoccupations. Pour l'architecte, par exemple, ils proposent les trois points de vue suivants :

- *Comment l'architecture du système est-elle structurée sous forme d'unités de code ?* à cette préoccupation répond le point de vue modules,
- *Comment est-elle structurée comme un ensemble d'éléments qui ont un comportement à l'exécution et des interactions ?* c'est le point de vue composant & connecteur (C & C) qui satisfait cette préoccupation,
- *Comment est-elle reliée aux structures non-logicielles de son environnement ?* Le point de vue allocation permet de répondre à cette préoccupation.

Selon ces différents points de vue, une description d'architecture est considérée dans cette thèse comme la vue C & C de leur modèle de vue. En effet, les participants dans cette vue sont les architectes et les développeurs chargés de la maintenance et de l'évolution (voir le chapitre 6). Les préoccupations de ces participants sont la structuration du système sous la forme d'éléments de calcul, de stockage de données et de communication, qui sont interconnectés. Ces éléments collaborent pour assurer les fonctionnalités du système. Les langages des points de vue sont les langages de description d'architecture (voir les sous-sections suivantes).

2.3.4 Styles architecturaux

Ces systèmes se conforment parfois à certains patrons dits d'architecture. Ces patrons sont la garantie de certains attributs qualité (comme la maintenabilité, les performances, la fiabilité, etc). Un style ou un patron architectural[2] définit un vocabulaire d'éléments de conception (types de composants et de connecteurs) et l'ensemble de règles de bonne formation (contraintes de style) qui doivent être satisfaites par toute architecture écrite dans ce style.

A ce jour, il n'existe pas de catalogue standard de tous les styles architecturaux existants. Shaw et Garlan [141] ou Buschmann et al. [19] présentent une liste non-exhaustive et empirique des styles. Le *pipe & filter* est un exemple de style architectural. Dans une architecture organisée selon ce style, les composants appelés *filters* reçoivent un flux de données en entrée, font un certain traitement sur ces données, et produisent ensuite un flux de données en sortie. Ces composants sont reliés entre eux par des connecteurs de type *pipe* qui véhiculent ces flux de données.

Ces formes récurrentes et réutilisables sont souvent spécialisées pour obtenir des descriptions d'architecture. La spécialisation de certains styles forme parfois d'autres styles plus précis. Le *pipeline* constitue un bon exemple. En effet, un pipeline est un pipe & filter avec quelques contraintes structurelles additionnelles. Un composant ne peut être connecté qu'au composant qui le suit avec un seul connecteur. De plus, les connecteurs dans le sens contraire de la direction du flux de données ne sont pas autorisés. Ces contraintes vont être détaillées dans les chapitres 6 et 7.

Dans [141], les auteurs fournissent quelques exemples de styles architecturaux. Parmi les styles cités, on retrouve : i) les systèmes de flux de données, comme le pipe & filter ; ii) les systèmes d'appels et retours d'appels, comme les systèmes orientés objets ou les systèmes en couches ; iii) les systèmes centrés sur les données, comme les tableaux noirs ou les systèmes hypertextes.

Certains auteurs considèrent un style architectural comme un point de vue [25]. Une vue est donc une application spécifique d'un style à un système donné. Le style client-serveur est, par exemple, une prescription pour un système où les composants peuvent être des clients ou des serveurs, avec quelques restrictions sur leur topologie et leurs interactions. Une vue client-serveur montre qu'un système S est construit avec 10 serveurs et 50 clients. Chacun représente une unité d'exécution possédant un nom et des propriétés. Cette vue montre aussi que les clients

[2]Dans la suite de ce mémoire, ces deux termes sont utilisés indifféremment. En effet, un style architectural est considéré comme un modèle récurrent d'une architecture. Ceci correspond exactement à la définition que nous pouvons donner à un patron architectural.

ne doivent pas communiquer entre eux, et un client donné ne peut être relié qu'à 3 serveurs au maximum.

Le terme **décision architecturale**, récurrent dans ce mémoire, se réfère à tout choix fait lors de la description d'une architecture donnée. Il peut représenter, par exemple, le choix d'un style architectural donné. Il peut être également –et plus simplement– un invariant architectural, comme un invariant fixant le nombre des sous-éléments qui composent un élément architectural précis.

2.3.5 Langages de description d'architecture (ADL)

Afin que ces architectures puissent être décrites et ces décisions puissent être formalisées, il existe un certain nombre de langages, dits langages de description d'architecture (ADL : *Architecture Description Language*). "*Un ADL est un langage (formel ou semi-formel) qui fournit des dispositifs pour modéliser l'architecture conceptuelle d'un système logiciel, qu'on distingue de son implémentation*" [102]. Ces langages permettent donc de modéliser des systèmes logiciels selon les items énumérés dans la section 2.3.3, et dans certains cas conformes à certains styles architecturaux.

Il existe une multitude d'ADL dans le monde académique et dans l'industrie. xADL [31] ou Acme [55] représentent des exemples d'ADL à objectif général. En d'autres termes, ils fournissent les éléments de base pour décrire une architecture logicielle. Au delà de cette description basique, certains autres ADL se focalisent sur des aspects particuliers de la conception d'architectures logicielles. Koala [156] ou Darwin [95] supportent la modélisation et l'analyse d'architectures distribuées et de leur reconfiguration dynamique. WRIGHT [3] s'intéresse à la formalisation des connexions entre composants. C2SADEL [103] fournit les moyens de décrire des architectures dans un style architectural particulier (C2 [101]), qui se prête bien à la description de systèmes événementiels comme les interfaces graphiques utilisateurs.

Un état de l'art, des abstractions architecturales modélisées dans ces ADL, plus détaillé est présenté dans le chapitre 7. Un autre volet en relation avec les architectures logicielles, et important à discuter dans le cadre de cette thèse, est la qualité logicielle dans les architectures. J'aborde dans la section suivante ce point qui constitue une hypothèse de travail sur laquelle s'appuie cette thèse.

2.3.6 Qualité dans les architectures logicielles

Il est admis que les décisions architecturales sont conduites par les attributs qualité requis dans les documents de spécification [11]. En effet, ce ne sont pas les fonctionnalités attendues d'un logiciel qui déterminent son architecture, mais plutôt la manière avec laquelle ses fonctionnalités vont être fournies. Par exemple, si nous décidons d'utiliser le style architectural de système en couches, c'est en réponse à l'exigence d'un niveau de maintenabilité élevé. Ici, l'attribut qualité est la maintenabilité et la décision architecturale est un système en couches. Dans la section 1.2.1, j'ai présenté des exemples d'exigences sur certains attributs qualité et leurs implémentations sous forme de décisions architecturales.

Dans le domaine de la qualité logicielle, on considère une caractéristique qualité comme toute propriété observable ou mesurable, statique ou dynamique d'un produit ou un processus

logiciel. A un attribut qualité, des métriques peuvent être associées. Une métrique est une mesure qualitative ou quantitative pour observer ou mesurer la qualité. Une exigence qualité est une spécification des valeurs acceptables des caractéristiques qualité.

Un modèle de qualité est une taxonomie des caractéristiques qualité et des relations entre ces caractéristiques. L'objectif de ces modèles est triple : i) une spécification détaillée d'un système logiciel ; ii) une évaluation de la conception ; iii) des tests du système.

Il existe dans la littérature plusieurs modèles de qualité. En 1977, McCall et al. ont fourni une caractérisation des propriétés (facteurs) de qualité basée sur trois perspectives ou préoccupations dans le développement logiciel [99]. La première représente la révision des produits logiciels. Elle est composée des caractéristiques suivantes : la maintenabilité, la flexibilité et la testabilité. La deuxième perspective s'oriente vers la transition de produits. Elle est représentée par la portabilité, la réutilisabilité et l'interopérabilité. La troisième préoccupation concerne les opérations des produits. Elle est constituée des attributs suivants : correction, efficacité, fiabilité, intégrité et utilisabilité. Ces onze caractéristiques qualité sont appelées facteurs. Elles décrivent la vue externe d'un logiciel, telle qu'elle est perçue par ses utilisateurs. Les auteurs ont raffiné ces facteurs en vingt trois critères qualité. Chaque critère décrit la vue interne d'un logiciel, telle qu'elle est perçue par ses développeurs. A ces critères, ils ont associé des métriques.

En 1978, Barry Boëhm a proposé un autre modèle de qualité [13]. Ce modèle ressemble beaucoup au précédent, car il est hiérarchique avec trois niveaux de caractérisation. Le premier niveau (le plus haut dans la hiérarchie) du modèle de Boëhm définit trois préoccupations :

- l'utilité du logiciel, de trois points de vue : i) efficacité, ii) facilité d'utilisation iii) et fiabilité ;
- la maintenabilité ;
- la portabilité.

Le second niveau est constitué de sept caractéristiques : portabilité, fiabilité, efficacité, utilisabilité, testabilité, compréhensibilité, flexibilité. Comme dans le modèle de McCall, le troisième niveau est représenté par les métriques associées aux caractéristique précédentes. Malgré sa ressemblance avec le modèle de McCall, le modèle de Boëm contient une variété plus grande de caractéristiques et se focalise plus sur la maintenabilité, ainsi que les caractéristiques et les métriques qui lui sont associées.

ISO[3] 9000 (et son évolution ISO 9000 :2000) est une famille de standards bien connue dans le domaine de la qualité en général (logicielle ou non). Cette famille est composée de plusieurs standards liés, entre autres, aux processus d'assurance qualité (ISO 9001 :2000), à l'audit des systèmes de gestion de la qualité (ISO 19011 :2000) et à la gestion de la qualité des organisations (ISO 9004 :2000). En plus de cette famille de standards, ISO a fourni en 2001 le standard ISO/IEC 9126 pour l'évaluation de produits logiciels : caractéristiques qualité et les directives pour leur utilisation [67][4]. Ce standard est basé sur les deux standards précédents et ajoute une autre caractéristique, qui représente la fonctionnalité. Il est composé de quatre parties : i) Partie 1 : Modèle de qualité, ii) Partie 2 : Métriques externes, iii) Partie 3 : Métriques internes, et iv) Partie 4 : Métriques de la qualité en utilisation. La partie 1 de ce standard (modèle de

[3]ISO veut dire *International Organization for Standardization* : www.iso.org

[4]*ISO 9126 :2001 : Software Product Evaluation : Quality Characteristics and Guidelines for their Use*.

qualité) définit deux niveaux de caractéristiques qualité : le premier niveau représente les facteurs de qualité classiques (illustrés dans la Figure 2.5), et le deuxième niveau représente les sous-caractéristiques de chacune des caractéristiques du niveau supérieur.

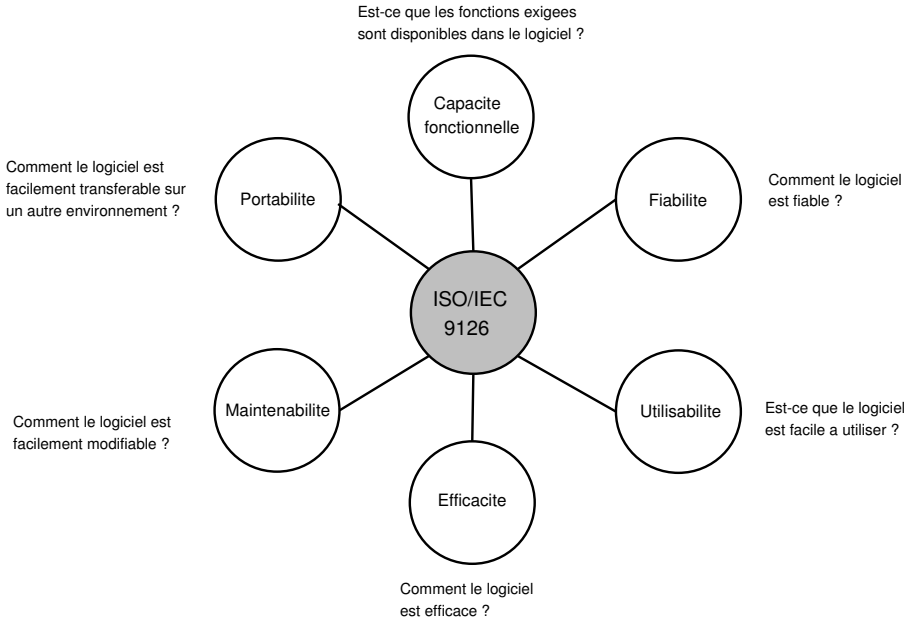

FIG. 2.5 – Extrait du modèle de qualité ISO/IEC 9126

Les différentes caractéristiques qualité du premier niveau sont définies, dans ce standard, de la manière suivante :

- **Capacité fonctionnelle :** C'est la capacité d'un logiciel à répondre aux besoins fonctionnels, de sécurité et d'interopérabilité (*Functionnality*),
- **Portabilité :** C'est la capacité d'un logiciel à être transféré d'un environnement à un autre et la facilité de son intégration, adaptation, installation et co-existence (*Portability*),
- **Maintenabilité :** C'est la capacité d'un logiciel à être facilement analysable, modifiable et testable (*Maintainability*),
- **Efficacité :** C'est la capacité d'un logiciel à fournir ses services de manière efficace vis à vis du temps d'exécution et de la consommation des ressources système (*Efficiency*)
- **Utilisabilité :** C'est la capacité d'un logiciel à être attractif, facilement compréhensible et opérable (*Usability*),
- **Fiabilité :** C'est la capacité d'un logiciel à fournir ses services dans des conditions précises et pendant une période déterminée (*Reliability*).

En 2002, le SEI[5] propose un modèle de qualité spécialisé pour les architectures logicielles [11]. Ils classifient les attributs qualité selon trois niveaux différents. Le premier niveau représente les qualités du système. Il est constitué des attributs suivants : disponibilité, mo-

[5] *Software Engineering Institute, Carnegie Mellon University* : www.sei.cmu.edu

difiabilité, performance, sécurité, testabilité et utilisabilité. Le deuxième niveau concerne les qualités d'affaire (*business qualities*). Les auteurs citent les attributs suivants : le temps de mise sur le marché (*Time To Market*), le coût et les bénéfices, la durée de vie projetée du système, le marché visé, l'ordonnancement de la distribution du produit (*Rollout Schedule*) et l'intégration aux anciens systèmes. Le dernier niveau concerne les qualités directement liées à l'architecture : l'intégrité conceptuelle, la correction et la complétude, et la capacité de construction architecturale.

Klein et al. ont proposé des styles architecturaux dits ABAS (*Attribute-Based Architectural Style*) [81]. Ces styles constituent, tout comme les styles architecturaux classiques, des modèles récurrents de conception à la seule différence qu'un ABAS vise un attribut qualité particulier. Ceci permet de mieux cerner les exigences qualité et fournit une description réutilisable d'une solution à un problème récurrent.

Comme vous pouvez le constater dans ce travail, les caractéristiques qualité considérées sont les attributs en relation avec le produit. Ces attributs ne peuvent être observés ou mesurés que statiquement (à l'arrêt du logiciel). Tous les attributs qualité de type processus logiciel, comme la vitesse de découverte ou de fixation de bogues, ou les attributs qualité d'affaire, comme, le *time-to-market* ne sont pas traités, car non implémentés directement par des décisions architecturales.

Je me base, dans le travail présenté dans ce mémoire, sur le modèle de qualité ISO 9126. En effet, ce modèle me semble être le plus approprié pour les besoins de cette thèse, car le plus complet et le plus fin. Il classifie les attributs qualité sous forme d'une forêt, composée de six arbres. Chaque arbre représente un attribut qualité particulier. Ses feuilles sont les sous-caractéristiques exposées précédemment. Comme précisé, ci-dessus, seuls les attributs qualité statiques sont considérés. Les caractéristiques qualité dynamiques comme l'efficacité ou la fiabilité ne sont pas prises en compte, car l'évolution traitée dans ce travail est faite à l'arrêt de l'exécution du logiciel (évolution statique). Notez que, dans la suite de ce mémoire, le terme performance se substitue au terme efficacité, car il est le plus utilisé dans la littérature.

Bien que le modèle de qualité du SEI ait été bien spécialisé pour les architectures logicielles, celui-là est moins complet que le modèle ISO 9126 car il s'intéresse à d'autres types de qualités qui ne sont pas intéressantes dans le cadre de cette thèse, à savoir les qualités d'affaire. Par contre, le modèle ISO 9126 ne concerne que la qualité du produit comme précisé dans [67], ce qui correspond exactement à l'objectif visé dans cette thèse. En effet, l'objectif est de documenter les décisions architecturales liées aux éléments architecturaux (composants, entre autres) et donc les produits et non pas les processus ou le domaine d'affaire.

2.3.7 Définition d'un composant logiciel

Le domaine des architectures logicielles est étroitement lié au domaine des technologies de composants logiciels. En effet, le concept de première classe commun étudié dans ces deux domaines est le composant. La définition de ce dernier reste encore assez vague dans le premier domaine, alors que dans le second, on commence à se mettre d'accord sur certaines définitions comme celle proposée par Clemens Szyperski dans [146]. Cette définition stipule qu'*un composant est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite ses dépendances de contexte uniquement. Un composant logiciel peut être déployé*

indépendamment et est sujet à une composition par de tierces entités.

La spécification UML 2 de l'OMG donne une autre définition du terme composant [122]. Dans cette définition, un composant est perçu comme *une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en terme d'interfaces fournies et requises. En tant que tel, un composant sert comme un type, dont la conformité est définie par ces interfaces fournies et requises (incluant à la fois leur sémantique statique et dynamique).*

On se focalise dans la première définition sur les aspects composition, interfaces contractuelles et déploiement, alors que dans la deuxième, la spécification UML 2 parle de modularité, de substitution, d'interfaces requises et fournies et de typage. Ce sont deux visions différentes des composants : dans la définition de Szyperski, on se place dans un contexte d'assemblage où ces entités (composants) ont déjà été développées et l'on voudrait les assembler pour former une application. Ces composants spécifient des interfaces sous forme de contrats, et peuvent être déployés séparément. Dans la deuxième définition, on se place plutôt dans un contexte de modélisation objet où l'on perçoit le système comme étant un ensemble d'unités modulaires et substituables. Chaque unité (composant) est typée par ses interfaces, qui spécifient les services requis et fournis par ce composant.

Dans ce mémoire, on considère un composant comme étant une unité de composition, qui spécifie de manière explicite, ses services fournis ou requis à travers des interfaces contractuelles. Dans ce mémoire, je me limite à cette simple définition qui est le résultat de la fusion des deux définitions ci-dessus. Je ne m'intéresse donc pas dans mon travail aux aspects de déploiement ou de typage dans les composants.

Dans le domaine des architectures logicielles, on se focalise surtout sur la description haut niveau d'un système logiciel afin de pouvoir analyser ses attributs qualité. En d'autres termes, on se limite aux phases d'analyse et de conception du cycle de vie du logiciel. Dans le domaine des technologies de composants, on s'intéresse plutôt à l'aspect intergiciel (distribution, persistance, sécurité, packaging, déploiement, etc). En d'autres termes, on se limite généralement à la phase d'implémentation, de déploiement et d'exécution dans le cycle de vie. Naturellement, ces deux domaines se complètent. Il est tout à fait envisageable de commencer concrètement le développement d'un logiciel par la description de son architecture dans les phases d'analyse/conception à l'aide d'un langage dédié (voir ci-dessous), pour l'implémenter en composants logiciels.

2.3.8 Architectures logicielles vers assemblages de composants

Dans un cycle de développement logiciel où l'on décrit une architecture dans la phase d'analyse, il est intéressant d'investiguer dans quel sens le développement va s'orienter. Il existe de nombreux travaux qui s'intéressent à la transition d'une architecture modélisée avec un ADL vers un modèle UML. UML est un langage connu par les développeurs et supporté par plusieurs ateliers de génie logiciel. Il est tout à fait envisageable, par la suite, de produire du code à partir de ces modèles de conception. D'autres recherches s'intéressent plutôt à la transition directe d'un ADL à des implémentations à base de composants. Un autre axe de recherche s'intéresse plus particulièrement à la transition entre modèles UML et technologies de composants. Je présente ci-dessous, les travaux menés dans ces directions.

2.3.8.1 D'un ADL vers UML :

Medvidovic et al. présentent une approche pour modéliser les architectures logicielles dans le langage UML [104]. L'objectif de ce travail est de fournir un moyen standard, supporté par la majorité des outils du génie logiciel pour établir des descriptions d'architecture. Les auteurs proposent des projections (*mappings*) de concepts entre les ADL C2SADEL, WRIGHT et Rapide [92] et la version 1.5 du langage de modélisation de l'OMG. Ils introduisent des extensions à UML, basées sur les stéréotypes, les valeurs marquées et les contraintes OCL [121]. En utilisant des diagrammes de classes ou d'états, il est donc possible de modéliser des architectures structurées dans le style C2, des spécifications formelles des comportements de composants et de connecteurs WRIGHT, et des patrons événementiels de Rapide. L'objectif de ce travail n'est pas de fournir un moyen pour transiter entre les deux niveaux de conception, mais plutôt rendre possible la description des architectures C2SADEL, WRIGHT et Rapide dans UML. Par ailleurs, il est tout à fait envisageable d'établir un processus de transition entre les deux, du moment où les projections entre les concepts dans les différents méta-modèles ont été fournies.

En continuité à ce travail, Grünbacher et al. présentent une approche qui appelée CBSP (*Component, Bus, System, Property*) dans [60]. Cette approche vise à raffiner les spécifications des besoins, et à trouver un compromis entre ces spécifications, pour obtenir la description d'architecture la plus appropriée pour le système spécifié. Dans cette approche, les auteurs transitent par des modèles UML. Ces modèles sont spécifiés dans un profil qu'ils ont défini pour le style architectural C2.

Un travail plus récent que celui présenté dans le premier paragraphe a été développé dans [69]. Garlan, Clements et al. discutent une approche pour documenter la vue C & C (la vue Composant et Connecteur discutée dans la section 2.3.3) des architectures logicielles avec la version 2.0 du langage UML. Ils présentent d'abord les changements apportés à UML pour supporter la modélisation des composants logiciels. Ensuite, ils projettent chaque élément architectural présent dans la vue C & C, à savoir les composants, les ports, les interfaces, les connecteurs, les rôles et les configurations de systèmes, vers des concepts UML 2, en l'occurence, les classes, les ports, les interfaces, les associations, les classe-associations, les diagrammes de classes et les diagrammes d'objets. Tout comme dans le travail de Medvidovic et al., ce travail vise à explorer et à rendre possible des descriptions architecturales en UML 2. Avec la proposition de telles projections, il est tout à fait possible d'implémenter des transitions directes entre un ADL comme Acme vers UML.

Un travail antérieur à celui présenté ci-dessus a été proposé par certains autres auteurs. Dans [64, 54], ces auteurs présentent également des projections entre abstractions architecturales et éléments de modélisation orientée objet, à savoir des éléments du langage UML. Ces deux travaux discutent leur expérience utilisant UML et les différentes approches que l'on peut envisager pour modéliser une architecture en UML, mais ils ne proposent pas de méthodes ou d'outils pour obtenir un modèle UML à partir d'une description d'architecture faite avec un ADL particulier.

Dans [73], Kandé et al. présentent un profil UML pour la description des architectures logicielles. Ce profil est basé sur le standard IEEE 1471-2000 [66] et englobe également des concepts provenant des ADL. Ils ont fourni un certain nombre de projections du méta-modèle du standard IEEE vers un méta-modèle UML étendu, ainsi que des projections d'UML vers

l'ADL SADL (*Structural ADL* [115]). Les auteurs ont proposé ConcernBASE, approche pour décrire des architectures logicielles avec UML en instanciant le framework conceptuel IEEE 1471-2000. Ils projettent les concepts de base retrouvés dans ce standard, à savoir les points de vue et les vues. Ils font également une étude de cas sur le point de vue structurel dans ce standard. Ils associent donc aux différentes abstractions structurelles qui composent une architecture (composants, ports, connecteurs et configurations) des éléments de modélisation UML. Disposant d'un profil UML pour la description d'architectures conformes au standard IEEE, les auteurs fournissent également une méthode de transformation de ces modèles UML vers SADL. Cette transformation vise à valider leur approche sur le point de vue structurel du standard et à bénéficier des outils d'analyse et de vérification de SADL.

Goulao and Brito e Abreu proposent une approche pour projeter des descriptions d'architecture dans l'ADL Acme vers UML 2 [58]. Ceci permet de combler le fossé entre les ADL et l'implémentation en transitant par le langage UML (comme illustré dans le chapitre précédent de ce mémoire). Ce travail introduit un profil UML, comme celui de Kandé et al., mais limité uniquement aux descriptions d'architecture Acme.

Dans [126], Flavio Oquendo présente un profil UML 2 pour l'ADL ArchWare[6] (ADL proposé par ce même auteur). Tout comme dans le travail d'Ivers et al. [69], l'auteur introduit des projections entre constructions langagières entre un ADL et UML 2. Dans ce cas, l'ADL considéré est ArchWare. Ces projections servent donc à transiter entre cet ADL et le standard UML 2.

Malgré les divergences dans les objectifs des travaux présentés ci-dessus, un point leur est commun. Ces différentes approches partagent le fait que les architectures logicielles peuvent être modélisées en UML ou en des extensions d'UML. Des projections ont donc été fournies entre différents ADL et UML pour valider ces propositions. Ceci ne fait que conforter l'idée de pouvoir décrire des architectures logicielles lors de la conception ; et avant son implémentation, transiter par un modèle UML. (Ceci représente la première activité, concernant la transition entre l'étape de conception d'architecture et l'étape de conception de composants, introduite dans le chapitre précédent.) En effet, la transition UML-code a été depuis des années prise en compte dans la recherche sur le génie logiciel. Dans la partie suivante, ce volet sera discuté, mais je me limite aux technologies d'implémentation à base de composants.

2.3.8.2 D'UML vers les technologies de composants :

Les travaux les plus significatifs dans cette direction sont, sans doute, ceux sur les profils UML pour les technologies de composants EJB et CCM (*CORBA Component Model*). En effet, un effort a été dévoué pour projeter des concepts présents dans les méta-modèles EJB et CCM vers des concepts dans le méta-modèle UML.

L'objectif du profil UML pour EJB [138] est de fournir une approche standard pour la modélisation de programmes JAVA et d'applications à base de composants dans l'architecture EJB (Enterprise JavaBean). Ce profil définit un certain nombre d'extensions UML pour capturer la sémantique des éléments de l'architecture EJB. Comme tout autre profil, les extensions standards fournies sont représentées par des stéréotypes, des valeurs marquées et des

[6] Un autre profil pour UML 1.5 a été déjà proposé par le même auteur [4].

contraintes OCL. Pour chaque élément dans le méta-modèle EJB, par exemple les interfaces JAVA, les méthodes de création de composants EJB (*ejbCreate()*), les interfaces de cycle de vie des composants EJB (*home interfaces*) ; des stéréotypes et des contraintes leur ont été associés. Il est donc tout-à-fait envisageable de modéliser une application EJB entière dans ce profil. La génération de code pour des composants d'implémentation EJB peut être ensuite directement effectuée.

Tout comme le profil pour EJB, le profil UML pour les composants CORBA [124] permet la modélisation UML d'applications à base de composants CORBA. Ce profil introduit un certain nombre de stéréotypes, de valeurs marquées et de contraintes afin de spécialiser UML avec la sémantique du standard *CORBA Component*. Les différents éléments qui composent le méta-modèle CCM, comme *Component*, *Interface*, *Event*, *Emits* ont été projetés vers des éléments de modélisation UML, comme les classes stéréotypées *CORBAComponent*, *CORBAInterface* et *CORBAEvent*, ainsi que des associations stéréotypées, comme *CORBAEmits*. Ainsi, des modèles, représentant des configurations d'assemblages de composants CCM, peuvent être définis. Du code représentant l'implémentation de ces configurations d'applications à base de composants peut être ensuite généré.

Dans ce deuxième point, la deuxième activité, concernant la transition entre l'étape de conception de composants et l'étape d'implémentation de composants (introduite dans le chapitre précédent), a été discutée. L'objectif principal des travaux précédents n'est pas d'assurer cette tâche, mais l'un de leur objectifs secondaires est de faciliter l'obtention d'implémentations de composants EJB ou CCM tout en bénéficiant des avantages qu'offre le standard UML (outils supports, reconnaissance industrielle et académique, etc.).

2.3.8.3 D'un ADL vers les technologies de composants :

Il existe dans la littérature un certain nombre de travaux qui visent la transition directe entre descriptions d'architecture dans un ADL donné et des implémentations à base de composants.

Jeff Magee et al. parlent de transition entre des architectures en Darwin et des objets distribués CORBA dans [97]. Il est vrai que la plate-forme visée n'est pas une plate-forme de composants, mais ce travail est antérieur à des travaux plus récents sur la transition vers des plate-formes intergicielles à base de composants.

Dans [127], les auteurs discutent des différents aspects relatifs aux architectures logicielles qui peuvent être projetés dans les technologies de composants. L'étude est faite sur l'ADL C2SADEL qui est projeté vers CCM. Ils ne proposent aucune méthode de transition de l'une vers l'autre, mais ils concluent que les deux entités se complètent et forment ensemble une base solide pour produire des logiciels à base de composants de qualité supérieure.

Di Nitto et Rosenblum présentent dans [34] une étude faite sur plusieurs ADL, à savoir, Acme, Rapide et Darwin et leur capacité à décrire des aspects d'implémentation intergiciels. Les auteurs ont sélectionné deux styles architecturaux, à savoir C2 et JEDI [28], induits par deux infrastructures intergicielles d'implémentation fournies par les auteurs. Ils ont décrits ces deux styles dans les ADL précédents en projetant chaque abstraction architecturale présente dans un ADL vers un type d'élément architectural dans les styles imposés dans les technologies d'implémentation cibles. L'objectif de ce travail est de capturer (et donc d'enrichir) au plus tôt (la description architecturale) les aspects liés aux intergiciels, comme le support des styles

architecturaux, afin de personnaliser l'application et de pouvoir anticiper certaines vérifications avant le déploiement. L'aspect le plus intéressant dans ce travail dans le contexte de cette section sont les projections proposées par les auteurs qui visent à transformer des abstractions architecturales des ADL en entités des modèles d'implémentation.

Une autre approche a été présentée dans [32]. Dans ce travail, les auteurs parlent de projections entre l'ADL ZCL, proposé par ces mêmes auteurs, et les composants CORBA. L'objectif de ce travail est de fournir à travers les différents outils, développés pour les deux technologies, un environnement de développement allant de la conception, à l'implémentation puis à l'exécution de logiciels à base de composants. ZCL est un ADL, basé sur le langage de spécification formelle Z [143], et ressemble fortement à l'ADL WRIGHT. Dans cet ADL, on remarque la présence des composants, des ports, des interfaces, des connexions, ainsi que d'autres éléments architecturaux. La notion de connecteur n'existe pas dans la version initiale de ZCL. Elle a été ajoutée dans une version étendue de ZCL afin de répondre au besoin de ce travail. Les auteurs se sont focalisés sur l'aspect structurel de ces deux technologies, et ont développé une approche qui implémente des transformations entre ces deux mondes.

Dans [136], les auteurs ont proposé une approche pour combler le fossé entre Acme et CCM. Ils ont proposé une approche pour la transformation de descriptions d'architectures Acme vers des définitions dans le langage IDL CORBA [119]. L'approche présentée est purement déclarative. Les auteurs discutent d'un outil qui implémente cette approche avec XMI et XSLT. Certaines adaptations ont été effectuées sur Acme, exploitant ces capacités d'extension, pour qu'il soit projetable vers CCM.

Dans les travaux présentés ci-dessus, nous remarquons, que pour certaines abstractions architecturales dans les ADL, il n'existe pas de projections directes vers des abstractions dans une technologie de composants. Des adaptations sont donc nécessaires afin de garantir un passage d'un niveau à un autre avec préservation de la sémantique des éléments architecturaux transformés.

Le passage par des modèles UML constitue la solution qui facilite le passage d'un ADL vers une technologie de composants. En effet, nous remarquons que plusieurs travaux ont oeuvré avec succès sur le passage d'un ADL vers UML. De plus, les modèles UML résultants peuvent être facilement projetés vers des profils UML, et à partir de là, vers des technologies d'implémentation à composants. Ceci constitue l'approche présentée dans le chapitre précédent et l'hypothèse de travail (d'un processus de développement de logiciel à base de composants) sur laquelle je me suis appuyé dans cette thèse.

2.4 Ingénierie dirigée par les modèles

Dans le processus de développement de logiciels à base de composants que j'ai adopté dans cette thèse, je me place dans un contexte d'ingénierie dirigée par les modèles. En effet, je me mets dans le contexte (tel introduit dans la section précédente) où on transite successivement de modèles de conception vers des modèles d'implémentation. Dans cette thèse, je traite deux aspects importants dans ce domaine d'ingénierie. Le premier aspect est relatif à la méta-modélisation et le second à la transformation de modèles.

2.4.1 Méta-modélisation

Par définition, la méta-modélisation est l'activité de réalisation de méta-modèles. Un méta-modèle expose les constructions syntaxiques, leurs sémantiques, les liens qu'elles entretiennent entre elles, et les règles de bonne formation, nécessaires pour la définition de modèles donnés. Il représente donc la grammaire d'un langage de modélisation. Certains auteurs considèrent un modèle comme étant une abstraction d'un système physique, qui permet de prédire la qualité du système, d'analyser certaines propriétés lors de l'évolution ou de communiquer avec les autres intervenants dans le cycle de vie du système [16]. D'autres auteurs ne distinguent pas le code des modèles. Un modèle est défini comme une spécification formelle des fonctions, structure ou comportement d'un système [118]. Par spécification formelle, l'OMG veut dire, dans cette définition, que le langage utilisé pour établir cette spécification doit avoir une syntaxe et une sémantique bien définies, avec éventuellement des règles d'analyse, de preuve ou d'inférence pour ces constructions. Les langages de programmation vérifient cette définition. Le code est donc considéré comme un modèle, par ces auteurs.

Un méta-modèle possède des objectifs bien précis. Parmi ces objectifs :

1. décrire la syntaxe et la sémantique d'un langage donné pour la meilleure compréhension de ce dernier ;
2. décrire la syntaxe et la sémantique d'un langage donné afin d'automatiser la génération d'outils support pour ce langage (compilateurs, interpréteurs, etc) ;
3. raffiner la sémantique d'un langage donné ;
4. fournir un moyen de stocker et d'échanger des modèles instances de ce méta-modèle.

Ainsi, on peut déduire les différentes catégories d'utilisateurs de ces méta-modèles :

- les développeurs qui interviennent à une étape donnée du cycle de vie d'un logiciel, et qui recourent au langage de modélisation pour développer ces modèles ;
- les développeurs d'outils supports pour le langage représenté par ce méta-modèle ;
- les concepteurs initiaux de ce méta-modèle voulant raffiner la sémantique du langage proposé ;
- les développeurs d'outils utilisant les modèles instances de ce méta-modèle.

Dans le cadre de cette thèse, et comme je le préciserai ultérieurement, j'use de cette technique de méta-modélisation pour représenter des langages de description d'architecture et des technologies de composants. L'objectif précis de cette méta-modélisation est la possibilité de décrire les décisions architecturales définies lors du développement. Ceci rentre dans le cadre du troisième objectif cité ci-dessus. L'utilisateur des méta-modèles définis est donc le développeur. Ce dernier documente les décisions faites au niveau de l'architecture.

Il existe dans la littérature plusieurs travaux visant entre autre à proposer des méta-modèles des ADL ou les technologies de composants existants. Ces différents travaux peuvent être classifiés dans les catégories suivantes :

2.4.1.1 Approches basées sur les technologies de l'OMG

Dans cette catégorie, on retrouve tous les travaux proposant des méta-modèles définis dans les langages de l'OMG (*Object Management Group*[7]), à savoir, UML et MOF [120]. L'objectif principal des méta-modèles définis avec ces langages est de décrire la syntaxe et la sémantique d'un langage donné pour son apprentissage. L'avantage de ces deux technologies de méta-modélisation est la description graphique des méta-modèles. Comme précisé précédemment, ceci facilite considérablement la compréhension du langage représenté.

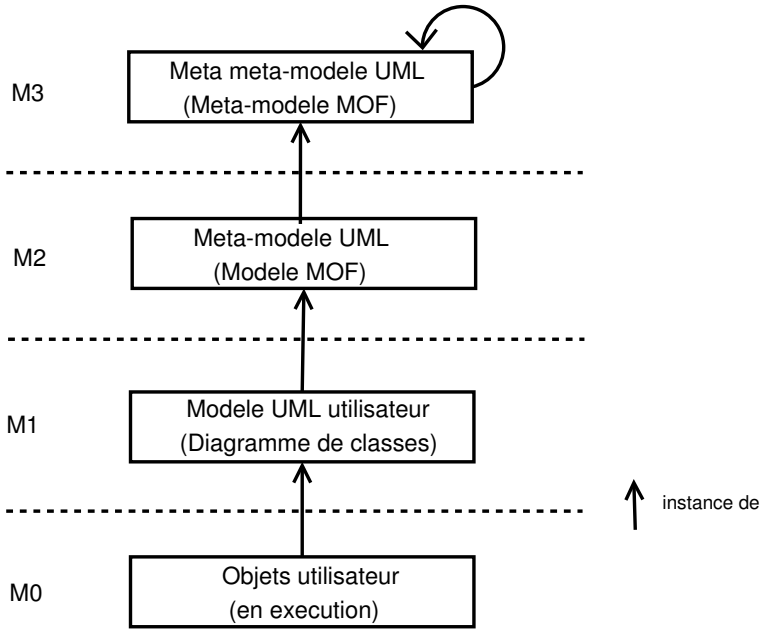

FIG. 2.6 – Niveaux de modélisation de l'OMG

La Figure 2.6 illustre les quatre niveaux de modélisation de l'OMG. Le niveau le plus bas (M0) représente les objets qui s'exécutent. Chaque niveau Mi est le niveau méta de Mi-1. Le niveau M1 représente les classes des objets du niveau M0. Plus généralement, il représente les modèles UML. Le niveau M2 représente le méta-modèle UML, décrit en MOF. Le niveau supérieur représente le méta méta-modèle d'UML et donc le méta-modèle MOF. Il est évident que nous pouvons imaginer d'autres niveaux méta au delà, mais les spécifications de l'OMG se limitent à ces quatre niveaux de modélisation, car ils sont les plus significatifs. Ces spécifications font l'hypothèse que tout niveau supérieur peut être décrit en MOF.

UML est le langage de modélisation objet de l'OMG. Il couvre plusieurs aspects de la modélisation, à savoir, la description statique, structurelle, dynamique et comportementale. Bien que ce standard fournisse un méta-modèle assez riche permettant la spécification de méta-modèles, l'OMG a proposé un sous-ensemble de ce langage dédié à la description de méta-

[7]L'OMG est une organisation internationale dont l'objectif est de promouvoir la théorie et la pratique de la technologie orientée objets dans le développement logiciel. Site web de cette organisation : www.omg.org

modèle, appelé MOF. Les différences entre ces deux langages, extraites de la spécification de MOF, sont précisées dans le Tableau 2.1. Nous remarquons que MOF est une simplification d'UML et une adaptation de ce dernier pour la spécification de concepts présents uniquement dans les méta-modèles. Par exemple, lorsqu'on décrit des méta-modèles, les classes-association ne sont jamais utilisées [120]. C'est la raison pour laquelle, ces dernières ont été omises dans la spécification de MOF. La version officielle de MOF actuellement fournie par l'OMG est la 1.4, qui correspond à la version 1.5 d'UML. La version 2.0 de MOF (correspondant à UML 2.0) est en cours d'adoption.

Le méta-modèle UML	**Le méta méta-modèle MOF**
Association n-aire	Association binaire
Classe-association	Néant
Néant	Constante
Types de données et sous-types	Sous-types de données
Dépendance (classe)	Néant
Généralisation (classe)	généralise (association)
Interface	Classe (comme interface)

TAB. 2.1 – Différences entre abstractions dans UML 1.5 et MOF 1.4

MOF est le standard de l'OMG qui fournit un cadre pour la gestion de méta-données, et un ensemble de services qui permettent le développement et l'interopérabilité de systèmes dirigés par les méta-données et les modèles. [120]. La méta-modélisation MOF joue un rôle important dans l'architecture dirigée par les modèles[8][6]. En effet, dans un cycle de développement où l'on transforme des modèles de type UML, par exemple, d'une étape à une autre, l'aspect méta-modélisation n'est pas de moindre importance. Il est nécessaire, pour passer d'une étape à une autre, de définir des projections entre méta-modèles. Le langage standard de définition de ces méta-modèles, proposé dans MDA, est MOF. La spécification 2.0 de MOF décrit son architecture, qui est composée de deux formalismes : EMOF et CMOF. EMOF (*Essential MOF*) est le sous-ensemble de MOF qui permet la définition basique de méta-modèles en fournissant les concepts de base de diagrammes de classes. CMOF (*Complete MOF*) étend EMOF pour la description de méta-modèles plus complexes en fournissant entre autres des mécanismes d'extension tels que la fusion (*merge*), l'importation (*import*) et la combinaison (*combine*).

MOF a également pour objectif de stocker et d'échanger des méta-modèles entre plusieurs outils, par le biais du format XMI (*XML Metadata Interchange* [123]). XMI est un autre standard de l'OMG, dont le but est la Sérialization en XML (l'écriture dans des documents XML) de méta-modèles définis avec MOF. Plus généralement, la spécification XMI vise la Sérialization de modèles UML.

Il existe une multitude de méta-modèles MOF définis pour les architectures logicielles, pour des ADL particuliers, et pour les technologies de composants. Parmi les travaux qui ont proposé de tels méta-modèles :

– Dans [104] et [74], les auteurs présentent des méta-modèles d'architecture. L'objectif de ces méta-modèles est de proposer des profils UML pour la description d'architecture. En effet, les auteurs de ces deux travaux adaptent UML, à travers ces méta-modèles

[8]MDA : *Model-Driven Architecture* [118]. Site web : www.omg.org/mda

pour pouvoir modéliser des architectures logicielles. Leur objectif correspond au quatrième objectif cité ci-dessus (raffiner le langage de modélisation). Le premier travail utilise, dans l'une des approches proposées, OCL au niveau du méta-modèle UML afin de contraindre certaines constructions dans UML pour qu'elles puissent modéliser des éléments architecturaux. Dans le second travail, les auteurs ajoutent des stéréotypes au méta-modèle MOF d'UML pour représenter des éléments d'architecture. Ils enrichissent également ce méta-modèle avec des concepts provenant du standard IEEE 1471-2000 (voir section précédente).

– Dans [98], les auteurs présentent un méta-modèle MOF de composants. Ce méta-modèle vise à introduire le modèle de composants proposé par ces auteurs. L'usage de MOF est purement illustratif. Les auteurs n'exploitent pas ce langage pour un usage particulier. Les profils UML pour EJB et CCM constituent également de bons exemples de travaux proposant des méta-modèles MOF pour ces deux technologies.
  Il existe également d'autres travaux ayant mis en place des méta-modèles MOF pour plusieurs ADL et technologies de composants. L'objectif de ces méta-modèles est parfois lié à l'introduction d'un nouvel ADL ou d'un modèle de composants. Mais souvent, l'objectif est l'utilisation de ces derniers lors de la spécification de projections et donc implémenter des transformations visant les modèles.

Il existe dans la littérature plusieurs autres approches de méta-modélisation basées sur MOF. Parmi ces approches, citons :

– Ecore : Ecore est le noyau du framework de modélisation EMF d'Eclipse [37]. Il permet la définition de méta-modèles simples. A partir de ces méta-modèles Ecore, le support logiciel EMF permet de générer automatiquement des outils (par exemple, éditeurs et compilateurs) pour les modèles instances.
– Kermeta (Kernel Metamodeling) : Kermeta [154] est un langage pour la définition de méta-modèles. Il étend EMF et Ecore en permettant d'associer de la sémantique aux méta-modèles afin qu'ils puissent être exécutables. Kermeta est un langage orienté objets (classes, héritage, etc.) et utilise de manière intensive le langage OCL.
– KM3 (Kernel MetaMetaModel) : KM3 [71] est un langage simple pour la description de méta-modèles utilisés lors de la transformation de modèles à l'aide du langage ATL [72]. KM3 simplifie MOF et Ecore pour ne contenir que les concepts minimaux pour décrire des méta-modèles sous la forme de diagrammes de classes (14 concepts, plus exactement). Ceci facilite considérablement par la suite la transformation de modèles instances de ces méta-modèles.

2.4.1.2 Approches basées sur les technologies du W3C

Dans cette catégorie, on retrouve tous les travaux proposant des méta-modèles définis dans les langages du W3C (*World Wide Web Consortium*[9]), à savoir, XML Schema [158]. Le W3C est un consortium dont l'objectif est de spécifier et de développer des technologies pour des systèmes interopérables sur le Web. Parmi les technologies standardisées par cet organisme, on retrouve le langage XML (*eXtended Markup Language*) et la spécification XML Schema. A la

[9] Site web de ce consortium : www.w3.org

différence des DTD (*Document Type Definition*, une technologie qui a précédé XML Schema), XML Schema est basée sur XML et offre des possibilités de typage plus évolués.

Récemment, plusieurs ADL basés sur XML ont vu le jour. D'autres sont passés de la syntaxe concrète textuelle classique à une syntaxe XML. Dans [31], les auteurs discutent les avantages de tels ADL. Il est de même pour UML, où le format standard pour le stockage et l'échange des modèles est un format XML (XMI). Les langages de configuration des technologies de composants sont également au format XML. En effet, les avantages de XML sont son extensibilité et sa portabilité sur plusieurs environnements.

XML Schema constitue donc une technologie importante dans la spécification de méta-modèles, sous forme de grammaire XML, pour ce type de langages. C'est donc une technologie bas niveau pour la description de méta-modèles, qui vise principalement l'automatisation de la génération d'interfaces de programmation et de compilateurs pour ces langages[10].

Il existe en pratique plusieurs méta-modèles fournis sous la forme de XML Schema. On retrouve, par exemple, pour les ADL xArch [68] et xAcme [160], et pour les technologies de composants Fractal [17], EJB [145] et CCM [119].

2.4.1.3 Autres approches

Les deux approches ci-dessus constituent des approches de méta-modélisation relativement modernes. D'anciennes approches, comme BNF (Backus-Naur Form [7]) et EBNF (Extended BNF [159]), permettent la définition de grammaires de langages. BNF et EBNF fournissent des notations textuelles pour définir un langage donné (souvent un langage de programmation). Il existe en pratique une multitude d'extensions pour ces méta-langages. Ils offrent tous des possibilités pour décrire plus formellement des langages donnés afin de pouvoir générer des outils supports pour ces langages de manière automatique.

D'autres langages ont vu le jour quelques années plus tard. Ces notations comblent l'aspect graphique qui manquait aux méta-langages. Parmi ces notations, on retrouve les diagrammes de syntaxe (*Railway Tracks*). Elles permettent la visualisation graphique de la syntaxe d'un langage afin de faciliter son apprentissage. Les éléments graphiques utilisés dans ces langages sont assez basiques, comme les flèches ou les barres verticales.

Généralement, les ADL sont présentés à l'aide de ce type de formalismes. Ceci est le cas, naturellement, pour tout langage de modélisation. L'usage des technologies dans les approches précédentes (UML, MOF, DTD et XML Schema) est réservé exclusivement à une utilisation de ces méta-modèles (grammaires, ou plus simplement syntaxes) dans le contexte de l'ingénierie dirigée par les modèles.

Je m'intéresse dans cette thèse à la méta-modélisation MOF de langages de description d'architecture et des technologies de composants. Ces méta-modèles vont être utilisés avec OCL pour la formalisation des décisions architecturales prises lors du développement. Le choix de cette technologie est justifié par le fait qu'elle soit un standard adopté de plus en plus par les outils du génie logiciel. Elle fournit également une représentation graphique, qui rend facile la compréhension et l'utilisation des méta-modèles.

[10] Un exemple d'outils qui permettent ce type d'automatisations APIGen : http ://www.isr.uci.edu/projects/xarchuci/tools-apigen.html

2.4.2 Transformation de modèles

Le second aspect, dans l'ingénierie dirigée par les modèles, traité dans cette thèse est celui de la transformation de modèles. Il est évident que cet aspect est important dans l'ingénierie dirigée par les modèles [140]. En effet, l'un des objectifs principaux de cette discipline d'ingénierie est de transformer successivement des modèles pour aller de modèles haut niveau vers des modèles spécifiques à la plate-forme d'exécution du système modélisé. Dans la littérature et en pratique, plusieurs approches pour la transformation de modèles existent. Czarnecki et Helsen [30] présentent un cadre de classification et de comparaison de ces différentes approches. Ce cadre se base sur les éléments suivants :

- les règles de transformation : ces règles sont composées d'un côté gauche accédant au modèle source et d'un côté droit accédant au (ou générant le) modèle cible. Ils peuvent être des méta-variables (éléments dans le modèle source ou cible), des patrons (fragments de modèles) ou des expressions logiques (expressions exprimant des relations, par exemple). Ces deux côtés peuvent être syntaxiquement séparés ou regroupés dans la même construction syntaxique. Les règles peuvent également être paramétrées, et peuvent être décrites d'une manière déclarative ou impérative.
- la portée d'application des règles : les côtés gauche et droit des règles de transformation peuvent avoir comme portée la totalité du modèle (source et cible, respectivement) ou une partie de ces modèles.
- les relations modèle source - modèle cible : les modèles cibles peuvent être de nouveaux modèles ou des modèles existants. Dans ce dernier cas, les modèles cibles peuvent être les modèles sources ou d'autres modèles mis à jour. La mise à jour peut être destructive ou par extension seulement.
- la stratégie d'application des règles : l'application des règles peut être déterministe et donc le choix de l'endroit où la règle s'applique est fait selon une manière standard (parcours classique d'arbre, par exemple) ; ou bien indéterministe et donc concurrent parfois ou interactif.
- l'ordonnancement des règles : l'ordre dans lequel les règles individuelles vont être appliquées peut être précisé de manière explicite ou implicite. Dans le premier cas, l'ordre est défini par celui qui a spécifié les règles, de manière dissociée des règles, ou bien l'ordonnancement est spécifié à l'intérieur des règles. Dans le deuxième cas (ordonnancement implicite), c'est l'outil interprétant les règles qui détermine l'ordre.
- L'organisation des règles : les règles peuvent être rassemblées et organisées dans des modules. Elles peuvent également réutiliser d'autres règles par héritage ou composition logique.
- La traçabilité : durant la transformation de modèles, les liens de traçabilité entre modèles sources et cibles peuvent être préservés. Ces liens peuvent être ajoutés aux modèles sources, aux modèles cibles ou séparément. La création de ces liens peut être faite manuellement ou de manière automatique.
- La direction : la direction des règles peut être unidirectionnelle ou bidirectionnelle.

Ces travaux associent aux différents critères de classification les langages de transformation de modèles existants. Parmi les langages discutés, je citerai ceux publiés dans la littérature comme VIATRA [157], ceux utilisés dans les outils MDA, comme XDE [65] et ceux soumis

comme réponses à l'appel de propositions pour le standard MOF/QVT (MOF *Query/View/-Transformation* [125]), comme TRL [2] ou MTL [41]. MOF QVT est le standard en cours de normalisation par l'OMG pour fournir un langage qui permet la définition de requêtes, la description de vues et la spécification de transformations sur des modèles.

Dans [51], Gardner et al. présentent une revue des différentes soumissions pour MOF/QVT. Ils proposent tout d'abord un cadre de classification des différentes soumissions. Ensuite, ils placent chaque soumission dans ce cadre. Enfin, ils discutent les recommandations que les soumissions existantes doivent prendre en compte afin de satisfaire les exigences exactes de MOF/QVT. Leur cadre de classification ressemble à celui de Czarnecki et Helsen, mais ne se limite pas seulement aux transformations de modèles. Il traite les deux autres aspects de MOF/QVT à savoir les requêtes et les vues. Brièvement, leurs critères de classification sont :

- les règles de transformation peuvent être définies de manière déclarative, impérative ou hybride (déclarative et impérative à la fois),
- les projections entre concepts dans les deux méta-modèles (source et cible) peuvent être unidirectionnelles, bidirectionnelles ou les deux à la fois,
- les cardinalités des entrées/sorties des transformations peuvent être *1-to-1* pour des transformations d'un modèle source en un modèle cible, *1-to-N*, *N-to-1* ou *M-to-N*,
- les transformations peuvent être dirigées par les modèles sources, les modèles cibles, ou peuvent être arbitraires, c'est-à-dire, parfois c'est le modèle source qui détermine la transformation à effectuer et parfois c'est le modèle cible.

Dans le chapitre 7 de ce mémoire, je présenterai des projections des différents méta-modèles, définis pour les ADL et les technologies de composants, vers un méta-modèle générique pour les architectures. L'objectif de cette transformation sera détaillé dans le chapitre en question. Ces projections entre méta-modèles sont implémentées par la suite comme étant des règles de transformation de descriptions d'architecture et de configurations de composants. Ces mêmes projections servent à implémenter des transformations de contraintes architecturales.

L'approche suivie dans cette thèse pour la transformation de descriptions d'architecture, configurations de composants et de contraintes architecturales (considérés comme modèles) se positionne dans le cadre de classification des approches de transformation de modèles de Czarnecki et Helsen comme suit :

- les règles de transformation : les deux côtés des règles de transformation sont formés par des patrons représentant des fragments du modèle source et cible. Les deux côtés sont regroupés dans la même unité syntaxique et sont décrits avec un langage à la fois déclaratif et impératif.
- la portée d'application des règles : les côtés gauche et droit des règles de transformation ont comme portée la totalité des modèles source et cible respectivement.
- les relations modèle source - modèle cible : les modèles cibles sont de nouveaux modèles. Il n'y a pas de mise à jour faite sur des modèles existants.
- la stratégie d'application des règles : l'application des règles est déterministe. Il n'existe qu'une seule règle à appliquer à la fois lors du parcours du modèle source.
- l'ordonnancement des règles : l'ordre dans lequel les règles individuelles sont appliquées est implicite. C'est l'interpréteur de ces règles de transformation qui définit l'ordre d'application des règles.
- L'organisation des règles : Les règles sont organisées sous la forme de *templates*, qui

peuvent être réutilisées par d'autres règles. Ces templates forment des sortes de librairies pouvant être appelées par d'autres règles de transformation.
– La traçabilité : durant la transformation de modèles, aucun lien de traçabilité entre modèles sources et cible n'est maintenu. Ces liens de traçabilité ne sont pas très importants dans l'approche présentée dans cette thèse (voir section 7.4).
– La direction : Les règles de transformation sont unidirectionnelles.

Par rapport au cadre de classification de Gardner et al., l'approche de transformation suivie dans cette thèse fournit des règles de transformation définies de manière hybride (comme précisé dans le paragraphe précédent). Les projections de concepts architecturaux dans les différents méta-modèles sont unidirectionnelles. Les cardinalités des entrées/sorties des transformations sont *1-to-1*. Par rapport au dernier critère, les transformations sont dirigées par les modèles sources.

2.4.2.1 Transformation de contraintes OCL

Il existe une multitude de travaux sur la transformation de contraintes OCL. Ce genre de travaux est intéressant à citer ici, car les contraintes architecturales, formalisant les décisions architecturales, présentées dans ce mémoire sont écrites dans une version légèrement modifiée d'OCL (voir chapitre 6).

Dans [56], Giese et Larsson présentent une approche qui permet de simplifier les contraintes OCL générées automatiquement par des outils. Les auteurs affirment que les outils, comme ceux permettant l'instanciation de patrons de conception (TCC de Borland, par exemple), génèrent souvent des contraintes OCL assez complexes qui ne peuvent pas être lues et comprises facilement. Il est donc indispensable de les simplifier afin qu'elles puissent être mieux maintenues. Cette simplification est basée sur l'application d'un certain nombre de règles de transformation simples. Ces règles sont interprétées de manière répétée par un moteur de règles, jusqu'à ce que les contraintes ne soient plus simplifiables. Parmi les règles de simplification qui sont appliquées, les règles de types primitifs, les règles de types collections et les règles dépendant du modèle dans lequel navigue la contrainte. Si on prend le cas des types primitifs et plus particulièrement le type logique (*Boolean*), toutes les expressions dans la contrainte OCL du genre false and uneExpression ou true and uneExpression peuvent être remplacées respectivement par false et true. Les auteurs ont établi un catalogue de ces règles de transformation, et ont implémenté un interpréteur de ces règles.

Cabot et Teniente [21], discutent aussi la simplification des contraintes OCL en changeant leur contexte par un autre élément du modèle dans lequel naviguent les contraintes. En effet, les contraintes peuvent être simplifiées en changeant le contexte et en effectuant quelques transformations sur le corps de la contrainte. Principalement, les changements effectués se font sur les navigations, car à partir d'un nouveau contexte, les contraintes changent de parcours entre les éléments du modèle. L'approche proposée vise à produire automatiquement plusieurs alternatives d'une contrainte donnée, où les navigations sont réduites au minimum. Parmi ces alternatives, la meilleure contrainte (la plus simple et lisible) est sélectionnée et remplace l'ancienne contrainte. Tout comme le travail précédent cette approche vise à rendre plus lisible et donc plus maintenable les contraintes générées par les ateliers de génie logiciel. Ces contraintes sont souvent assez longues et complexes.

Un autre travail en cours sur la transformation de contraintes est présenté dans [20]. L'auteur propose une approche pour transformer automatiquement les contraintes OCL attachées à des modèles statiques UML, lorsque ces derniers évoluent. Il fournit un exemple simple de l'évolution d'un héritage entre deux classes, dans un diagramme de classes UML, vers une composition. L'auteur présente les étapes nécessaires pour que les contraintes OCL associées à ce diagramme de classes évoluent elles aussi. Principalement, dans cet exemple, ce sont les navigations de cette contrainte qui changent. L'auteur ne détaille pas l'implémentation de cette approche.

Les deux premières approches visent la transformation des contraintes qui naviguent dans un modèle constant. L'objectif est juste de simplifier ces contraintes. La dernière approche vise la transformation de contraintes qui naviguent dans un modèle qui évolue. L'approche de cette thèse vise à transformer des contraintes qui naviguent non pas dans un modèle, mais dans un méta-modèle (comme précisé dans le chapitre 6). En plus, les contraintes sources et cibles naviguent dans des méta-modèles différents (voir chapitre 7).

2.4.2.2 Transformation d'architectures

Il existe dans la littérature un certain nombre de travaux concernant la transformation, non pas des contraintes, mais des descriptions d'architecture et des architectures logicielles en général. Dans [14], Bosch et al. présentent une approche pour la transformation de modèles d'architecture. Cette transformation vise à garantir les attributs qualité requis dans les documents de spécification. Les transformations dont discutent les auteurs sont : i) l'imposition d'un style ou un patron architectural, ii) l'application d'un patron de conception, iii) la conversion d'attributs qualité en fonctionnalité, et iv) la distribution des besoins qualité sur les éléments architecturaux qui composent le système. Les transformations discutées ici sont des transformations à un niveau haut d'abstraction. Les auteurs proposent une approche générale qui vise, par le biais de la transformation d'une architecture, la garantie lors de la conception des attributs qualité. Dans cette thèse, les transformations traitées sont appliquées sur des descriptions d'architecture, qui sont des vues bien particulières d'une architecture (vues composant & connecteur). Le travail présenté dans ce mémoire se situe en aval, dans le cycle de vie, par rapport au travail de Bosch et al. Dans cette thèse, une supposition est faite, qui consiste à dire que les attributs qualité ont déjà été implémentés dans l'architecture à transformer.

Carrière, Woods et Kazman discutent dans [22] la transformation des architectures logicielles afin de garantir de nouveaux attributs qualité (améliorer la maintenabilité et les performances et rendre plus facile l'amélioration de la fiabilité). Les auteurs présentent une méthode qui vise à transformer des architectures et par la suite inférer les transformations qu'il faut faire sur le code. Ils discutent d'abord l'extraction d'une description d'architecture enrichie d'informations sur les propriétés des éléments architecturaux et leur types (sockets par exemple pour des connecteurs). Ensuite, cette description est utilisée pour analyser l'architecture du système et savoir quels sont les éléments architecturaux à changer afin de satisfaire les nouveaux besoins qualité (transformer les sockets en connecteurs publish/subscribe). A partir de là, des transformations au niveau du code sont déduites. Ces transformations sont définies à l'aide d'un langage à base de règles appelé Refine et appliquées au code. Les transformations discutées ici visent un travail de ré-ingénierie afin de faire évoluer l'architecture et le code d'un

système face à une évolution des besoins en qualité.

De manière similaire, dans [42], les auteurs discutent de trois types de transformations sur des modèles de graphes des architectures logicielles. Après une modélisation sous forme de graphes d'une architecture logicielle à base de modules (composants primitifs) et sous-systèmes (composants composites), plusieurs types de transformations peuvent être appliqués. Chaque type de transformation vise un objectif bien précis : i) la transformation pour la compréhension : elle permet d'expliciter certaines propriétés dans le graphe afin de mieux comprendre la structure de l'architecture. Par exemple, ce type de transformations peut générer les liens entre composants composites à partir des liens entre leurs sous-composants ; ii) la transformation pour l'analyse : elle permet de découvrir des informations sur le système, comme les modules qui interagissent de manière cyclique. Ce type d'informations permet de déterminer d'éventuels problèmes dans le système, et comment l'architecture va être modifiée pour les résoudre ; et iii) la transformation pour la modification : elle permet de changer la structure du système en réponse aux problèmes découverts lors de la transformation pour l'analyse. Tous ces types de transformations sont utilisés dans l'activité de maintenance de logiciels large-échelle.

Les transformations discutées dans les deux papiers présentés ci-dessus visent un même méta-modèle d'architecture (une même technologie), alors que les transformations proposées dans cette thèse transitent d'un méta-modèle à un autre. En outre, l'objectif de ces transformations n'est pas le même. Les auteurs de ces travaux visent l'évolution d'un logiciel face à une évolution de ces besoins de qualité dans le premier cas, et la maintenance corrective dans le second cas, par la transformation, alors que, dans ce mémoire, l'objectif des transformations n'est pas de corriger ou de faire évoluer l'architecture du système.

2.5 En résumé

L'évolution d'un logiciel englobe l'ensemble des activités qu'il subit après sa mise en service. Parmi ces activités, la compréhension de l'architecture avant l'application des modifications et les itérations entre les tests de non-régression et l'application des modifications sont les plus coûteuses. Parmi ces tests de non-régression sur l'aspect non-fonctionnel, on retrouve les vérifications de la non affectation des attributs qualité exigés initialement. Les ayant capturés lors de la conception architecturale, à travers les décisions architecturales, cette partie de tests de non-régression peut être anticipée et évaluée en ligne durant l'application des changements (a priori). Ceci permet de réduire le nombre d'itérations citées ci-dessus.

Une architecture logicielle est ici considérée comme le premier document logiciel où l'on prend les premières décisions de conception. Ces décisions ne sont pas innocentes. Elles sont déterminées par les attributs qualité exigés dans les documents de spécification. Les modèles de composants, tels que EJB ou CCM, en tant que technologies d'implémentation, offrent plusieurs avantages, comme la réutilisabilité et la maintenabilité. Il existe, sans doute, une relation logique étroite entre les architectures logicielles et les logiciels à base de composants. Il est tout à fait envisageable de définir, au moment de la conception, une description d'architecture à l'aide d'un ADL pour l'implémenter dans une technologie de composants. Dans un tel contexte, l'évolution de logiciels à base de composants est perçue comme une évolution gros grain, c'est à dire, une évolution architecturale de ces logiciels. Cette évolution, comme illustré

dans le chapitre précédent, peut entraîner une perte des décisions architecturales, et par là de certains attributs qualité. La solution proposée dans cette thèse pour résoudre les problèmes d'évolution d'architecture à base de composants est construite en partie sur un langage de formalisation des décisions architecturales. Ce langage se base sur un ensemble de méta-modèles MOF des architectures logicielles et des modèles de composants. La préservation des décisions architecturales durant le développement (avant l'évolution) se base sur ce même langage et sur une méthode définie à l'aide de transformations de descriptions d'architecture, de configuration de composants et de contraintes architecturales.

Deuxième partie

État de l’art

Afin de répondre aux problématiques évoquées dans le chapitre 1, la notion de contrat d'évolution est proposée. Un contrat d'évolution est une documentation qui unifie les descriptions formelles de deux aspects logiquement interdépendants dans le processus de développement d'un logiciel à base de composants. Le premier aspect concerne les décisions architecturales prises lors du développement. Le deuxième aspect est lié aux spécifications non-fonctionnelles (de qualité) implémentés par ces décisions.

Dans cette partie, un état de l'art des différents travaux connexes est présenté. Dans le premier chapitre, j'aborderai les travaux portant sur la description des décisions architecturales. Ensuite, les travaux sur la description des spécifications non-fonctionnelles seront détaillés, dans le second chapitre. L'originalité des contrats d'évolution est que les deux aspects sont liés et encapsulés dans un même concept. L'objectif de cela est d'assister l'activité d'évolution architecturale de logiciels à base de composants avec des informations sur la qualité. Cette assistance permet ainsi de contrôler l'évolution de ces architectures. Dans le dernier chapitre, je présenterai les avancées de recherche menées dans ce sens (i.e. sur le contrôle de l'évolution).

Chapitre 3

Documentation des décisions architecturales

Sommaire

3.1 Introduction à la documentation des décisions

Il existe une multitude de langages de description d'architecture qui permettent la documentation des décisions architecturales. Ces langages fournissent les moyens nécessaires pour spécifier des contraintes architecturales ou des styles architecturaux. Il existe également un autre type de langages de formalisation des décisions architecturales, qui permettent de formaliser les patrons de conception. Ces différentes approches seront discutées dans les sous-sections suivantes.

3.2 Langages de contraintes pour les architectures logicielles

En plus de la possibilité de décrire des architectures sous forme de composants, de connecteurs et de configurations, certains ADL permettent la définition de contraintes architecturales. Une contrainte architecturale est un prédicat dans la logique du premier ordre spécifiant un invariant architectural. En d'autres termes, une contrainte définit des restrictions sur les types des éléments autorisés dans la description à laquelle est associée cette contrainte et des règles sur leurs topologies.

Parmi les langages permettant de décrire des contraintes architecturales, on retrouve l'ADL WRIGHT [3]. WRIGHT propose un ADL pour formaliser les descriptions architecturales et plus particulièrement la formalisation des connecteurs. Ce langage est basé sur le langage de spécification formelle Z [143]. Le langage de contraintes de WRIGHT introduit les ensembles et les opérateurs suivants :

- *Components* : l'ensemble des composants dans la configuration,
- *Connectors* : l'ensemble des connecteurs dans la configuration,
- *Attachments* : l'ensemble des liens dans la configuration. Chaque lien est représenté par une paire de paires ((composant,port),(connecteur,rôle)),
- *Name(e)* : le nom d'un élément e, où e est un composant, connecteur, port ou rôle,
- *Type(e)* : le type d'un élément e,
- Ports(c) : l'ensemble des ports d'un composant c,
- *Computation(c)* : le calcul effectué par le composant c,
- *Roles(c)* : l'ensemble des rôles d'un connecteur c,
- *Glue(c)* : la colle représentée par le connecteur c (la façon avec laquelle le connecteur gère l'interaction entre les composants),

En plus des types prédéfinis ci-haut, chaque déclaration, nommée dans une description architecturale, peut jouer le rôle d'un nouveau type de données. La contrainte suivante stipule que la configuration doit avoir une topologie en étoile :

$$\exists center : Components \bullet$$
$$\quad \forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments$$
$$\wedge \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port$$
$$\quad \mid ((c, p), (cn, r)) \in Attachments$$

Le premier prédicat indique qu'il existe un composant ("center") qui est attaché à tous les connecteurs de la description. Le second prédicat indique que tous les composants doivent être attachés à un connecteur. Ainsi, cette contrainte garantit que tout composant est connecté au composant représentant le centre de l'étoile. Des exemples plus complexes seront introduits dans la sous-section suivante.

Un autre langage permettant de décrire des contraintes architecturales est le langage Armani [113]. Ce langage hérite de l'ADL Acme, mais a été conçu spécialement pour capturer l'expertise dans la conception des architectures logicielles. Il permet de décrire trois classes d'expertises de conception architecturale. Premièrement, le vocabulaire de conception, qui représente la forme la plus basique de l'expertise. Ce vocabulaire permet de décrire une architecture sous la forme :

- de composants et de leurs ports,
- de connecteurs et de leurs rôles,
- de systèmes qui représentent des configurations de composants et de connecteurs,
- de représentations qui permettent de décrire de manière hiérarchique des systèmes,
- de propriétés qui représentent des annotations non structurelles attachées aux éléments architecturaux.

Deuxièmement, Armani permet de décrire les règles de conception. Ces règles sont décrites à l'aide du langage de prédicats d'Armani. Ce langage est basé sur la logique de premier ordre avec des termes composables, des fonctions définies par les utilisateurs et des capacités de

quantification limitées (les variables peuvent être quantifiées sur des ensembles finis seulement). Parmi les fonctions prédéfinies dans ce langage, on retrouve les fonctions de vérification des types (par exemple, satisfiesType(e :Element,t :Type) :boolean), les fonctions de graphes (par exemple, attached(con :Connector,comp :Component) :boolean), les fonctions de relation père-fils (par exemple, parent(c :Component) :System), les fonctions d'ensembles (par exemple, size(s :set) :integer). Tout comme les autres langages de prédicats, Armani fournit les opérateurs classiques arithmétiques, booléens, de comparaison, et de navigation dans les propriétés des éléments contraints (le point "."). Comme dans l'ADL WRIGHT (le langage Z), les quantificateurs fournis dans Armani sont le *forall* et le *exists*. Il permet la définition de deux types de prédicats, les heuristiques et les invariants. Ces deux entités sont définies de la même manière, sauf que les heuristiques ne sont pas destinées à des vérifications de type. L'exemple ci-dessous représente un invariant et une heuristique spécifiés en Armani :

```
Invariant Forall c1,c2 : component in sys.Components |
              Exists conn : connector in sys.Connectors |
                Attached(c1,conn) and Attached(c2,conn);
Heuristic Size(Ports) <= 5;
```

L'invariant précise que tous les composants du système doivent être connectés entre eux deux à deux. La configuration forme un graphe complet. L'heuristique précise que le nombre de tous les ports doit être inférieur ou égal à cinq.

La troisième classe d'expertise de conception architecturale représente les styles architecturaux. Cet aspect sera abordé dans la sous-section suivante.

Les deux types de contraintes, décrites avec l'ADL WRIGHT et Armani, visent des modèles particuliers (des architectures bien définies). Elles ont donc comme contexte des éléments d'une architecture bien précise. Il existe un autre type de contraintes qui visent plutôt des méta-modèles. Elles s'appliquent donc à toutes les instances de ces méta-modèles (à toutes les architectures).

Dans [104], Medvidovic et al. proposent différentes manières de modéliser les architectures logicielles dans le langage de modélisation UML (version 1.5). Ils proposent, dans la première approche, d'utiliser UML tel quel. Cette approche permet aux développeurs d'utiliser les outils standards conformes à la spécification UML pour décrire des architectures. Par contre, la modélisation des architectures devient assez complexe, du fait qu'UML ne fournit pas toutes les possibilités d'expression des ADL (composants, connecteurs, etc). La seconde approche consiste à étendre UML de manière standard avec les stéréotypes, les valeurs marquées et les contraintes OCL. La dernière approche consiste à étendre le méta-modèle UML avec de nouveaux concepts (méta-classes) pour les composants, les ports, les connecteurs, etc. Cette nouvelle notation a pour inconvénient de ne pas être conforme aux spécifications UML (extension non standard du méta-modèle UML) et les outils actuels pour UML ne fonctionnent plus avec ce genre de descriptions d'architecture.

Dans la seconde approche, les auteurs utilisent le langage OCL pour contraindre au niveau méta le langage UML. Par exemple, les deux contraintes ci-dessous sont exprimées sur le méta-modèle UML.

```
context Class inv:
self.interface->size() = 2

context Association inv:
self.associationEnd->size() = 2
```

Le premier invariant permet de contraindre le méta-modèle UML à supporter les composants de l'ADL C2SADEL comme étant des classes qui ne peuvent définir que deux interfaces. Le deuxième prédicat stipule que les associations, représentant les liens entre composants et connecteurs dans l'ADL C2SADEL ne peuvent avoir que deux extrémités.

L'approche présentée dans ce mémoire, à l'opposé de l'approche ci-dessus, permet d'écrire des contraintes qui naviguent bien dans des méta-modèles, mais qui s'appliquent uniquement à une instance particulière de ces méta-modèles (une architecture bien précise). De plus, il est possible de décrire des contraintes de type évolution avec le langage de contraintes architecturales proposé dans cette thèse. Ces contraintes permettent de comparer deux versions différentes d'une même architecture. Ceci n'est pas possible avec les langages de contraintes cités ci-dessus.

Il existe une multitude d'autres langages de spécification de contraintes architecturales. Ces langages permettent la description de contraintes portant sur l'aspect comportemental. Comme cet aspect ne fait pas l'objet de la thèse, ces langages ne seront pas abordés dans ce mémoire. Par contre, il est important de noter que ces langages sont tous associés aux ADL. A ma connaissance, il n'existe aucun langage de contraintes architecturales pour les technologies de composants. Les seuls langages proposés dans la littérature pour les technologies de composants, comme [27], visent la spécification de contraintes fonctionnelles à la Eiffel [108].

3.3 Langages de formalisation des styles architecturaux

Il existe une multitude de langages permettant la formalisation des styles architecturaux. Ces langages varient des langages de description d'architecture classiques (ADL) aux langages fournis par des environnements dédiés à la description, à l'analyse et à l'exécution même de descriptions d'architectures guidées par les styles architecturaux. Ces langages de formalisation de styles architecturaux ont été classifiés dans ces deux catégories. Certains parmi eux sont abordés dans les deux sous-sections suivantes :

3.3.1 Langages de description d'architecture

Parmi les langages permettant la formalisation des styles architecturaux, les langages de description d'architecture. En effet, en plus de la description basique des architectures (composant, connecteurs et configurations) et la spécification des contraintes architecturales, ces langages permettent de décrire des styles architecturaux. Ils fournissent donc les constructions langagières nécessaires pour décrire des types de composants et de connecteurs, et des règles (contraintes) régissant leur configuration. Cette description est appelée style architectural. Un style répond à une exigence particulière et permet de résoudre un certain problème de conception. Cette solution est donc réutilisée par un architecte pour décrire l'architecture particulière d'un système afin de bénéficier des avantages de ce style.

Parmi les langages permettant également de définir des styles architecturaux, l'ADL WRIGHT. Si on prend l'exemple d'un style architectural donné, comme le *pipe & filter* [141], en utilisant WRIGHT, ce style peut être décrit de la manière suivante :

$Style\ Pipe-Filter$
$Component\ Filter$
$Port\ Input\ [description\ comportementale\ de\ ce\ port]$
$Port\ Output\ [description\ comportementale\ de\ ce\ port]$
$Computation\ [description\ comportementale\ de\ la\ relation\ entre\ les\ ports]$
$Connector\ Pipe$
$Role\ Source\ [description\ comportementale\ de\ ce\ role]$
$Role\ Sink\ [description\ comportementale\ de\ ce\ role]$
$Glue\ [description\ comportementale\ de\ la\ relation\ entre\ les\ roles]$
$Constraints$
$\forall\ c : Connectors \bullet Type(c) = Pipe$
$\wedge\ \forall\ c : Components \bullet Type(c) = Pipe$
$\wedge\ \forall\ c : Components \bullet \exists\ p : Ports(c);\ cn : Connectors;\ r : Roles(cn) \bullet$
$((c,p),(cn,r) \in Attachments \wedge Type(p) = Input \wedge Type(r) = Sink$
$\quad \vee\ Type(p) = Output \wedge Type(r) = Source$

Dans ce style, tous les composants sont de type *filter*. Chaque composant de ce type possède deux ports : un port d'entrée des données et un port de sortie. Tous les connecteurs dans ce style sont de type *pipe*. Chaque connecteur de ce type possède deux rôles : un rôle source et rôle puits. Le port d'entrée est relié au rôle puits et le port de sortie au rôle source.

Dans WRIGHT, les styles peuvent être spécialisés (raffinés) pour obtenir des sous-styles. Généralement, dans ces sous-styles, on ajoute des contraintes structurelles ou on spécialise les types des éléments. Le style ci-dessus peut être spécialisé pour obtenir par exemple le style *pipeline* [141] de la manière suivante :

$Style\ Pipeline : Pipe-Filter$
$Constraints$
$\sharp Components = \sharp Connectors + 1$
$\wedge\ \forall\ c : Components \bullet \exists\ p : Ports(c);\ cn : Connectors;\ r : Roles(cn) \bullet$
$((c,p),(cn,r) \in Attachments$
$\wedge\ \exists\ s : seq\ Components \mid ran\ s = Components \wedge \sharp\ s = \sharp\ Components$
$\bullet\ \forall\ cn : Connectors;\ i,j : \mathbb{N};\ p1 : Ports(s(i));\ p2 : Ports(s(j)) \mid$
$\{((s(i),p1),(cn,Sink)),((s(j),p2),(cn,Source))\} \subseteq Attachments \bullet$
$i = j - 1$

Dans cette description du sous-style, on hérite de tout le vocabulaire du style *pipe & filter* et on ajoute par conjonction toutes les contraintes. Les contraintes additionnelles qui sont le listing ci-dessus expriment le fait que dans un pipeline, le nombre de composants est égal au nombre de connecteurs plus un. Les autres invariants stipulent que les composants forment une séquence, et chaque couple de deux composants consécutifs est relié par un connecteur unique.

Armani permet également la formalisation des styles architecturaux. Ci-dessous, j'illustre l'exemple d'une contrainte Armani qui formalise le style architectural pipeline [141]. Comme

détaillé dans le chapitre 7, une architecture peut être perçue comme un graphe dont les noeuds représentent les composants et les arêtes représentent les connecteurs.

```
Style Pipe-and-Filter = {
// Definition du vocabulaire du style (les differents types
// utilises dans les invariants)
Port Type inputT = ... // Description comportementale du port
Port Type outputT = ... // Description comportementale du port
Role Type sourceT = ... // Description comportementale du role
Role Type sinkT = ... // Description comportementale du role

Component Type Filter = ... // Description du composant Filter
Connector Type Pipe = ... //  Description du connecteur Pipe
// Definition des contraintes
// La liste des contraintes est commentee ci-dessous
...
}
```

1. La première contrainte stipule la définition de noeuds et d'arêtes, et que chaque arête doit relier deux noeud :
   (a) un noeud représente un composant avec des ports en entrée (input) et des ports en sortie (output)

   ```
   invariant forall comp: Component
   in self.Components |
     forall p: Port in comp.Ports |
       satisfiesType(p,inputT)
       or satisfiesType(p,outputT)
   ```

   (b) Une arête représente un connecteur avec exactement deux rôles – un rôle puits (sink) et un rôle source – :

   ```
   invariant forall conn: Connector
   in self.Connectors |
     size(conn.Roles) == 2
     and exists r: Role in conn.Roles |
     satisfiesType(r,sinkT)
     and exists r: Role in conn.Roles |
     satisfiesType(r,sourceT)
   ```

   (c) Chaque connecteur (arête) relie deux composants (noeuds). Le port en entrée au rôle puits et le port en sortie au rôle source :

   ```
   invariant forall  conn: Connector
   in self.Connectors |
     forall r: Role in conn.Roles |
      exists comp: Component
      in self.Components |
        exists  p: Port in comp.Ports |
          attached(p,r)
          and ((satisfiesType(p,inputT)
          and satisfiesType(r,sinkT))
          or (satisfiesType(p,outputT)
          and satisfiesType(r,sourceT)))
   ```

2. La deuxième contrainte implique deux sous-contraintes :
   (a) Le graphe doit être connexe :

```
invariant forall c1, c2: Component
in self.Components |
  c1 != c2 -> reachable(c1,c2)
```

   (b) et, il contient n-1 arêtes, n étant le nombre de noeuds :

```
invariant size(self.Connectors)
== size(self.Components)-1
```

3. La dernière contrainte stipule que le graphe, représentant ainsi un arbre, doit être une liste. Ceci est exprimé comme suit :

```
invariant forall comp: Component
in self.Components |
  size(comp.Ports) == 2
  and exists p: Port in comp.Ports |
  satisfiesType(p,inputT)
  and exists p: Port in comp.Ports |
  satisfiesType(p,outputT)
```

Dans cette section, les langages de description d'architecture offrant des possibilités de formaliser des styles architecturaux ont été présentés. Ces ADL fournissent un certain nombre d'outil pour interpréter ces contraintes. Ils permettent de vérifier qu'une architecture donnée respecte un style donné.

3.3.2 Langages et environnements dédiés

L'un des premiers travaux sur la formalisation de styles est celui de Garlan et al. dans [53]. Les auteurs présentent un système appelé Aesop. Ce système permet de représenter graphiquement des architectures décrites selon un style particulier. L'environnement graphique permettant de représenter cette architecture fournit un certain nombre d'outils (d'édition ou d'analyse) propres au style en question. Derrière de telles descriptions, la formalisation des styles est représentée par une librairie de modules en langage C. Comparativement aux approches présentées ci-dessous où les descriptions des styles se font avec des langages de contraintes de haut niveau, cette approche se base sur du code. La définition des styles dans le système Aesop se base sur un mécanisme de typage. Les auteurs ont fourni un modèle composé d'abstractions architecturales (*Component*, *Connector*, *Port*, *Role*, *Representation* et *Binding*). Tous ces éléments forment des classes (types). Les sous-types de tous les éléments fournissent un vocabulaire des styles architecturaux. Un *Filter* est un sous-type de *Component*, un *Pipe* est un sous-type de *Connector*, etc. De plus, chaque style possède un certain nombre de règles de configuration (contraintes de style) définies sous la forme de méthodes. Aesop introduit également des mécanismes d'interprétation sémantique des styles et de vérification statique de l'architecture. La sémantique du vocabulaire associé à un style est définie de manière formelle et sert à réaliser des vérifications de types et autres vérifications statiques. Ce système incorpore également des outils de génération de code qui permettent de produire des programmes exécutables.

Dans [89], l'auteur présente un langage nommé ASL (*Architecture Style Language*) dédié pour la description des styles architecturaux. Ce langage permet également la description des architectures de systèmes dynamiques. Il se base sur deux autres langages ArchWare ADL et AAL. ArchWare ADL est un langage de description d'architecture basé sur le π-calcul [112]. En ArchWare ADL, les éléments architecturaux sont définis comme des comportements. Chacun est décrit par un ensemble d'actions ordonnancées qui spécifie à la fois le traitement interne de l'élément architectural et les interactions avec son environnement. Un élément architectural communique avec les autres via une interface appelée *connexion*. Il est lié aux autres à travers des liens appelés *unification*. ArchWare AAL permet de décrire des patrons comportementaux. Il s'agit de prédicats qui ont la valeur vraie si l'ordonnancement des actions d'un comportement donné vérifie certaines règles qu'on exprime. Ces règles expriment l'agencement possible entre des actions de communication. Un style architectural formalisé dans ASL est composé de trois concepts : les constructeurs (qui fournissent un support à la description architecturale), les contraintes (qui permettent de définir des règles structurelles et de comportement, et des prédicats sur les attributs qualité, liés à ce style), et les analyses (qui permettent d'étudier une série de propriétés du style sur l'architecture). Le constructeur et les contraintes du style architectural pipe & filter sont illustrés ci-dessous.

```
Pipe_Filter is style extending Component where {
  styles {
    InputPort is style extending Port ...
    OutputPort is style extending Port ...
    Filter is style extending Component ...
    Pipe is style extending Connector ...
    ...
  }

  constraints   {
    only_Pipes_or_Filters is constraint {
      to styleInstance.connectors apply
        forall(c | c in style Pipe ) and
          to styleInstance.components apply
            forall(c | c in style Filter )
    },
    pipes_cannot_be_connected is constraint {
      to styleInstance.connectors apply
        forall(p1,p2 | p1 in style Pipe and p2 in style Pipe
          implies not attached(p1,p2)),
    },
    one_to_one_port_connection is constraint {
      to styleInstance.connectors apply
        forall(f | to f.ports apply
          exists(fp | to styleInstance.connectors apply
            exists([0..1]p | to p.ports apply
              exists(pp | pp attached to fp)
            )
          )
        )
    }
  }
}
```

Dans cette définition de style, on retrouve les contraintes du style *pipe & filter*. Ces contraintes sont les suivantes : i) les éléments sont soit dans le style Pipe, soit dans le style Filter, ii) deux éléments dans le style Pipe ne peuvent pas être attachés, iii) un port ne peut être attaché qu'à un seul autre port, iv) un port dans le style InputPort ne peut être attaché qu'à un port de style OutputPort et inversement. Les définitions des éléments InputPort, OutputPort, Filter et Pipe sont purement comportementales et sont omises dans l'exemple ci-dessus.

Tous ces langages et environnements permettent de décrire des sortes de familles d'architectures et englobent des langages de contraintes pour décrire les contraintes des styles. Dans cette thèse, le langage proposé pour la description des décisions architecturales est un langage de contraintes basé sur la logique de premier ordre tout comme ces langages. Il permet la spécification de contraintes sur la structure d'une architecture donnée, alors que la description de l'architecture en question est faite à l'aide d'un ADL donné. Ce langage complète donc les ADL existants n'offrant pas cette possibilité avec des fonctionnalités de formalisation des contraintes des styles architecturaux. L'apport de ce langage par rapport aux ADL, fournissant déjà un langage de contraintes, sera discuté dans le prochain chapitre.

3.4 Langages de formalisation des patrons de conception

La spécification de patrons de conception est souvent faite à l'aide de langages textuels, de langages de modélisation UML ou des fragments de code. Il existe également un certain nombre de travaux qui discutent la spécification formelle de ces patrons à l'aide de différents types de logique. Ces approches n'ont pas été proposées pour remplacer les autres approches dites semi-formelles, mais plutôt pour les compléter. En effet, les approches de spécification de patrons de manière textuelle ou graphique contiennent souvent des ambiguïtés, ce qui rend leur compréhension et leur interprétation assez difficiles. Les approches formelles sont donc proposées pour résoudre ce problème. Dans cette section, nous discutons quelques uns de ces travaux.

Dans [85], les auteurs présentent une approche qui vise à modéliser de manière formelle les patrons de conception en proposant des extensions au méta-modèle UML[1]. Les auteurs utilisent le langage OCL pour décrire des contraintes (structurelles et comportementales) représentant deux exemples de patrons de conception (le visiteur et l'observateur [50]). Ces contraintes sont définies sur des méta-modèles d'éléments UML spécifiés sous la forme de méta diagrammes de collaboration. Des mécanismes d'association de ces diagrammes de niveau méta à leurs instances (des occurrences de patrons de conception) sont ensuite définis. Ceci permet de modéliser de manière précise des patrons de conception dans le langage UML.

Kim et al. présentent dans [79] une autre approche pour représenter les patrons de conception dans un langage appelé RBML (*Role-Based Metamodeling Language*). Ce langage permet de représenter des familles de modèles UML (à savoir des patrons de conception), sous la forme de méta-modèles UML. A ces méta-modèles sont associés des contraintes OCL pour une meilleure précision de ces représentations semi-formelles. Une SPS (Spécification de Patron Statique) définit des contraintes OCL sur un méta-modèle UML pour restreindre l'espace

[1] La version du langage UML prise en compte ici est la 1.3. Les auteurs discutent également les possibilités d'application de l'approche dans le méta-modèle UML 2.

de solutions qu'impose un patron sur l'aspect structurel. Les liens entre les modèles et les SPS sont représentés par des dépendances UML illustrant l'application du patron au modèle.

Dans ces travaux, les auteurs utilisent le langage OCL au niveau méta-modèle. Dans cette thèse, OCL est utilisé de la même manière, sauf que le contexte des contraintes n'est pas un élément du méta-modèle UML (de niveau M2) comme c'est le cas dans ce travail, mais un élément du modèle (niveau M1). L'objectif de la formalisation n'est pas le même dans les approches de ces deux travaux et ceux de cette thèse. L'objectif des auteurs de ces travaux derrière la formalisation des patrons de conception est de fournir un moyen aux développeurs UML de décrire de manière précise un patron. L'objectif de cette thèse est de fournir un langage pour documenter les décisions architecturales, à savoir le choix d'un patron de conception particulier. Cette documentation sert à préserver ce patron lors de l'évolution. Ceci permet de garantir les caractéristiques qualité implémentées par ce patron.

Taibi et Ling Ngo exposent dans [147] une autre approche pour la spécification formelle de patrons de conception, dite approche équilibrée (*balanced approach*). A travers cette approche, les auteurs proposent un langage, nommé BPSL (*Balanced Pattern Specification Langugage*). BPSL est un langage qui regroupe à la fois la spécification structurelle de patrons de conception, assurée par un sous-ensemble de la logique de premier ordre (FOL : *First Order Logic*) et la spécification comportementale, assurée par un sous-ensemble de la logique temporelle d'actions. Le sous-ensemble de FOL utilisé pour la partie structurelle utilise les éléments de base de cette logique, à savoir les quantificateurs, les connecteurs logiques et des symboles de prédicats agissant sur des symboles variables. Ces symboles variables représentent les classes, les attributs, les méthodes, les objets et les valeurs non-typées. Les auteurs ont défini un ensemble minimal regroupant les relations entre ces symboles. Parmi ces relations, on retrouve l'héritage, l'invocation, etc. Les auteurs détaillent aussi le sous-ensemble de la logique temporelle des actions utilisé pour la spécification de la partie comportementale (qui n'est pas intéressante dans le cadre de cette thèse). Ils présentent également la manière avec laquelle ils ont intégré ces deux formalismes.

Les variables symboles et les relations entre elles sont définies dans ce travail comme partie intégrante du langage de spécification des patrons (les deux approches qui le précèdent les représentaient sous la forme de méta-modèles UML). Comme précisé dans le prochain chapitre, ces variables et ces relations sont encapsulées dans des méta-modèles MOF. OCL, le langage choisi pour la formalisation des patrons de conception navigue dans ces méta-modèles afin de faire référence à ces variables. Le méta-modèle est un paramètre en entrée pour le langage OCL et peut être étendu sans impact particulier sur le support logiciel fourni avec (voir chapitre 8). En outre, l'objectif de ce travail est de rendre plus compréhensible les patrons de conception afin de pouvoir choisir le patron qui résout le mieux le problème rencontré par un développeur. Par contre, comme précisé précédemment, l'objectif de cette thèse est de formaliser les patrons afin de les préserver lors de l'évolution.

Il existe une multitude d'approches pour la description formelle des patrons de conception. L'objectif de toutes ces approches varient de l'application d'un patron [111] à la génération de code [1]. Certaines approches ont pour objectif plutôt de détecter des patrons [1] dans des documents de conception ou dans le code, alors que d'autres visent une meilleure compréhension des patrons afin de choisir le patron qui résout le mieux le problème de conception rencontré (approches ci-dessus). Certaines autres approches visent plutôt la composition de patrons et la

spécification de leur aspect temporel [110]. L'objectif de cette thèse est certes de pouvoir documenter une décision de conception concernant le choix d'un patron de conception particulier, mais le but recherché est surtout de pouvoir les rendre persistants lors de l'évolution.

3.5 Autres approches informelles

Dans la section 2.3.3, différentes approches pour documenter les architectures logicielles ont été présentées. En plus de documenter les concepts de base composant une architecture logicielle, certaines approches, comme celle dans [25], requièrent les documentations des décisions architecturales et les raisons derrière ces décisions (*architecture design rationale*). Dans [148], Les auteurs présentent un état de l'art de l'utilisation et de la documentation de ces deux aspects. Ils ont fait une étude expérimentale et les chiffres fournis montrent que les décisions architecturales et leurs raisons constituent un aspect important dans la conception logicielle et concluent que ce domaine de recherche mérite encore plus d'investigation. De nouvelles techniques et supports doivent être fournis pour mieux documenter et utiliser ces décisions et les raisons derrière ces décisions.

Clements et al. présentent, dans [25], une approche qui fournit un cadre pour la documentation des différentes vues d'une architecture logicielle (discutées dans la section 2.3.3) de manière concise. Elle propose de définir cette documentation selon les sept principes suivants :

1. "Écrire du point de vue de l'utilisateur de cette documentation" : Tout d'abord, il faudra déterminer qui sont les utilisateurs de cette documentation (développeurs, clients, personnes s'occupant de la maintenance, etc). Ensuite, il est important de définir ce que ces utilisateurs veulent savoir. Enfin, il est appréciable de rendre les informations faciles à trouver.
2. "Éviter les répétitions inutiles" : Les informations doivent être stockées dans un seul endroit. Ceci rend la documentation facile à utiliser et à modifier. La répétition est souvent source de confusion surtout s'il s'agit d'information répétée de manière légèrement différente.
3. "Éviter l'ambiguïté" : La documentation est un moyen pour communiquer des informations. La précision de ces informations est une caractéristique importante, car elle permet une meilleure compréhension de cette documentation.
4. "Utiliser une organisation standard" : Cette organisation standard permet à l'utilisateur de retrouver facilement l'information recherchée. Elle permet également au rédacteur de cette documentation d'ajouter ou de modifier facilement cette documentation.
5. "Enregistrer les décisions architecturales et les raisons derrière ces décisions" : Il faudra noter des informations concernant les décisions architecturales et exposer les raisons qui ont poussé le développeur à prendre ces décisions et non pas d'autres. Ceci permettra de se rappeler ultérieurement des informations précieuses lors de la maintenance, par exemple.
6. "Maintenir la documentation à jour, mais pas trop à jour" : Une documentation à jour fournit des informations réelles sur le système développé ou en cours de développement.

Cette documentation ne doit pas être mise à jour très fréquemment au point de devenir assez coûteuse ou qu'elle crée une frustration chez ses utilisateurs.

7. "Revoir la documentation" : Il faudra revoir la documentation avec ses utilisateurs afin que les informations qu'elle contient soient toujours utiles.

En prenant en compte ces principes de documentation, les auteurs de ce livre ont proposé un *template* qui sert à documenter une vue particulière d'une architecture. Ce *template* est composé : i) d'une présentation initiale de la vue (graphique ou textuelle). Cette présentation illustre les éléments de la vue et les relations entre eux ; ii) d'un catalogue (dictionnaire) des éléments cités dans la première section ; iii) d'un diagramme de contexte qui contient des informations sur l'environnement du système et leurs relations avec le système ; iv) d'un guide de variabilité qui explicite les éléments du système qui varient en respectant certaines propriétés ; et v) d'une base architecturale ; vi) d'autres informations relatives à la version de cette documentation et à ses changements dans le temps. La section la plus intéressante dans le cadre de cette thèse est l'avant dernière section de la documentation. Elle peut contenir plusieurs autres éléments, à savoir un sommaire des besoins significatifs qui affectent cette vue, les décisions architecturales faites dans cette architecture et les raisons de ces décisions (éventuellement des alternatives à ces décisions), les résultats d'analyse validant ces décisions, et des suppositions sur l'environnement.

Nous remarquons que dans cette documentation, la cinquième section contient une description des décisions architecturales et les raisons de ces décisions. Cette clause de la documentation est décrite de manière informelle (textuelle), parce que les utilisateurs de cette documentation visés par les auteurs ne sont pas des outils. Dans cette thèse, les décisions architecturales et les raisons de ces décisions sont documentées de manière formelle dès lors que l'utilisateur de cette documentation, comme nous le verrons dans le prochain chapitre, est un algorithme d'assistance à l'évolution. Ces décisions sont vérifiées automatiquement lors de l'évolution.

Tyree et Akerman, dans [155], discutent l'importance de la documentation des décisions architecturales, ainsi que leur spécification comme entités de première classe dans les descriptions d'architecture. Ils présentent, tout comme l'approche précédente, un *template*, mais uniquement consacré pour la description des décisions durant le développement. Ce template est organisé en sections qui sont présentées ci-dessous.

- Issue : Dans cette section, le développeur décrit l'issue de conception architecturale qu'il adresse.
- Décision : Ici, la décision est clairement citée. Elle explicite la position prise par le développeur pour répondre à l'issue.
- Statut : Le statut de la décision est défini dans cette section. Les différent statuts possibles sont : suspendue, décidée ou approuvée.
- Groupe : Le groupe auquel appartient cette décision est défini ici. Les décisions peuvent être organisées en groupe selon une classification proposée initialement par les développeurs (intégration, présentation, données).
- Suppositions : Dans cette section, les suppositions faites sur l'environnement où la décision a été prise sont définies. Par exemple, les développeurs peuvent citer dans cette section des suppositions sur les technologies utilisées dans le développement.

- Contraintes : Ici, les développeurs peuvent expliciter toute contrainte additionnelle liée à l'environnement où la décision a été prise. Par exemple, ils peuvent spécifier une contrainte sur le coût du projet.
- Positions : Les différentes autres alternatives sont citées et expliquées dans cette section.
- Argument : Ici, les développeurs peuvent spécifier les raisons de la décision prise.
- Implications : Dans cette section, les implications de la décision sont explicitées. Des décisions impliquent parfois le changement de la spécification des besoins (ajout, modification, suppression). Elles peuvent également introduire le besoin de prendre d'autres nouvelles décisions.
- Décisions relatives : Ici, les développeurs peuvent spécifier, le cas échéant, les décisions liées à cette décision. Ceci peut être fait de manière graphique ou textuelle.
- Besoins relatifs : Comme précisé dans le chapitre 1, les décisions architecturales ne sont pas innocentes. La plupart du temps, elles ont comme origines des besoins non-fonctionnels. Ces besoins sont cités ou référencés ici.
- Artefacts relatifs : Dans cette section, les auteurs proposent que les documents de conception par exemple soient cités.
- Principes relatifs : Si l'entreprise par exemple possède une liste de principes convenus, ces derniers doivent être respectés par la décision. Ceux qui sont affectés peuvent être présentés ici.
- Notes : Dans cette dernière section, toute information complémentaire peut être ajoutée.

Les sections de ce *template* les plus intéressantes dans le cadre de cette thèse sont les sections de la décision et de l'argument. Il n'y a aucune contrainte pour que ces deux éléments soient définis de manière formelle dans cette documentation. Dans le contexte de cette thèse, ces deux éléments sont les briques de base d'un contrat d'évolution. Ils sont décrits de manière formelle, comme précisé précédemment. Ils sont donc évalués et permettent d'automatiser certaines vérifications. D'autres sections dans la documentation ci-dessus, comme les besoins relatifs et les artefacts relatifs, ne sont pas de moindre importance. En effet, comme vous pourrez le remarquer plus tard dans ce mémoire, le premier élément est considéré dans les contrats d'évolution. Cet élément constitue une information importante lors de l'évolution. Le deuxième élément est le concept auquel est rattaché un contrat d'évolution. Dans le cadre de cette thèse, les artefacts considérés sont les descriptions d'architecture et les configurations de composants.

Par ailleurs, ces travaux se focalisent sur l'aspect méthodologique de la description de ces décisions. Ils ne montrent pas l'usage de ces décisions lors de l'évolution. Le mot décision reste assez abstrait. Il n'est pas considéré comme une contrainte implémentant une décision concrète faite sur l'architecture, comme c'est le cas dans cette thèse. En effet, c'est la vue composant & connecteur qui est traitée dans ce mémoire. Comme vous pourrez le constater, les décisions architecturales illustrées dans les chapitres 6 et 7 sont de niveau conception et de niveaux en aval, et sont directement projetées sur cette vue architecturale. Tyree et Akerman parlent de décisions architecturales de niveau spécification, comme : l'extension d'un système B au delà de ces fonctionnalités originales par l'implémentation d'un traitement interactif pour les produits financiers qu'il gère [155]. Cette décision est d'un niveau d'abstraction plus élevé que les décisions architecturales prises en compte dans cette thèse.

Certains auteurs parlent de documentation des suppositions de conception (*Design Assumptions*). Dans [83], Lago et van Vliet présentent une approche pour rendre explicite ces suppo-

sitions afin d'enrichir les modèles d'architecture. Les auteurs définissent les suppositions de conception comme les raisons derrière les décisions prises à la volée pendant la conception (grâce à l'expérience des développeurs par exemple), et qui sont souvent implicites. Ce sont des invariabilités qui doivent être explicitées afin qu'elles puissent être utilisées pour mieux comprendre une architecture lors de l'évolution. Les auteurs proposent un moyen pour documenter les suppositions et les lier aux éléments architecturaux affectés par ces suppositions. Ils fournissent plusieurs types de liens représentés sous la forme de relations F-impacts pour un impact fonctionnel d'une supposition à un élément fonctionnel (*functional feature*), et des relations S-impacts pour un impact structurel d'une supposition à un élément structurel (composant, interface, service, etc). D'autres relations de type Realizes sont définies dans le sens inverse (entre les éléments fonctionnels et structurels et les suppositions), et des relations de type Is-a sont définis entre suppositions. Ces dernières permettent d'établir une hiérarchisation des suppositions en différentes catégories.

Dans ce travail, les auteurs traitent à la fois l'aspect fonctionnel et l'aspect structurel d'une architecture. Ils lient donc à deux modèles différents (représentant ces deux aspects) les suppositions de conception. Dans cette thèse, on ne traite que l'aspect structurel d'une architecture. Toutes les décisions architecturales sur cet aspect sont donc documentées dans des contrats. En outre, ce travail introduit une hiérarchisation des suppositions, alors que, dans cette thèse, les raisons des décisions restent atomiques. Par contre, l'approche présentée dans ce travail propose de documenter seulement les raisons derrière les décisions et les lier aux éléments architecturaux, alors que, dans ce mémoire, on s'intéresse également à la documentation des décisions.

3.6 En résumé

Dans ce chapitre, j'ai présenté différentes approches pour la documentation des décisions architecturales. Ces approches peuvent être formelles ou informelles. Les approches formelles visent à automatiser certains traitements, et les approches informelles proposent des documentations pour une meilleure compréhension d'un logiciel. Parmi les traitements possibles permis par les approches formelles, on retrouve l'application de patrons de conception, l'extraction de styles architecturaux et de patrons de conception du code, l'automatisation de la génération de code, etc. Dans cette thèse, l'approche proposée pour la documentation des décisions architecturales est formelle. Elle vise à préserver ces décisions lors de l'évolution. Parmi les décisions architecturales adressées dans ce mémoire, on retrouve le choix d'un style architectural ou un patron de conception particulier, l'explicitation d'invariants architecturaux et la spécification de contraintes d'évolution (celles qui impliquent des comparaisons de versions d'architectures). De plus, à la différence des approches existantes, le langage proposé, dans ce travail, pour la documentation de ces décisions est basé sur des standards bien connus (OCL et MOF). Ces standards sont utilisés de manière originale et assez simple pour documenter tous ces choix de conception.

Chapitre 4

Documentation des propriétés non-fonctionnelles

Sommaire

4.1 Introduction à la documentation des propriétés NF

Depuis une dizaine d'années, la modélisation des besoins ou exigences non-fonctionnelles[1] (*NFRs : Non-Functional Requirements*) a fait l'objet de plusieurs travaux de recherche. Ce domaine reste d'actualité de nos jours et les auteurs, contribuant à cette discipline, affirment que des investigations approfondies sont nécessaires. En effet, le domaine des besoins fonctionnels est bien maîtrisé car il est le plus simple à cerner, alors que le domaine des besoins non-fonctionnels reste assez varié et ses différents aspects sont assez difficiles à regrouper au sein d'un même formalisme [117]. Ces NFRs s'intéressent à des aspects assez hétérogènes, à un haut niveau d'abstraction, et qui sont d'ordre extra-fonctionnel[2]. Ces propriétés non-fonctionnelles sont adressées par des besoins non-fonctionnels. Ces NFRs limitent leur domaine de valeurs (par des valeurs ou des intervalles de valeurs de métriques) ou précisent leur degré de satisfaction (faible, moyen ou élevé). Par exemple, un NFR peut aborder la propriété non-fonctionnelle de performance en stipulant que le temps de réponse d'un service X rendu par un composant Y doit être inférieur à 10 mili-secondes. Dans un autre exemple, une propriété non-fonctionnelle comme la maintenabilité peut être adressée par un besoin non-fonctionnel de

[1] Les deux termes (besoins et exigences) sont confondus dans la littérature. Dans ce mémoire, c'est le terme besoins non-fonctionnels qui sera adopté.

[2] Certains auteurs comme dans [33] préfèrent utiliser le terme propriété extra-fonctionnelle pour désigner toute propriété qui ne concerne pas une fonctionnalité offerte ou requise par un logiciel.

la manière suivante : les composants X et Y doivent être facilement maintenables, car ils souvent sujets à des modifications. Dans un troisième exemple de NFR, on peut s'intéresser à une propriété non-fonctionnelle comme le langage de programmation utilisé pendant le codage, ou le serveur d'application dans lequel sont déployés les composants. Il peut s'agir d'un besoin non-fonctionnel exigeant l'usage du langage de programmation C++, ou du serveur d'application J2EE WebSphere d'IBM[3].

Nous remarquons que, parmi les propriétés non-fonctionnelles, on retrouve les attributs qualité. Il est important de noter qu'en ce moment il n'y a pas de consensus sur le fait que les attributs qualité sont tous d'ordre non-fonctionnel [11]. Prenons l'exemple de l'attribut qualité sécurité. Supposons qu'on voudrait ajouter à un composant donné des capacités d'authentification des utilisateurs afin de garantir un besoin dans les documents de spécification adressant cet attribut. L'ajout de l'authentification à ce composant est considéré comme ajout d'une nouvelle fonctionnalité. En effet, ce service vient s'ajouter à la liste des services fonctionnels garantis par ce composant. Malgré cela, la plupart des travaux, dans la littérature, considèrent ces attributs comme caractéristiques qui adressent l'aspect non-fonctionnel d'un logiciel.

Il existe, dans la littérature et la pratique, plusieurs approches pour la documentation des propriétés non-fonctionnelles. D'un point de vue langagier, ces travaux peuvent être classifiées dans deux catégories. La première catégorie regroupe les approches formelles et la seconde les approches semi-formelles et informelles (purement textuelles). L'objectif des approches formelles est d'automatiser certains traitements sur des informations extraites des besoins non-fonctionnels, comme l'assistance à la sélection de composants COTS (*Commercial Off-The-Shelf*) [5]. Quant aux approches semi-formelles ou informelles, leur but est de proposer des méthodes pour raffiner les besoins et les adjoindre aux besoins fonctionnels, jusqu'à obtention de documents de conception et d'architecture [10]. Une autre manière de classifier les différentes approches de documentation des propriétés non-fonctionnelles consiste en des approches dites "centrées produits" et d'autres "orientées processus". Les approches centrées produits s'intéressent à la description des besoins non-fonctionnels afin de pouvoir les mesurer ou les observer sur le produit logiciel après l'avoir développé. Les approches orientées processus proposent plutôt des moyens pour représenter les besoins non-fonctionnels et des méthodes de développement logiciel orientées par ces représentations de NFR. La plupart des approches formelles de descriptions des NFRs se placent dans la première catégorie (centrée produits), alors que les approches semi-formelles et informelles se placent dans la seconde (orientée processus). Ces deux approches (centrée produits et orientée processus) se complètent et permettent ensemble de représenter et d'utiliser les besoins non-fonctionnels [116].

4.2 Approches centrées produits

Xavier Franch a proposé NoFun un langage formel pour la spécification des propriétés non-fonctionnelles de logiciels [45]. Le recours à un langage formel a plusieurs avantages. Il permet de spécifier de manière homogène, aussi bien, les propriétés non-fonctionnelles que les propriétés fonctionnelles. Il permet également, lors de la maintenance, d'automatiser la consultation des propriétés non-fonctionnelles lors de l'application de changements au système. Lors

[3] Site Web de WebSphere d'IBM : www.ibm.com/websphere

de la sélection d'une implémentation donnée (composant COTS) pour un document de conception, il est facile d'automatiser le choix de la meilleure implémentation (celle qui correspond le mieux aux propriétés non-fonctionnelles). Enfin, la réutilisation est, tout comme la sélection, facile à optimiser. L'auteur introduit trois concepts représentant les liens entre des propriétés non-fonctionnelles et composants d'implémentation. Ces concepts sont :

- les attributs non-fonctionnels (*NF-attribute*) : Ces attributs ont la même sémantique que les facteurs ou caractéristiques qualité ISO 9126 (maintenabilité, réutilisabilité, etc.).
- les comportements non-fonctionnels (*NF-behaviour*) d'implémentations de composants : Ceux-ci représentent des affectations de valeurs aux attributs non-fonctionnels. Elles représentent les valeurs qui sont utilisées dans l'implémentation composant courante.
- les besoins non-fonctionnels (*NF-requirements*) : Ils représentent une contrainte impliquant un sous ensemble des attributs non-fonctionnels qui sont utilisés par un composant.

Les attributs non-fonctionnels possèdent un certain nombre de caractéristiques. Chaque attribut appartient à un domaine. Ce domaine fixe l'ensemble des valeurs et opérations valides, comme les types Boolean, Integer, Enumeration, etc. Les attributs sont classifiés, selon que leurs valeurs peuvent être évaluées à partir d'autres attributs ou non, en deux genres : dérivé ou basique, respectivement. Chaque attribut possède également une portée. Elle précise les composants où cet attribut est utilisé. Ils peuvent être utilisés dans des composants individuels, dans des librairies de composants réutilisables, dans des systèmes logiciels (assemblage de composants), ou dans des *clusters* (ensemble de systèmes). Ces attributs sont attachés par des liens (*mapping*) à des composants ou à des opérations. En outre, les attributs peuvent avoir des définitions multiples. En effet, l'auteur affirme que comme il n'y a pas de consensus dans la littérature par rapport à la définition de chaque attribut, il est possible de donner plusieurs définitions à un même attribut. Par contre, pour un attribut possédant plusieurs définitions, une seule doit être choisie dans la portée où cet attribut va être utilisé. Franch a proposé de définir des modules pour les attributs qui sont associés à des spécifications de composants. A leurs différentes implémentations, il associe, comme précisé ci-dessus, des comportements non-fonctionnels (*NF-behaviour*). A chacune des caractéristiques d'un attribut est affectée une valeur. Les composants déclarent leurs besoins en terme de propriétés non-fonctionnelles, pour les composants avec lesquels ils vont être assemblés, à travers les *NF-requirements*. L'exemple d'une spécification de besoins non-fonctionnels est donnée ci-dessous :

```
behaviour module for IMP_LIBRARY_1
behaviour
...
Cette partie contient les differents attributs et leurs valeurs
dans l'implementation du composant IMP_LIBRARY_1
...
requirements on SET: max(reliability)
                     time(put,remove) = 1
                     min(time(intersect))
            on LIST: implemented with
                       ORDERED LIST
end IMP_LIBRARY_1
```

Supposons un composant LIBRARY qui utilise deux autres composants LIST et SET pour composer des listes et des ensembles de livres, de membres, etc. Dans l'exemple ci-dessus,

l'implémentation IMP_LIBRARY_1 définit un besoin non-fonctionnel sur SET comme suit : l'implémentation doit être la plus fiable possible ; ensuite, le coût des insertions et des retraits d'éléments doit être constant ; enfin, l'intersection d'ensemble doit être la plus rapide possible. Pour le composant LIST, on exige une implémentation particulière sous la forme de liste ordonnée.

NoFun est un langage permettant de décrire de manière précise les propriétés non-fonctionnelles de composants logiciels. La principale utilisation de ce langage fournie par l'auteur est pour la sélection parmi des composants existants celui qui possède la meilleure implémentation [45]. Par meilleure implémentation, on veut dire l'implémentation qui s'approche le plus des besoins non-fonctionnels (*NF-requirements*) définis dans la spécification d'un composant. Je reviendrai sur l'utilisation de ce langage lors de la maintenance dans le chapitre 5.

Bass et al. [11] utilisent les scénarios pour caractériser les différents attributs qualité d'un système (attributs du premier niveau, dans leur modèle qualité présenté dans le chapitre précédent). Un scénario est une spécification générique d'un attribut qualité précis. Une spécification générique désigne ici une spécification construite avec le même ensemble de concepts pour tous les attributs qualité, malgré leur diversité. En effet, pour l'attribut qualité "performance", on parle souvent "d'événements". Pour l'attribut "sécurité", on parle "d'attaques". On se réfère aux "pannes", lorsqu'on aborde l'attribut qualité "disponibilité". On utilise le terme "entrée utilisateur" dans le domaine de l'attribut qualité "utilisabilité". Dans un scénario, on utilise le terme générique "stimulus". Un scénario est la spécification d'un besoin d'une propriété non-fonctionnelle liée à la qualité. Il est composé de six éléments :

- la source du stimulus : Cet élément représente l'entité (humain, un module logiciel, ou autre) qui a généré le stimulus ;
- le stimulus : Il représente la condition qui nécessite une considération lorsqu'elle arrive au système ;
- l'environnement : Ceci englobe les conditions sous lesquelles le stimulus apparaît ;
- l'artefact : Cet élément représente la partie du système ou le système entier stimulé ;
- la réponse : La réponse est l'activité entreprise après l'arrivée du stimulus.
- la mesure de la réponse : Quand la réponse se produit, elle doit être mesurable de telle sorte que le besoin non-fonctionnel puisse être testé.

Dans cette approche, il existe deux types de scénarios. Le premier concerne des scénarios dits généraux, qui sont indépendants d'un système particulier. Ils peuvent donc concerner n'importe quel système. Les scénarios du second type sont dits concrets. Ils concernent un système particulier en cours de développement. Les attributs qualité dans cette approche sont caractérisés par des scénarios généraux. Afin de traduire une caractérisation d'un attribut en un besoin non-fonctionnel, les scénarios généraux relatifs doivent être rendus spécifiques au système.

L'exemple d'une caractérisation de l'attribut qualité maintenabilité (appelé modifiabilité, dans le modèle qualité du SEI utilisé dans ces travaux) est donné ci-dessous. Ceci représente donc le scénario général de l'attribut qualité maintenabilité.

- la source du stimulus : Utilisateur final, développeur ou administrateur système ;
- le stimulus : Volonté d'ajouter ou de modifier les fonctionnalités ou les attributs qualité ;
- l'environnement : Au moment de l'exécution, de la compilation, de la construction, ou de la conception ;

- l'artefact : Interface utilisateur du système, la plate-forme, l'environnement (le système qui interopère avec le système cible) ;
- la réponse : Localiser les endroits dans l'architecture à changer, faire la modification sans affecter les autres fonctionnalités, tester la modification, ou déployer la modification ;
- la mesure de la réponse : Le coût en terme de nombre d'éléments affectés, l'effort, ou l'argent.

L'exemple d'un scénario concret de maintenabilité peut être le suivant : "un développeur souhaite changer l'interface utilisateur pour rendre le fond de l'écran bleu". Dans ce cas, la source est le développeur. Le stimulus est représenté par la volonté de changer l'interface utilisateur. L'environnement est le moment de conception. L'artefact représente le code. La réponse à ce stimulus est représentée par le fait que la modification est faite sans effet de bord. La mesure de la réponse est 3 heures.

Les auteurs fournissent un catalogue des scénarios généraux des différents attributs qualité de leur modèle. Ils présentent également un certain nombre d'exemples de scénarios concrets. Tous ces scénarios sont utilisés dans des méthodes d'évaluation des architectures logicielles [26]. Ces méthodes visent à vérifier si une architecture conçue est bien conforme aux spécifications de qualité initialement établies. En effet, une architecture logicielle correspond bien aux premiers documents de conception dans lesquels on voit apparaître les attributs qualité. Il est donc intéressant, si ce n'est pas indispensable, d'évaluer cette architecture avant d'avancer dans le processus de développement vers l'implémentation. Parmi ces méthodes, on retrouve SAAM (*Software Architecture Analysis Method*) [76], ATAM (*Architecture Tradeoff Analysis Method*) [75], ARID (*Active Reviews for Intermediate Designs*) [26] et CBAM (*Cost Benefit Analysis Method*) [114]. ATAM permet d'étudier, en plus de l'analyse basique des architecture (SAAM), les compromis entre décisions architecturales pour garantir plusieurs attributs qualité. ARID permet d'évaluer des architectures partielles (non encore finies) pendant le développement. CBAM s'adresse à l'aspect économique de l'évaluation des architectures.

Dans l'approche de Franch, un *NF-attribute* a la même sémantique qu'un scénario général dans l'approche de Bass et al. En effet, cela représente une caractérisation générique d'un attribut qualité. Par ailleurs, le *NF-behaviour* et le *NF-requirement* représentent un scénario concret. En effet, ces deux concepts sont attachés à un composant (système) particulier et les valeurs des différentes caractéristiques de l'attribut sont affectées. La première approche est formelle et est basée sur des prédicats. La seconde approche est plus fine car elle détaille plus finement les besoins non-fonctionnels. Par contre, elle est moins formelle et est basée sur les scénarios.

Plusieurs autres travaux, classés dans cette catégorie d'approches de documentation de propriétés non-fonctionnelles centrées produits, existent dans la littérature. Ces travaux couvrent des domaines assez variés, comme la qualité de service des logiciels distribués [48], les aspects transactionnels des intergiciels [161] ou les propriétés de qualité des systèmes temps réel [80]. Tous ces travaux fournissent des notations pour décrire les propriétés non-fonctionnelles spécifiques à ces domaines. Ils proposent ensuite des méthodes et des outils pour l'évaluation de ces documents non-fonctionnels sur des documents de conception ou du code. Ils ne peuvent être détaillés pour des raisons d'homogénéité entre les différentes sections de ce mémoire.

4.3 Approches orientées processus

Parmi les travaux les plus significatifs dans ce domaine, on retrouve [116] et [29]. Mylopoulos, Chung et Nixon dans [116] présentent un framework pour la description et l'utilisation des propriétés non-fonctionnelles, basé sur les buts (*goals*). Ce framework se base sur les cinq concepts suivants :

- un ensemble de buts représentant les besoins non-fonctionnels, les décisions de conceptions, et les arguments pour ou contre d'autres buts ;
- un ensemble de types de liens pour lier les buts ;
- un ensemble de méthodes génériques pour raffiner les buts en d'autres buts ;
- une collection de règles de corrélation pour l'inférence d'interactions potentieles entre buts ;
- une procédure d'étiquetage qui détermine le degré avec lequel un besoin non-fonctionnel est adressé par un ensemble de décisions de conception.

Il existe dans ce framework trois types de buts : les buts de besoins non-fonctionnels (*NFR-Goals*), les buts de satisfaction[4] (*SatGoals*) représentant les décisions de conception, et les buts d'argumentation (*ArgGoals*) ou arguments représentant les raisons des décisions. Chaque but possède une sorte associée (par exemple, *Performance* ou *Reliability*), et zéro ou plusieurs paramètres. Ces sortes peuvent être raffinées en d'autres sortes (par exemple, *Space Performance* et *Time Performance*). Les buts d'argumentation ont toujours une sorte de type *Claim*, *Formal Claim* ou *Informal Claim* selon que la description de l'argument est formelle ou non. Cette approche propose de concevoir un système donné en raffinant les différents types de buts. Les liens sont définis entre un but donné et ses buts fils. Ils sont définis également entre un lien reliant deux buts et un but d'argumentation, pour indiquer le support négatif ou positif du raffinement. Une proposition est un couple composé de buts et de liens. Les auteurs ont défini plusieurs types de liens. Ces types permettent de distinguer les propositions qui peuvent satisfaire (ou potentiellement satisfaire) un but des autres propositions, des buts qui contribuent partiellement à la satisfaction d'autres buts, etc. Il est à la charge du concepteur de raffiner les buts et de s'assurer de la satisfaction des buts et de ces raffinements. Le framework fournit un certain nombre de méthodes basiques de raffinement de buts. Chaque raffinement est représenté par un lien. Ces méthodes permettent d'obtenir plusieurs buts fils à partir d'un but donné. Elles sont de trois types : méthodes de décomposition, méthodes de satisfaction et méthodes d'argumentation. Les méthodes de décomposition permettent de passer d'un haut niveau d'abstraction à un bas niveau d'abstraction pour les buts représentant les besoins non-fonctionnels (*NFRGoals*). Les méthodes de satisfaction permettent de définir un lien entre un but *NFRGoal* et un but *SatGoal* représentant une décision de conception. Celles-ci permettent donc d'avancer dans la conception du système. Les méthodes d'argumentation permettent de formaliser le lien entre des buts *SatGoals* et des justifications de ces derniers (*ArgGoals*). Les règles de corrélation permettent de formaliser les conflits entre les buts (performance vs. sécurité, par exemple). La procédure d'étiquetage permet de labeliser les noeud ou les arcs du graphe de buts partiellement construit précédemment. Elle permet de décider quel noeud ou arc est satisfait ou pas,

[4] Le mot satisfaction désigne ici "garantie à un certain degré". En effet, les propriétés non-fonctionnelles ne sont, généralement, pas satisfaites complètement, mais plutôt satisfaites à un certain degré. En anglais, ceci est désigné par le mot *to satisfice* et non *to satisfy*

conflictuel ou indéterminé.

En construisant ce type de graphes étiquetés de buts, il est possible de concevoir un système en prenant en considération toutes les propriétés non-fonctionnelles requises dans les spécifications. Le développeur devient capable de faire les décisions de conception qui justifient le mieux la satisfaction des besoins non-fonctionnels. Cette approche permet également de choisir, parmi les décisions, celles qui ne mettent pas en conflit les besoins non-fonctionnels. Les propriétés non-fonctionnelles discutées dans ces travaux sont liés au domaine des systèmes d'information (*accuracy* et *performance*).

Les besoins non-fonctionnels sont adressés de manière légèrement différente par Cysneiros et Sampaio do Prado Leite [29]. En fait, cette approche se base sur l'ancienne approche de Mylopoulos et al. et l'étend en prenant en compte des langages standards de modélisation, à savoir UML. Les auteurs proposent de définir les besoins non-fonctionnels de la même manière que l'approche précédente. Ces descriptions sont, ensuite, intégrées dans des diagrammes de cas d'utilisation, de classes, de séquence et de collaboration.

Il existe, dans la littérature, un certain nombre d'autres travaux de recherche visant à proposer des méthodes de conception guidées par les besoins qualité, mais spécifiques aux architectures à base de composants. Dans [10], Len Bass et al. présentent une méthode de conception qui, à partir de la spécification des besoins sur les attributs qualité, permet d'obtenir des architectures décrites selon certains styles architecturaux. Les besoins non-fonctionnels sont décrits sous forme de scénarios (présentés ci-dessus). Les besoins fonctionnels sont décrits sous forme de cas d'utilisation. Une primitive d'attribut est une collection de composants et de connecteurs qui collaborent pour garantir un attribut qualité. Certains styles architecturaux sont considérés comme primitives d'attributs (ou de conception). Ces primitives guident la conception de l'architecture. Cette méthode de conception appelée ADD (*Attribute-Drive Design*) ressemble beaucoup à celle de Mylopoulos, mais elle n'est pas formelle. Elle est basée sur la décomposition récursive et le raffinement. Dans chaque étape de décomposition, les primitives d'attributs sont choisies pour satisfaire un ensemble de scénarios qualité. Ensuite, les fonctionnalités sont affectées pour instancier les types de composants et de connecteurs fournis par les primitives. Les étapes lors de chaque décomposition sont les suivantes :

1. Choisir les pilotes architecturaux parmi l'ensemble des scénarios de qualité et les besoins fonctionnels. Un pilote architectural est une combinaison de buts fonctionnels et de qualité qui forment l'architecture ou l'élément architectural considéré.
2. Choisir les primitives d'attribut et les types de composants fils pour satisfaire les pilotes architecturaux. Cette étape est conçue pour satisfaire les besoins de qualité.
3. Instancier les éléments de conception et allouer les fonctionnalités des cas d'utilisation. Cette étape permet de satisfaire les besoins fonctionnels. Les pilotes architecturaux de qualité permettent de choisir un style architectural particulier, par exemple. Par la suite, les pilotes architecturaux fonctionnels permettent de déterminer les instances des composants et connecteurs définis par le style.
4. Vérifier et raffiner les cas d'utilisation et les scénarios de qualité et les rendre comme contraintes pour les éléments fils. Cette étape vérifie que rien d'important n'a été oublié. Elle prépare les éléments fils pour plus de décomposition ou à l'implémentation.

Les étapes précédentes sont appliquées pour chaque élément de conception (le système en entier au début). Toutes les entrées requises de ces étapes doivent être disponibles (contraintes, besoins fonctionnels et besoins non-fonctionnels). Ces étapes sont répétées pour chaque élément de conception qui nécessite plus de décomposition.

Par ailleurs, dans d'autres travaux de ces mêmes auteurs [11], les tactiques architecturales sont introduites. Ces tactiques permettent de garantir un attribut qualité en particulier lors de la conception d'une architecture. Les auteurs présentent un catalogue des différentes tactiques architecturales utilisées pour garantir les attributs qualité de disponibilité, de maintenabilité (appelée modifibilité dans leur modèle), de performance, de sécurité, de testabilité et d'utilisabilité. L'encapsulation de l'information, l'utilisation d'intermédiaires ou la restriction des chemins de communication sont des exemples de tactiques pour l'attribut qualité maintenabilité. Plusieurs des tactiques d'architecture sont souvent implémentées par un même style architectural. Les concepteurs utilisent généralement directement ces styles comme primitives d'attributs. Par contre, ces tactiques peuvent être combinées de différentes manières pour obtenir des styles différents, ou pour spécialiser un style particulier.

Des versions plus évoluées des styles architecturaux classiques (introduits dans le chapitre précédents) ont été proposé par Klein et al. [81]. Ils sont appelés ABAS (*Attribute-Based Architecture Styles*). Ces styles sont basés sur les attributs qualité et sont utilisés comme décisions architecturales lors de la conception. Un ABAS est le triplet composé :

- de la topologie des types de composants et la description du patron d'interaction de données et de contrôle entre ces composant (définition standard d'un style),
- d'un modèle d'un attribut qualité spécifique qui fournit une méthode de raisonnement sur le comportement des types de composants qui interagissent dans le patron défini,
- et du raisonnement qui résulte de l'application du modèle spécifique à l'attribut aux types des composants qui interagissent.

Les auteurs de ces styles affirment que ces derniers peuvent être utilisés comme décisions architecturales documentées dans un processus de conception afin de définir des architectures conformes à des besoins en attributs qualité. Mais leur travaux se focalisent plutôt sur l'aspect analyse d'une architecture par rapport à un modèle de qualité, et n'adressent pas en détail la conception architecturale. Ils exploitent les travaux de Bass et al. pour représenter les attributs qualité sous la forme de scénarios. Ils ont proposé principalement six ABAS : l'ABAS client/serveur de synchronisation pour l'attribut qualité performance, les ABAS d'indirection de données (dépositaire de données, dépositaire de données abstraites, et le *publish/subscribe*) pour la maintenabilité, l'ABAS en couches pour la maintenabilité également, et l'ABAS de redondance (le simplex) pour la disponibilité. Il fournissent aussi des méthodes de composition de plusieurs ABAS pour adresser plusieurs attributs qualité simultanément et sans conflits.

Dans cette thèse, l'objectif n'est pas de montrer comment, à partir des besoins non-fonctionnels, on peut obtenir une documentation des propriétés non-fonctionnelles (objet de l'état de l'art de la première sous-section de cette section). La supposition est faite qu'on utilise l'une des méthodes existantes répondant à cet objectif. Les contrats d'évolution sont définis ensuite à partir des résultats obtenus par ces méthodes. L'objectif n'est pas, non plus, de proposer des méthodes de conception qui permettent d'obtenir une architecture conforme à cette spécification, à partir d'une spécification de besoins non-fonctionnels. Ceci a été présenté afin d'établir un état de l'art des différentes approches de documentation des propriétés non-

fonctionnelles. Ceci justifie également l'hypothèse de travail principale de cette thèse. Celle-ci stipule que les décisions architecturales sont déterminées par les attributs qualité requis dans les spécifications.

4.4 En résumé

Les contrats d'évolution permettent de documenter les liens entre les spécifications non-fonctionnelles et les décisions de conception implémentant ces spécifications. Les décisions architecturales sont formalisées avec un langage dédié offrant plusieurs possibilités d'expression. Les propriétés non-fonctionnelles sont documentées de manière simple se basant sur le modèle de qualité ISO 9126. Le chapitre suivant détaille ces deux aspects.

Le seul travail permettant de formaliser ce genre de documentations (liens propriétés non-fonctionnelles et décisions architecturales) est celui de Mylopoulos et al. En effet, les auteurs proposent de lier des propriétés non-fonctionnelles et des décisions exprimées sous forme de buts. Ces deux aspects sont représentés par des graphes. Cette sorte de documentation est utilisée pour accompagner l'activité de conception. Elle permet de garantir l'obtention de documents de conception "conformes" aux besoins non-fonctionnels. L'approche présentée dans ce mémoire utilise approximativement le même concept : lier des propriétés non-fonctionnelles (*NFRGoal*), à des décisions de conception (*SatGoal*) et aussi le *rationale* de ces décisions (*ArgGoal*). A la différence des travaux de Mylopoulos et al., cette thèse se focalise sur l'aspect architectural de logiciels à base de composants. De plus, cette documentation est utilisée pour contrôler l'activité d'évolution d'un logiciel et non pas sa conception.

Chapitre 5

Contrôle de l'évolution des logiciels

Sommaire

5.1 Introduction au contrôle de l'évolution

La plupart des travaux applicatifs dans le domaine de l'évolution du logiciel se focalisent sur la préparation d'un logiciel à l'évolution. En d'autres termes, ces travaux proposent des techniques et des outils pour que l'évolution soit faite sur un logiciel facilement maintenable. D'autres travaux s'intéressent plutôt à la correction du logiciel après évolution, vis à vis des spécifications des besoins (techniques de tests de non-régression [137] ou méthodes d'évaluation des architectures [26]), ou bien vis à vis de la maintenabilité (techniques de *refactoring* [107], par exemple). Ces deux catégories de travaux se distinguent donc selon le moment où le contrôle est effectué (avant l'évolution ou après l'application des changements). Le travail présenté dans cette thèse fournit une méthode de contrôle de l'évolution à la volée. En d'autres termes, le contrôle est effectué lors de l'application des changements. Il se situe donc en aval des travaux de la première catégorie, et en amont des travaux de la seconde.

Peu de travaux adressent le contrôle de l'évolution d'un logiciel au moment de l'application des changements. Dans l'état de l'art présenté ci-dessous, ces travaux sont classifiés en deux catégories. Ces deux catégories représentent les travaux répondant au critère portant sur l'objectif du contrôle. Deux objectifs se distinguent :

- Contrôle pour le maintien de la cohérence d'un logiciel, ou le maintien de la cohérence d'un ensemble de documents formant un même logiciel,
- Contrôle pour la préservation de propriétés non-fonctionnelles (attributs qualité) en général.

La deuxième approche peut englober la première si l'on considère que la cohérence est une propriété non-fonctionnelle. J'ai cependant jugé, utile, dans ce mémoire, de séparer les deux, du moment où la première constitue un domaine de recherche à part entière. La seconde

englobe les travaux adressant la spécification des propriétés non-fonctionnelles en général et les attributs qualité en particulier. Généralement, les auteurs de ces mêmes travaux s'orientent vers l'exploitation de ces derniers dans l'évolution et la maintenance.

5.2 Contrôle pour le maintien de la cohérence

Une évolution d'un logiciel est dite cohérente si, après application des changements, le logiciel ne contient pas des parties ou des documents entiers qui ne sont pas homogènes avec les parties ou les documents qui ont évolués. Ces derniers n'évoluent pas avec le reste du logiciel. En d'autres termes, ces travaux étudient l'impact des changements, la détection des incohérences et la propagation des changements. Parmi les travaux les plus récents et les plus significatifs dans cette catégorie, ceux sur les contrats de réutilisation (*Reuse Contracts* [144]). Ces contrats permettent l'encapsulation de certaines informations utiles pour la propagation des changements lors de l'évolution d'un modèle objet. Ils se limitent à l'étude des classes abstraites et de l'héritage. Une étude plus avancée, fournissant d'autres supports, est fournie dans la thèse de Tom Mens [105]. Ces contrats forment une documentation d'interfaces, soit un ensemble de méthodes. Ils sont implémentés par des classes du modèle. Chaque définition de méthode dans un contrat contient :

- un nom unique,
- une annotation *abstract* ou *concrete* pour désigner respectivement une méthode abstraite ou une méthode concrète,
- une éventuelle clause de spécialisation. Dans cette clause, on définit toutes les invocations de cette méthode à d'autres méthodes dans le même contrat. Dans ces contrats, à la différence des interfaces de spécialisation [84], seules les invocations de niveau conception sont définies. Les invocations de niveau implémentation ne sont pas prises en compte.

Les auteurs ont identifié six types d'évolutions. Ces évolutions sont représentées par des opérateurs d'évolution. En réalité, il existe trois opérateurs : concrétisation (rendre concrètes des méthodes abstraites), raffinement (surcharger des méthodes concrètes) et extension (ajouter de nouvelles méthodes). Les trois autres sont l'inverse des premiers : abstraction, grossissement et annulation. Un modificateur de réutilisation est un contrat représentant un opérateur d'évolution entre un contrat de réutilisation et son évolution.

Dans le listing ci-dessous, on retrouve un exemple d'un contrat de réutilisation. Ce contrat s'appelle *View* et décrit une classe abstraite contenant une méthode abstraite *Draw* et une méthode concrète *Update*. Cette dernière utilise la première méthode.

```
Reuse Contract View
  Abstract
    Draw
  Concrete
    Update      {Draw}
End Reuse Contract
```

Le contrat *DragableView* illustré ci-dessous représente une extension du contrat *View*. L'extension est représentée simplement par les mots clés *is an extension of*.

```
Reuse Contract DragbleView
is an extension of View
  Abstract
    Draw
  Concrete
    Update      {Draw}
    Drag        {Draw}
End Reuse Contract
```

Ce contrat ajoute la méthode concrète *Drag* qui utilise la méthode abstraite *Draw*. De la même manière, il est possible de décrire des contrats représentant les autres opérateurs d'évolution.

Dans une hiérarchie de classes, les contrats associés à des classes parentes (super-classes) sont dits contrats de base. Les contrats associés aux classes fils (sous-classes) sont appelés contrats dérivés. Les auteurs investiguent le changement d'une classe père par une nouvelle classe. Ils proposent d'établir un changement des contrats de réutilisation. Les modificateurs et les contrats sont utilisés ensemble pour détecter de manière automatique les incohérences lors de l'évolution. Ces incohérences sont déduites de l'impact de l'évolution d'une classe père sur les classes fils.

Mens et D'Hondt présentent une approche pour l'utilisation des contrats de réutilisation dans des modèles UML [106]. Ils proposent un certain nombre de mécanismes pour automatiser l'évolution, afin de détecter les conflits évoqués ci-dessus. Ces contrats sont nommés contrats d'évolution. Un contrat d'évolution est un accord entre le fournisseur d'un artefact et celui qui le fait évoluer. Il permet de spécifier la modification représentant l'évolution d'un élément de modélisation vers un autre élément de modélisation. Afin d'identifier les différents types de modifications, les auteurs définissent plusieurs types de contrats. Un type de contrat correspond au modificateur dans les contrats de réutilisation. Il existe quatre types de contrats : ajout, suppression, connexion et déconnexion.

Les extensions UML fournies pour le support des contrats d'évolution sont les suivantes :

- Un contrat d'évolution est une spécialisation de la méta-classe *Dependency*,
- Un contrat d'évolution est défini entre des éléments de la méta-classe *NameSpace*,
- Un contrat d'évolution doit être stéréotypé pour indiquer le type du contrat,
- Le rôle de la modification d'un contrat d'évolution est une séquence non vide d'éléments (*ModelElements*) qui vont être modifiés,
- La sémantique exacte d'une modification est spécifiée à l'aide d'expressions OCL.

Le stéréotype d'un contrat d'évolution correspond au type de contrat. Il définit comment un élément va être modifié. En plus des types cités ci-dessus, les auteurs proposent deux autres types (composites) permettant de composer des contrats. Il s'agit de promotion et séquentialisation. Le premier permet de définir un contrat d'évolution de haut niveau comme un ensemble de contrats de bas niveau. Le second permet de définir un contrat comme une séquence d'autres contrats.

Tout comme les contrats de réutilisation, ces contrats sont utilisés pour détecter les incohérences entre les éléments dans un même document de conception lors de l'évolution. Ces incohérences portent principalement sur les problèmes d'héritage entre classes dans un modèle UML. Malgré le fait qu'ils partagent le même nom, les contrats d'évolution présentés dans ce mémoire sont conceptuellement différents des contrats d'évolution de Mens et D'Hondt. Ils se

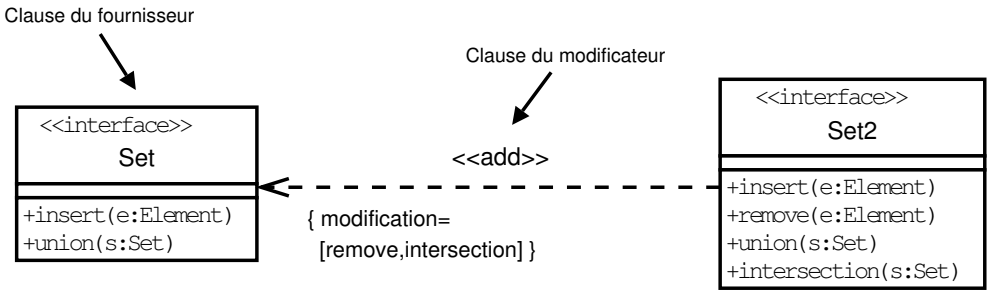

FIG. 5.1 – Contrat d'évolution dans l'approche de Mens et D'Hondt

distinguent sur les aspect suivants :

- L'objectif : comme précisé précédemment l'objectif des contrats d'évolution de Mens et D'Hondt est de détecter les conflits entre éléments de modélisation dans un modèle objet. L'objectif des contrats d'évolution introduits dans ce mémoire est de documenter les décisions de conception et de les lier avec leur raisons (*rationale*). Ceci a pour objectif d'assister (de contrôler) l'évolution dans le cas où une modification affecte une décision architecturale et, par conséquent, altère son *rationale*. L'objectif est d'un niveau d'abstraction plus haut. Le grain d'abstraction des documents de conception considéré n'est pas le même. Ils traitent dans leur travail des modèles objets, alors que dans cette thèse, les documents de conception pris en compte sont de niveau architectural (de niveau composant).
- L'organisation : Les deux types de contrats ne sont pas organisés de la même manière. Dans [106], les contrats contiennent deux clauses, une clause définie par le fournisseur des artefacts et une clause définie par celui qui réalise l'évolution (le modificateur). La clause du fournisseur contient le document de conception proprement dit (par exemple, une classe dans un modèle objet). Ceci est illustré dans la figure 5.1. La clause du modificateur contient la manière dont les éléments dans ce document évoluent (voir Figure 5.1). Le contrat d'évolution, dans cette thèse, ne contient pas l'évolution en soi des documents de conception (les changement à opérer), mais plutôt des informations qui doivent être préservées lors de l'évolution. Il encapsule des liens (*bindings*) entre les décisions de conception et les besoins qui ont conduits à ces décisions.
- Le langage : Les langages utilisés sont complètement différents. Les contrats de Mens et D'Hondt sont spécifiés dans un profil particulier d'UML. Les contrats proposés dans cette thèse sont définis à l'aide d'un langage fournissant un mécanisme de liens (*binding*). Ce langage se base, entre autres, sur OCL et MOF.
- Le mode d'utilisation : Le contrat d'évolution de Mens et D'Hondt est utilisé pour automatiser la détection des conflits relatifs à des problème de cohérence du document de conception. Dans ce mémoire, le contrat d'évolution est utilisé dans un mécanisme de notification automatique. Il détecte lors de l'évolution les décisions de conception qui ont été violées. Il notifie l'impact de cette violation sur les besoins non-fonctionnels dans les spécifications.

Dans les travaux d'Andreas Rausch, une autre approche est proposée. A la différence des approches précédentes qui se concentrent sur l'évolution de hiérarchies de classes, elle a pour objectif d'assurer une évolution cohérente de logiciels à base de composants [132]. Cette approche propose un modèle formel de composants supportant le concept de contrat de besoins/assurances (*Requirements/Asssurances Contracts*). Dans ce modèle, un composant doit spécifier ces besoins (*Requirements*) de manière explicite. Il s'agit des fonctionnalités requises par le composant de son environnement. Les assurances doivent aussi être spécifiées. Ce sont les propriétés que le composant assure à son environnement, en supposant que ces besoins sont satisfaits. Une fois les composants spécifiés, un développeur peut définir les dépendances comportementales entre les composants. Ces dépendances sont décrites en précisant pour chaque composant quelles sont les assurances qui garantissent ses besoins. Ces dépendances explicites sont appelées contrats de besoins/assurances (*r/a-contracts*). Les besoins et les assurances dans ce modèle sont modélisés sous la forme de types de fonctions. Les besoins ou les assurances d'un composant particulier sont définis à l'aide d'une fonction qui calcule pour un type de composant donné l'ensemble des prédicats (représentant les propriétés requises ou assurées). Un contrat est un *mapping* entre les propriétés requises d'un composant et les propriétés assurées (fournies) d'autres composants. Il existe aussi un prédicat *fulfilled* qui permet de vérifier si toutes les propriétés requises d'un composant sont fournies par les propriétés d'autres composants. Lors d'une évolution, des outils permettent de vérifier si ce prédicat est vérifié ou non. La présentation du formalisme utilisé pour décrire ce genre de contrats et les composants sous-jacents demande beaucoup d'explications. C'est pour cette raison que des exemples de contrats ne seront pas présentés ici.

Alors que les contrats de réutilisation permettent de spécifier des dépendances intra-interfaces, les contrats de besoins/assurances permettent de spécifier des dépendances inter-interfaces. Ces interfaces sont définies pour des composants distincts. A chaque pas d'évolution, ceci permet de déduire l'impact d'un changement d'un élément de conception sur le reste du système. Les éléments de conception dans les contrats de réutilisation sont les classes et les interfaces, alors que, dans les contrats de besoins/assurances, ce sont les composants. Les dépendances entre éléments de conception traités dans les contrats de réutilisation sont les liens d'héritage, alors que, dans les contrats de besoins/assurances, il s'agit de dépendances de type connecteurs architecturaux.

Bien que les contrats de besoins/assurances prennent en compte des aspects liés à des composants logiciels, la différence avec les contrats d'évolution, présentés dans cette thèse, reste fondamentale. Les contrats de besoins/assurances ont pour objectif d'étudier l'impact d'un changement d'un composant sur les autres composants, à travers les dépendances entre interfaces qui sont spécifiées explicitement dans ces contrats. Les contrats d'évolution introduits dans cette thèse ont pour objectif d'étudier l'impact d'un changement sur les décisions architecturales (comme les styles architecturaux auxquels l'architecture se conforme). Ceci permet de déduire l'impact sur les attributs qualité.

Au lieu de ne s'intéresser qu'aux documents d'un seul niveau d'abstraction (des modèles UML, par exemple), il existe des travaux qui portent sur le maintien de la cohérence entre les documents de différents niveaux d'abstraction (conception et implémentation) [133]. Ces travaux répondent à la problématique de logiciels multi-dimensionnels, composés de plusieurs artefacts qui évoluent indépendamment les uns des autres. Les incohérences apparaissent ici

entre les artefacts. Il n'existe pas de formalisme commun pour modéliser tous les artefacts. Par conséquent, les incohérences sont assez difficiles à gérer. Ces travaux proposent un framework commun (non une unique notation commune) où les documents de conception sont considérés comme des contraintes sur les documents d'implémentation. Par exemple, l'existence d'un diagramme de classes UML permet de générer une contrainte imposant l'existence, dans le code, des classes, de tous les attributs et méthodes qui apparaissent dans le diagramme. Ces contraintes ont la forme suivante : $\forall(x \in S)\varphi(x)\Theta(x)$, où : S représente l'ensemble des informations qui constituent la source de la contrainte (exemple, diagramme de classe UML), $\varphi(x)$ est une restriction sur S (exemple, les classes qui ont un nom non nul et qui sont des classes (et non des interfaces)) et $\Theta(x)$ les conditions de validité de la contrainte (exemple, dans les sources, il doit exister des classes avec les mêmes noms). Cette méta-contrainte est implantée dans une syntaxe XML. Un gestionnaire de contraintes génère des requêtes SQL correspondant au contenu de la contrainte à destination d'une base de données d'abstractions des différents artefacts. Les contraintes sont vérifiées à chaque fois qu'un artefact est modifié.

Ces travaux traitent la cohérence d'une évolution inter-dimensions (inter-artefacts), alors que, dans le chapitre 6 de cette thèse, on se préoccupe plutôt d'évolutions intra-dimensions (évolution d'une architecture ou d'une configuration de composants). On souhaite, à travers l'utilisation des contrats d'évolution, préserver certains choix de conception lors de l'évolution d'un même artefact. En d'autres termes, on voudrait préserver ces choix entre l'ancienne version d'un artefact et sa nouvelle version.

Dans le chapitre 7, une approche pour la traçabilité des choix de conception sera présentée. A la différence de la traçabilité présentée dans les travaux ci-dessus et réalisée lors de l'évolution, dans l'approche présentée dans cette thèse, la traçabilité est réalisée lors du développement (avant l'évolution).

5.3 Contrôle pour le maintien d'attributs qualité

Comme une extension de l'approche présentée dans la section 4.2 sur la formalisation de spécifications de besoins non-fonctionnelles, Franch et al. ont développé une approche pour le support des besoins non-fonctionnels lors de l'évolution [46]. Comme discuté précédemment, l'approche présentée permet le choix de la meilleure implémentation parmi plusieurs implémentations d'une même spécification de composants. Par meilleure implémentation, les auteurs veulent dire l'implémentation qui correspond le plus aux besoins. En d'autres termes, l'implémentation dont les valeurs pour les différentes propriétés non-fonctionnelles sont les plus proches des exigences. A chaque évolution des besoins non-fonctionnels (plus de fiabilité ou de performance, par exemple), ce même algorithme de sélection est rejoué pour déterminer la meilleure implémentation. En effet, les modules spécifiant les besoins non-fonctionnels attachés à un composant, sont modifiés pour prendre en compte les nouveaux besoins. L'algorithme de sélection prend en entrée toutes les implémentations existantes pour ce composant. Il sélectionne celle qui se conforme le plus aux nouvelles spécifications.

Les composants sont perçus de manière différente dans l'approche ci-dessus et dans ce mémoire. Franch et Botella considèrent les composants comme des types abstraits de données, au sens donné dans [70]. Un composant possède une définition explicitant le nom du type

et ses opérations. Il possède également plusieurs implémentations, utilisant les structures de données classiques, dont les propriétés non-fonctionnelles sont connues. Dans ce mémoire, un composant est considéré comme un élément architectural sujet à une composition, avec des interfaces requises et fournies explicites (définition fournie dans la section 2.3.7). Ces travaux ne discutent que l'évolution des besoins non-fonctionnels, alors que dans cette thèse, l'étude est faite sur l'évolution architecturale. Cette évolution peut être la conséquence d'une évolution de besoins non-fonctionnels ou fonctionnels. En outre, leur approche propose parmi plusieurs alternatives d'implémentation, la sélection de celle qui se conforme le plus à une évolution des besoins. Dans ce mémoire, l'approche proposée vise plutôt une assistance à l'évolution : lors de l'application des changements architecturaux, on notifie leur impact sur les décisions de conception et sur les propriétés non-fonctionnelles.

Une autre approche pour le contrôle de l'évolution vis-à-vis des attributs qualité a été présentée dans [94]. Cette approche fournit un mécanisme de vérification de politiques d'évolution. Une politique d'évolution est spécifiée comme un ensemble de prédicats dans la logique de premier ordre qui sont vérifiés sur un modèle. Ce modèle peut représenter un produit ou un processus. Comme exemple de politique d'évolution pouvant être spécifiée dans cette approche, on peut trouver le prédicat suivant : la valeur estimée de la somme des lignes de code ajoutées pour tous les composants ne doit pas dépasser la croissance moyenne, plus un pourcentage d'erreur (par exemple 10 %). Ce genre de politiques est évalué sur l'architecture d'une application à base de composants, enrichie d'informations sur l'historique de son évolution (la valeur estimée des lignes de codes ajoutées, par exemple). L'approche présentée dans ces travaux propose aussi un *framework* de raisonnement (contextuel), qui permet d'établir à partir des résultats retournés par le mécanisme de vérification des politiques d'évolution, un retour (*feedback*). Ce *feedback* permet aux développeurs d'améliorer les modèles en question.

Comme dans cette thèse, cette approche propose une méthode pour l'assistance à l'évolution. Le *feedback* du *framework* contextuel constitue le rapport de notification de l'algorithme d'assistance présenté dans le chapitre 6. Une politique d'évolution constitue approximativement une clause du contrat d'évolution. Par contre, la différence fondamentale entre ces deux travaux, réside dans le fait que, dans ce mémoire, on traite la vue microscopique de l'évolution (voir section 2.2.3), alors que l'approche présentée ci-dessus traite plutôt la vue macroscopique de l'évolution. En effet, la politique présentée dans le paragraphe précédent porte sur un historique de l'évolution de tous les composants d'un système. Les clauses du contrat d'évolution, comme exposé dans le chapitre 6, portent plutôt sur l'évolution d'un élément particulier d'une version à une autre.

Dans ses travaux sur le contrôle de l'évolution, Massimiliano Di Penta introduit un framework appelé *Evolution Doctor* [35]. Ce framework vise, lors de l'évolution, à détecter et diagnostiquer certaines caractéristiques du logiciel afin de lancer certaines activités de refactoring. Lors de la phase de diagnostic ce framework génère des graphes de dépendances entre objets du système. Ces graphes servent par la suite à calculer un certain nombre de métriques, à détecter les clones (interfaces, opérations, etc), et à vérifier la présence ou non de dépendances circulaires. Ces différentes informations sont utilisées, par la suite, pour lancer des opérations de refactoring. Ceci est effectué après chaque évolution. Une approche similaire, se basant sur certaines métriques pour des modèles objets, et sur la détection de patrons de conception est présentée dans [61].

Ces deux approches visent des modèle objets, alors que l'approche présentée dans cette thèse vise des application à base de composants. Elles se concentrent sur la détection d'anomalies dans le code par rapport à l'attribut qualité de maintenabilité, principalement. Cet attribut est considéré préalablement requis par tout logiciel, dès lors que chaque logiciel évolue (première loi de Lehman). Il doit donc être facilement maintenable. La détection est faite à l'aide d'un catalogue prédéfini de métriques liées à cet attribut. Ensuite, des actions de réparation des anomalies sont lancées. Dans cette thèse, l'approche proposée se limite à l'assistance de l'évolution. En d'autres termes, elle notifie simplement les anomalies. Par contre, elle couvre bien plus d'attributs qualité que la maintenabilité. Elle s'intéresse exclusivement aux attributs énoncés dans les spécifications des besoins non-fonctionnels. Ensuite, libre au développeur d'utiliser les techniques nécessaires pour réparer ces anomalies.

Dans la littérature, on retrouve un état de l'art sur les techniques utilisées pour "diagnostiquer" et rechercher la dégénérescence dans les architectures logicielles, et les méthodes employées pour traiter cette dégénérescence [63]. Comme précisé dans la section 1.3.2, la dégénérescence d'une architecture est la déviation de l'architecture actuelle par rapport à l'architecture planifiée. La majorité des technologies présentées dans ce travail se focalise sur la reconnaissance des styles architecturaux et des patrons de conception dans le code. Ceci aide à identifier les déviations en comparant les architectures et leur propriétés avant et après l'évolution. Les auteurs discutent également les techniques de visualisation des changements architecturaux. Ces techniques permettent de comprendre l'évolution architecturale et donc de déduire les portions dégénérées de l'architecture évoluée. Pour traiter cette dégénérescence, les auteurs présentent un état de l'art des techniques de refactoring.

5.4 En résumé

L'approche proposée dans cette thèse pour contrôler l'évolution d'un logiciel est basée sur des contrats. Il existe, dans la littérature, plusieurs approches qui utilisent de tels concepts. Ces approches adressent l'aspect de maintien de la cohérence d'un logiciel après son évolution. Elles garantissent donc que les changements appliqués sont bien répercutés sur les différentes parties d'un même artefact ou sur plusieurs artefacts en même temps. La solution proposée dans ce mémoire vise la préservation des propriétés non-fonctionnelles à travers le maintien des décisions architecturales les garantissant, définies de manière formelle. Cette manière originale d'utiliser les contrats permet de notifier un développeur des conséquences de ces changements sur la qualité initialement requise. Ces notifications servent comme un mécanisme d'alerte lors de l'évolution. Le développeur est libre de maintenir ses changements ou les annuler. L'évolution est donc contrôlée de manière souple, mais le développeur effectue ses modifications tout en connaissant les conséquences de ces dernières.

Dans cette thèse, une documentation est présentée. Elle représente une description des décisions architecturales, et leurs liens avec les propriétés non-fonctionnelles qu'elles garantissent. Cette documentation permet, grâce à un algorithme d'assistance, de contrôler l'évolution d'un logiciel. Il existe, dans la littérature, différentes approches pour la documentation des décisions architecturales et des propriétés non-fonctionnelles. Parmi les approches pour documenter les décisions architecturales, on trouve des langages de contraintes architecturales, des langages de formalisation des styles architecturaux et des patrons de conception. En plus de ces notations formelles, il existe des approches informelles de documentation. Leur unique objectif est de faciliter la compréhension d'un logiciel, car elles ne permettent pas, du fait de leur aspect informel, d'automatiser certaines vérifications, nécessaires à l'assistance à l'évolution. Dans cette thèse, un langage est proposé pour la documentation des décisions architecturales. Ce langage introduit une notation formelle. Ceci permet d'automatiser la vérification de la préservation ou non des décisions architecturales lors de l'évolution.

Parmi les approches pour documenter les propriétés non-fonctionnelles, il existe également des approches formelles et des approches informelles. La documentation des propriétés non fonctionnelles est basique et informelle dans ce mémoire. Il s'agit simplement d'informations, extraites des spécifications des besoins et conformes à un certain modèle (ISO 9126). Ces informations sont utilisées lors de l'assistance pour notifier les développeurs de l'impact des changements sur la qualité.

Seule l'approche de Myloupolous réunit les deux aspects de documentation (décisions de conception et propriétés non-fonctionnelles), tout comme les contrats d'évolution. Par contre, à l'opposé de ces contrats, cette approche ne s'intéresse pas à l'évolution. Elle permet simplement de construire un système conforme aux spécifications des besoins non-fonctionnels. De plus, elle ne prend pas en compte les aspects architecturaux de logiciels à base de composants.

Dans la littérature, on trouve également plusieurs approches pour contrôler l'évolution d'un logiciel. De nombreux travaux se focalisent sur la préparation d'un logiciel à l'évolution. En d'autres termes, des solutions sont proposées pour renforcer l'attribut qualité de maintenabilité. Peu de travaux s'intéressent à la réaction à la volée (à chaud) à une évolution pour mieux la contrôler. Parmi ces travaux, il existe ceux qui se basent sur des contrats, comme c'est fait dans cette thèse. Mens et D'Hondt introduisent, dans leurs travaux sur le contrôle de l'évolution d'un logiciel pour le maintien de sa cohérence, le concept de contrat d'évolution. Ces contrats ont fondamentalement des objectifs différents de ceux des contrats d'évolution introduits dans cette thèse. Les contrats d'évolution de Mens et D'Hondt permettent de détecter les conflits (principalement les problèmes liés à l'héritage) dans des hiérarchies de classes, lors de l'évolution. Dans ce mémoire, les contrats servent à contrôler l'évolution afin de préserver des décisions architecturales et des attributs qualité, du type maintenabilité, portabilité et autres.

Il n'existe pas, dans la littérature, d'approches pour l'assistance à l'évolution avec des informations d'ordre non-fonctionnel. Ceci constitue l'apport fondamental de cette thèse. Cette assistance vise à diagnostiquer les anomalies dans un logiciel après évolution. Les anomalies ici représentent les altérations des décisions architecturales et les attributs qualité correspondants. L'assistance, dans cette thèse, ne vise pas à corriger ces anomalies mais plutôt à notifier au développeur les conséquences des changements qu'il effectue sur l'architecture et sur les besoins non-fonctionnels. Cette assistance permet de guider le développeur vers une évolution du logiciel sans déviation importante des spécifications des besoins non-fonctionnels désirés. Libre au développeur d'utiliser les solutions qu'il juge les plus adéquates pour réparer ces anomalies, ou plus simplement de renoncer aux changements qu'il a effectués.

Troisième partie

Contribution de la thèse

Afin de répondre aux problématiques énoncées dans le chapitre 1, deux approches ont été développées. La première approche se base sur le concept de contrat d'évolution. Un contrat d'évolution est un outil pour le maintien de propriétés non-fonctionnelles lors de l'évolution. Ceci permet d'anticiper certaines vérifications pendant l'évolution. Ces vérifications visent à réduire le nombre d'itérations entre les étapes de tests de non-régression et d'application des changements, discutés précédemment. Cette approche répond à la problématique 2 illustrée dans le chapitre 1. Elle sera détaillé dans le premier chapitre de cette partie.

Ces contrats englobent dans leur définition une formalisation des décisions architecturales. Ces décisions sont décrites sous la forme de contraintes. Cet aspect sera raffiné dans le second chapitre. L'approche permettant de tracer ces contraintes (et donc les décisions) tout au long du processus de développement sera présentée dans ce chapitre. Cette solution contribue à la résolution de la problématique 1 introduite dans le chapitre 1.

Dans le dernier chapitre, je présente l'architecture et le fonctionnement des deux outils prototypes développés pour implémenter ces deux solutions. Je commencerai d'abord par présenter AURES, un environnement pour l'assistance à l'évolution. Ensuite, l'outil ACE est détaillé. Il sert à tracer des contraintes architecturales entre les étapes du processus de développement d'un logiciel à base de composants.

Chapitre 6

Assistance à l'évolution architecturale dirigée par la qualité

Sommaire

6.1 Introduction au besoin d'assister l'évolution

L'évolution de l'architecture d'un logiciel peut avoir un objectif fonctionnel ou non-fonctionnel. L'évolution fonctionnelle vise à ajouter, modifier ou retirer des fonctionnalités du logiciel. L'évolution non-fonctionnelle a pour but, entre autres, d'améliorer la qualité du logiciel par ajout de nouveaux attributs qualité ou renforcement de certains attributs existants.

Considérons une évolution d'un logiciel sur le plan fonctionnel à partir d'une version donnée (représentée par le point à gauche de la Figure 6.1). L'axe en pointillés représente une

évolution fonctionnelle avec strict respect de la spécification non-fonctionnelle initiale. Cet axe est dit théorique, car, selon les discussions du chapitre 1, en évoluant, un logiciel perd certaines de ses propriétés non-fonctionnelles. L'évolution déviera donc vers un état à une distance Δ de l'axe théorique.

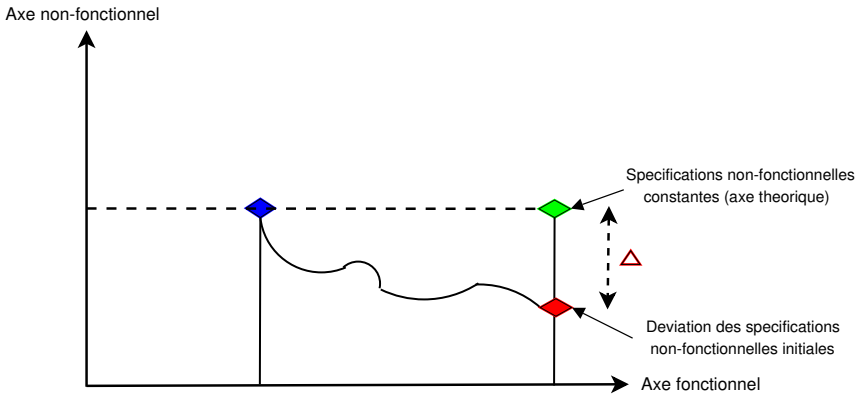

FIG. 6.1 – Évolution avec déviation des spécifications non-fonctionnelles initiales

Les tests de non régression effectués a posteriori permettent de réduire ce Δ, mais avec un coût qui n'est pas négligeable. Dans ce chapitre, je présente une approche qui aide à diminuer les coûts associés aux étapes de compréhension et de vérification de la non régression pour des applications conçues à l'aide de composants. Une approche qui, non seulement, promeut la présence d'une documentation non ambiguë et à jour, mais qui offre également un cadre de vérification de la non régression a priori selon la composante non fonctionnelle. Cette anticipation des tests de non régression permet d'orienter l'évolution de telle sorte à ce que Δ soit minimal.

Dans un premier temps, je commencerai par présenter mes hypothèses de travail. Ces hypothèses m'ont conduit à introduire le concept de "contrat d'évolution" et un algorithme d'assistance à l'évolution qui sera décrit ensuite. J'illustrerai, dans la section 6.4, un exemple d'évolution montrant le fonctionnement de cet algorithme d'assistance. Je conclurai ce chapitre en présentant la structure des contrats d'évolution, ainsi que leur langage de description.

6.2 Importance des liens unissant besoins qualité et architecture

6.2.1 Besoins qualité et architecture des logiciels

Il est admis que ce ne sont pas les fonctions attendues d'un logiciel qui déterminent son architecture, mais bien les attributs qualité requis [11]. L'exemple présenté dans la section 1.2.1 illustre ce propos. L'architecture du composant MACS n'a pas été dictée par les simples faits d'implanter telle ou telle fonctionnalité ou de disposer de tel ou tel composant sur étagères. Des choix architecturaux ont eu pour objectif l'obtention de certains besoins qualité.

Ainsi, le composant AdminDonnees a été introduit pour découpler les formats de données pris en charge par le composant de ceux usités à l'intérieur de celui-ci. Ce composant joue le rôle de "façade" au sens des patrons de conception. Ce découplage permet au composant de respecter le besoin qualité qui avait été formulé dans sa spécification initiale. De la même manière, la séquence de sous-composants CtlAccesDup et ArchiveurDup est une duplication architecturale de la fonction d'authentification. En effet, si la séquence normale CtlAcces et Archiveur n'est pas utilisable du fait, par exemple, d'une absence de données en entrée via le port GestionDonnees, il sera toujours possible à un certain nombre de personnes (dont les autorisations sont stockées localement dans le sous-composant CtlAccesDup) d'accéder au bâtiment. Cette duplication architecturale a été introduite pour garantir un haut niveau de disponibilité même dans un mode dégradé telle que réclamée dans la spécification du composant.

Cet exemple illustre bien l'importance des aspects qualité sur les choix architecturaux réalisés lors de la conception d'un composant.

6.2.2 Évolution et liens architecture-besoins

La connaissance des liens unissant attributs qualité et choix architecturaux est donc du plus grand intérêt pour les personnes en charge de l'évolution et ce à double titre. Premièrement, si l'architecture est bâtie sur la base d'une recherche de certaines propriétés de qualité, la construction d'une image mentale suffisante de cette architecture, préalable à toute modification, passe nécessairement par la reconstitution de cette connaissance. Faciliter cette reconstitution, c'est diminuer d'autant les efforts à consentir lors de la coûteuse phase de compréhension de la structure existante. Deuxièmement, la mise à disposition de ces informations peut éclairer un développeur lors de l'élaboration d'une stratégie d'évolution. Remettre en cause un choix architectural, c'est en effet se poser la question du devenir des attributs qualité dont ce choix visait l'obtention. Il est dès lors possible, à chaque étape d'un processus d'évolution, non seulement d'identifier les éléments architecturaux concernés par une évolution, mais également, d'identifier les risques potentiels d'altération de certaines propriétés de qualité. Sur les liens, unissant choix architecturaux et propriétés non fonctionnelles, peut alors s'établir une démarche cyclique allant du besoin vers la stratégie (recherche de la partie de l'architecture à modifier partant du trait de spécification concerné) et de la stratégie vers le besoin (évaluation de l'impact d'une modification de l'architecture sur la spécification).

Notons de plus, que si l'on dispose d'un moyen automatisant ce deuxième point, c'est-à-dire d'un mécanisme à même d'alerter des conséquences de modifications, on met en place les conditions nécessaires à une démarche de vérification a priori selon la composante non fonctionnelle. Dans l'exemple précédent, le fait de supprimer le composant AdminDonnees lors d'une évolution peut potentiellement manifester une perte du niveau de maintenabilité. Notifier automatiquement, au moment même de la suppression, cette perte potentielle peut aider l'architecte à déterminer s'il doit ou non maintenir sa décision.

La proposition consiste donc à expliciter les liens unissant les spécifications qualité et les choix architecturaux en usant, d'une part, d'un langage formel à même de décrire des choix architecturaux, et d'autre part, d'un mécanisme d'association capable de lier ces choix à des énoncés de spécifications de type attribut qualité. Le choix d'un langage formel garantira non seulement la non ambiguïté des descriptions, mais permettra également la proposition d'un

mécanisme d'alerte automatique comme celui évoqué plus haut. En effet, à l'aide d'un outil évaluant le respect des choix architecturaux documentés à la demande, il sera possible d'indiquer au développeur les choix architecturaux remis en cause et, par association, les attributs qualité potentiellement affectés. Sous réserve de réglementer les actions possibles suite à ces alertes, on se retrouve dans un système contribuant d'une part à la mise à jour de la documentation et d'autre part à la vérification a priori de la progression et de la non régression. Ce type de système augmente grandement les chances d'aboutir à une solution correctement documentée, remplissant les nouveaux besoins, tout en préservant les attributs qualité qui ne devaient pas être altérés. Il vient compléter, sans pour autant la remplacer, la vérification classique par test. En détectant, au plus tôt, certaines erreurs, on diminue d'autant le nombre des tests non passants et donc les coûteux allers-retours qui leur seraient associés.

6.3 Implantation sous la forme de contrats d'évolution

Dans l'approche que je propose, un choix architectural est perçu comme une contrainte dont on cherche à vérifier la validité à chaque "pas" d'une évolution. L'ensemble de ces contraintes, et les liens qui les associent aux propriétés qualité, constituent ce que j'appelle le **contrat d'évolution** d'un logiciel [152]. Dans un premier temps, je détaille la structure de ce contrat. J'indique ensuite de quelle manière ce contrat va être exploité pour assister le processus d'évolution.

6.3.1 Contrats d'évolution

Je parle de contrats car ils documentent les droits et devoirs de deux parties : le développeur de la précédente version du logiciel qui s'engage à garantir les attributs qualité, sous réserve du respect, par le développeur de la nouvelle version, des contraintes architecturales que le premier avait établies.

Le vocabulaire suivant est associé aux contrats d'évolution :

NFP (Propriété Non Fonctionnelle) : clause dans le document de spécification non fonctionnelle du logiciel relative à un attribut qualité (par exemple la clause "Le service de transfert devra s'exécuter en moins de 10ms" relative à l'attribut qualité "Performance") ;

AD (Décision Architecturale) : partie de l'architecture du logiciel qui cible une ou plusieurs NFP (par exemple le respect d'un style en Pipeline à un endroit particulier de l'architecture du logiciel). Pour sa description, une AD peut être construite, si nécessaire et dans un souci de factorisation, sur la base d'autres AD ;

NFT (Tactique Non Fonctionnelle) : couple (AD, NFP) définissant un lien entre une décision architecturale AD et une propriété non fonctionnelle NFP que vise cette décision (par exemple l'AD spécifiant un style en pipeline avec une NFP relative à la performance d'un service particulier offert par le logiciel) ;

NFS (Stratégie Non Fonctionnelle) : ensemble de toutes les NFT définies pour un logiciel particulier ;

La NFS est élaborée durant le développement de la première version d'un logiciel. Ses NFT peuvent apparaître dans chaque phase de développement où une AD motivée est faite. La NFS est donc construite graduellement et s'enrichit durant tout le processus de développement. Certaines NFT peuvent également être héritées d'un plan qualité (lui même instance d'un manuel qualité) et donc émergées avant le début du développement. Il est en effet fréquent dans les entreprises dotées d'une politique qualité mature, d'énumérer des règles architecturales à respecter dans des documents en annexe de leur manuel qualité.

6.3.2 Algorithme d'assistance à l'évolution

Lors d'un cycle d'évolution, une NFS ne doit pas être altérée n'importe comment, sous peine de conduire le logiciel à un état incohérent. Par exemple, il n'est pas concevable pour une NFP devant être maintenue, dans la nouvelle version du logiciel, de se retrouver sans une NFT associée après l'évolution. Au mieux cela manifeste une erreur de documentation (des choix architecturaux ont été faits pour garantir l'obtention de la NFP mais ils n'ont pas été documentés), au pire, cela traduit une possible régression (aucun choix architectural n'a été fait pour garantir l'obtention de la NFP dans la nouvelle version du logiciel). J'ai donc introduit des règles qui définissent les droits et devoirs d'un chargé de l'évolution. Un suivi strict de ces règles doit limiter le risque pour un logiciel d'atteindre un état incohérent. Une NFS ne peut évoluer que dans le respect des règles qui suivent [150].

- **Règle 1 :** "*Une version acceptable d'un logiciel est un système où chacune des NFP est impliquée dans au moins une NFT*". Cette condition garantit, à la fin du processus d'évolution, qu'il n'existe aucune NFP pendante (sans AD associée). Le non respect de cette règle implique de facto, soit le refus de la création d'une nouvelle version, soit l'obligation (en toute connaissance de cause) de modifier la spécification pour lui retirer les NFP incriminées ;
- **Règle 2 :** "*Nous ne devons pas interdire lors d'un pas d'évolution l'abandon d'une AD. Nous notifions simplement l'abandon de l'AD en précisant les NFP affectées (celles apparaissant avec l'AD dans une NFT)*". Il revient au développeur, pleinement averti des conséquences, du maintien ou non des modifications. Si la modification est maintenue, les NFT correspondantes seront éliminées. Cette flexibilité est essentielle car, dans la suite, on pourrait substituer l'AD abandonnée par une autre à même de conserver les NFP ciblées. De plus, on peut être amené à invalider temporairement une décision pour effectuer une modification spécifique. Dans les deux cas, la règle 1 imposera de documenter à nouveau les liens unissant la nouvelle AD et les NFP concernées.
- **Règle 3 :** "*Nous pouvons ajouter de nouvelles NFT à la NFS*". Durant une évolution, de nouvelles décisions architecturales peuvent compléter ou remplacer les anciennes.

6.4 Exemple d'évolution avec assistance

Je vais maintenant montrer sur un exemple comment ces règles permettent d'assister le développeur lors de son activité d'évolution. Je commence par décrire le contrat d'évolution

porté par le composant exemple avant son entrée dans un cycle d'évolution. Je détaille ensuite comment ce contrat d'évolution est exploité par l'algorithme précédent pour assister le développeur à chaque étape du cycle d'évolution.

6.4.1 Contrat d'évolution avant évolution

L'assistance associée aux règles précédentes est illustrée par le scénario d'évolution de la Figure 6.2. Dans ce système, je dispose avant évolution d'une NFS contenant 6 NFT (voir le bas de la figure). Ces NFT décrivent les liens unissant 5 AD (AD1, AD2, AD3, AD4, AD5) et 4 NFP (NFP1, NFP2, NFP3 et NFP4). Par exemple, il y a 3 NFT ayant en première composante NFP1. L'une d'entre-elles précise que NFP1 est en partie obtenue par le respect de la décision AD4.

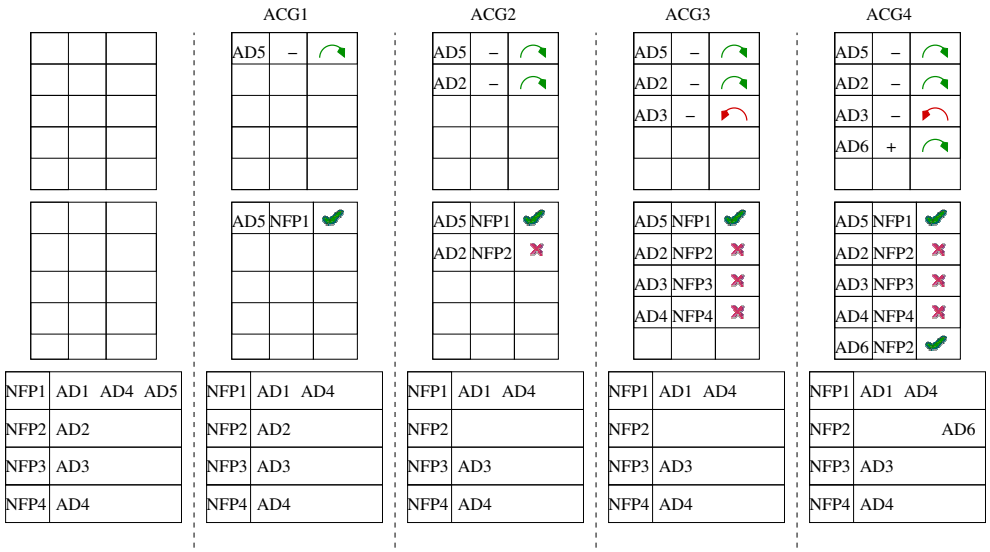

FIG. 6.2 – Assistance à l'activité de l'évolution architecturale avec le contrat d'évolution

Les changements architecturaux effectués à chaque pas d'évolution sont représentés dans le haut de la figure. Le symbole (-) indique que la décision architecturale correspondante a été perdue. Le symbole (+) manifeste que l'AD est préservée ou améliorée. Les flèches vers l'avant indiquent que le développeur a décidé de valider son changement, et les flèches vers l'arrière l'annulation de ce changement. Au milieu de la figure, j'illustre les réactions du système d'assistance face aux changements, en particulier les différentes alertes qu'il déclenche et l'indication de la correction (symbole de validation) de la NFS ou sa non correction (symbole en forme de croix) après le pas d'évolution.

6.4.2 Évolution avec assistance

Supposons que la personne réalisant l'évolution du logiciel, auquel est associée la NFS ci-dessus, applique un changement architectural ACG1. En évaluant la NFS, le système a détecté

que ce changement affecte l'AD5. On indique alors au développeur qu'il se peut que la NFP associée à AD5 (NFP1) soit altérée. Le développeur décide, malgré tout, de valider son changement (il le peut selon la règle 2). La NFS est considérée valide (règle 1) car il n'existe pas de NFP sans AD associée (voir le bas de la figure).

Ensuite, le développeur décide d'appliquer une modification architecturale ACG2. Le système lui notifie alors que AD2 et son NFP associée (NFP2) sont affectées. Il décide cependant de continuer (règle 2). Mais alors, la NFS devient invalide (règle 1). En effet, il existe désormais une NFP (NFP2) sans AD associée.

Plus tard, le développeur essaye d'appliquer un autre changement, ACG3. Il se voit notifié que AD3, et par conséquent NFP3, vont être affectées. En plus, cette AD est incluse dans la description d'une autre AD (AD4). En analysant la NFS et en parcourant les relations de dépendance entre AD, le système d'assistance notifie le développeur que AD4, NFP1 et NFP4 peuvent être également altérées (règle 2). Cette fois-ci, le développeur ne maintient pas son changement. Il faut noter que la NFS est toujours invalide (règle 1 non respectée).

A la fin, le développeur décide d'appliquer un nouveau changement architectural (ACG4). Il introduit en particulier une nouvelle AD (AD6). Sa motivation, avec ce changement, est de remettre dans l'architecture une AD qui garantisse l'obtention de NFP2. Il documente cette décision en ajoutant une NFT composée du couple AD6 et NFP2 (règle 3). Après ce changement, toutes les NFP ont des AD correspondantes. La NFS est à nouveau valide.

J'ai montré à travers ce scénario d'évolution comment l'approche proposée assiste le développeur durant ses opérations d'évolution. Cette assistance permet d'obtenir une architecture conforme aux besoins qualité souhaités. La règle 1 a un double intérêt. D'une part, elle force la mise à jour de la documentation. Et d'autre part, elle aide à garantir, en fin d'évolution, la non régression des propriétés qualité visées.

L'un des points délicats de cette approche est de pouvoir écrire les contrats d'évolution, et en particulier les contraintes architecturales, tout au long du cycle de vie d'un logiciel à base de composants. Je vais consacrer toute la section suivante à présenter un langage baptisé ACL qui a été créé pour cet objectif.

6.5 Langage d'expression des contrats d'évolution

Un contrat d'évolution est composé d'une part, d'un ensemble de contraintes architecturales (décrivant les Décisions Architecturales) et d'autre part, de la liste des liens (NFT) unissant ces contraintes avec des énoncés de spécification qualité (NFP). Les décisions architecturales vont être exprimées à l'aide d'un langage ad hoc nommé ACL. Les NFT vont, pour leur part, être décrites en usant d'une documentation XML. Dans cette section, je commence par présenter le langage ACL. Ce langage présente une structure très particulière qu'il est important de bien comprendre. Je commence donc par évoquer cette structure puis présente dans une deuxième sous-section la syntaxe et la sémantique du langage ACL. Dans une troisième sous-section et pour conclure, je décris la structure des documents XML utilisés pour stocker les NFT.

6.5.1 Structure du langage ACL

Je vais, dans un premier temps, lister les différents types de décisions architecturales (AD) que je souhaite prendre en compte. Je met en particulier en évidence la grande diversité des AD que l'on doit pouvoir exprimer. Je présenterai, ensuite, la structure langagière à deux niveaux que j'ai conçue ; une structure originale usant d'un langage de contrainte nommé CCL et de plusieurs méta-modèles regroupés au sein de profils. Cette structure originale facilite la prise en compte de la diversité. Je décris ensuite comment ces deux niveaux s'articulent pour permettre la description d'AD.

6.5.1.1 La grande diversité des AD exprimables

La proposition d'un langage d'expression des AD se heurte à deux difficultés. La première est qu'il est difficile de prévoir tous les types de contraintes que les développeurs pourraient être amenés à exprimer. Néanmoins, il est certain que des contraintes imposant des styles architecturaux, des patrons de conception, des règles de modélisation et de codage émanant de plans qualité doivent pouvoir être écrites.

Ces AD doivent pouvoir être exprimées avec ou sans passage à l'échelle. Il est en effet différent de dire : "les trois sous-composants respectent un style en Pipeline" et "quelque soit le nombre des sous composants, ils doivent respecter un style Pipeline". La première contrainte ne résistera pas à l'ajout d'un quatrième sous composant préservant le style Pipeline, la seconde oui.

Les AD peuvent également impliquer, soit la version en cours de modification uniquement (c'est le cas le plus fréquent), soit la version précédente et la version en cours. Le deuxième cas se présente lorsque certaines AD veulent réglementer, selon un mode différentiel, les structures acceptables. Par exemple, on peut vouloir interdire, pour des raisons de fiabilité, d'ajouter à un composant plus d'une interface fournie lors d'une évolution. Il faut donc être capable d'évaluer des "différences" entre deux versions successives de ce composant. Ces contraintes, qui sont de véritables contraintes d'évolution, ne sont pas inutiles comme l'indique [156].

La seconde difficulté est liée au fait que ce langage doit être utilisable à chaque étape du cycle de vie d'une application. Je dois donc disposer d'un langage capable de s'appliquer aussi bien sur des modèles de haut niveau que sur des modèles d'implantation. Un modèle de haut niveau représente une infrastructure abstraite d'un logiciel. En pratique, c'est un modèle fourni par un langage de description d'architecture (ADL). Un modèle d'implantation représente, au contraire, une infrastructure concrète d'un logiciel, c'est-à-dire un modèle dépendant d'une technologie particulière, telles que EJB [145], COM+/.net [109] ou CCM [119]. Pour répondre à cette diversité, j'ai conçu un langage, nommé ACL (*Architectural Constraint Language*).

6.5.1.2 La réponse à cette diversité : une structure langagière à deux niveaux

Pour résoudre ces difficultés, le langage ACL est doté d'une structure à deux niveaux. Le premier niveau permet l'expression de phrases de la logique des prédicats dans un contexte de modèles de type MOF. Il offre ainsi les opérations de navigation nécessaires sur ce type de modèle, des opérateurs ensemblistes, les connecteurs logiques et les quantificateurs usuels. Il

est assuré par une version d'OCL légèrement modifiée, baptisée CCL (*Core Constraint Language*). Le second niveau prend la forme d'un ensemble de méta-modèles au format MOF. Ces méta-modèles représentent les abstractions structurelles à contraindre, rencontrées dans les principaux langages de modélisation utilisés à chaque étape du cycle de vie. Ces abstractions sont donc introduites lors des phases en amont par les langages de description d'architectures (ADL, UML) et dans les phases de codage par les technologies de composants utilisées.

Chaque couple composé de CCL et d'un méta-modèle particulier représente un **profil ACL**. Chaque profil est utilisé dans une phase particulière du cycle de vie. Par exemple, nous pouvons utiliser le profil ACL pour xAcme [160] afin de documenter les AD lors d'une phase de conception architecturale usant du langage de description d'architecture xAcme. Par la suite, nous pouvons formaliser d'autres AD lors d'une phase de codage usant du modèle de composants CORBA à l'aide du profil dédié à CCM. Le profil xAcme est composé de CCL et du méta-modèle xArch établi pour xAcme (voir plus loin). Le profil CCM est lui composé de CCL et du méta-modèle CCM. Ainsi, à chaque étape du cycle de vie, un développeur utilise un profil particulier pour documenter ses AD. L'écriture de ces AD est d'autant plus facile qu'il manipule, pour ce faire, les mêmes abstractions que celles présentes dans le langage qu'il a coutume d'utiliser lors de cette étape.

Le langage ACL sépare donc clairement deux aspects que l'on trouve entrelacés dans les langages de contraintes architecturales de la littérature. En effet, tous ces langages sont dotés de grammaires dont le vocabulaire terminal mélange aussi bien des opérateurs de contraintes (navigation, opérateurs de la logique des prédicats), que des abstractions architecturales (composant, connecteur, interface, etc.). Ces langages ne peuvent donc être utilisés, par construction, que dans le contexte exclusif du langage de modélisation d'architecture auquel ils s'appliquent. La séparation de ces deux aspects permet, au contraire, d'obtenir un langage modulable, donc de taille réduite, à même de s'appliquer dans toutes les étapes du cycle de vie. De plus, la puissance du langage pour une étape particulière peut également être améliorée par simple amendement du méta-modèle concerné. Ces méta-modèles sont en effet des paramètres fournis en entrée du compilateur d'expressions ACL. Ils peuvent donc être modifiés sans qu'il soit nécessaire de réécrire le compilateur.

6.5.1.3 Articulation d'un couple (CCL, méta-modèle) au sein d'un profil

Pour comprendre comment cette structure à deux niveaux s'articule pour un profil donné, il est nécessaire de revenir sur le mode d'expression habituel des contraintes OCL dans un diagramme de classes. Dans ce type de diagramme, OCL s'utilise le plus souvent pour spécifier des invariants de classe, des pré-/post-conditions d'opération, des contraintes de cycle entre associations, etc. Ces contraintes restreignent le nombre des diagrammes d'objets valides instanciables depuis un diagramme de classes. Elles pallient un manque d'expressivité de la notation UML graphique qui, employée seule, peut autoriser dans certains cas l'instanciation de diagrammes d'objets non compatibles avec la réalité que l'on souhaitait modéliser. Les contraintes OCL sont décrites relativement à un contexte. Ce contexte est un élément du diagramme de classes, le plus souvent une classe ou une opération présente sur ce diagramme. Voici deux exemples de contrainte OCL.

```
context MemoireThese inv:
self.taille >= 100
context m:MemoireThese inv:
m.ecritPar->size() = 1
```

Dans les deux cas, le contexte est une classe (MemoireThese). Les deux contraintes sont écrites selon le point de vue d'une instance quelconque du contexte (ici un objet instance de la classe MemoireThese). Mais la contrainte exprimée doit être vérifiée par toutes les instances du contexte. Telle est l'approche adoptée par le langage OCL. La première de ces contraintes référence cette instance en usant du mot clé self. A l'opposé, la seconde introduit un identificateur ad-hoc (m) afin de la désigner. Ces deux modes de désignation, tolérés par OCL, sont sémantiquement équivalents. Pour toute contrainte OCL, les éléments apparaissant sont des éléments, soit prédéfinis dans le langage OCL (-¿, size(), etc.), soit des éléments du diagramme de classes atteignables par navigation depuis le contexte (attribut taille et association ecritPar).

Il est intéressant de se demander quel peut être le sens de contraintes OCL écrites non pas sur un modèle mais sur un méta-modèle. Un méta-modèle expose les concepts d'un langage et les liens qu'ils entretiennent entre eux. Il décrit une grammaire abstraite. Une contrainte ayant pour contexte une méta-classe va, de ce fait, limiter la puissance d'expression des règles de production de cette grammaire et donc le nombre des phrases (i.e. modèles) dérivables. Certaines structures de phrases sont écartées. Si ce méta-modèle décrit la grammaire d'un langage de description d'architectures, une contrainte exprime que seules certaines architectures (i.e. modèles) sont dérivables (i.e. instanciables) dans ce langage. Le langage est sciemment bridé dans son pouvoir d'expression car la description de certains types d'architecture n'est pas tolérée. Par exemple, on peut imposer que, dans tout modèle, les composants aient moins de 10 interfaces requises, en posant cette contrainte dans le contexte de la méta-classe Composant. Cette contrainte est exactement du type de celle que je souhaite pouvoir exprimer. Malheureusement sa portée est globale. Elle s'applique à tout composant et non à un composant particulier comme je souhaite le faire. Il ne reste donc plus qu'à trouver le moyen d'adjoindre à la contrainte précédente, un filtre permettant de restreindre sa portée aux seuls composants incriminés. Ce mécanisme de réduction de portée est présent dans le langage CCL dont je vais donner maintenant la description.

6.5.2 Syntaxe et sémantique du langage ACL

Comme nous venons de le voir, le langage ACL repose sur une structure à deux niveaux. Je présenterai successivement le contenu du premier niveau (le langage de contrainte CCL) et celui du second niveau (en prenant pour exemple deux profils).

6.5.2.1 Le premier niveau du langage ACL : le langage CCL

Le langage CCL est un dialecte OCL ; plus exactement de la version 1.5 de ce langage dont on pourra trouver la grammaire et la sémantique sur le site de l'OMG. Le langage OCL présente, en effet, plusieurs avantages. Il fait l'objet d'un standard dans le sillage d'UML et est donc connu de tous. Il était en effet important de ne pas ajouter un nième langage à la liste aussi longue que peu pratiquée des langages de contraintes architecturales. Il offre également

une puissance d'expression satisfaisante dans un format relativement simple et intuitif. Son efficacité a en particulier été démontrée dans le cadre de la maintenance [15]. Enfin, il existe des compilateurs de ce langage dont les sources sont libres de droit.

CCL est (presque) un sous ensemble du langage OCL sur le plan syntaxique et sémantique. La syntaxe du langage CCL est donné dans l'annexe A de ce mémoire. Les seules règles de production du langage OCL qui ne sont pas supportées par CCL sont celles relatives à l'expression des pré-/post-conditions. De plus, pour limiter la portée d'une contrainte à un composant particulier, je propose de modifier légèrement la syntaxe et la sémantique de la partie contexte d'OCL (CCL). Sur le plan syntaxique, j'impose que tout contexte introduise nécessairement un identificateur. Cet identificateur doit obligatoirement être le nom d'une ou plusieurs instances particulières de la méta-classe citée dans le contexte. Sur le plan sémantique, j'interprète la contrainte avec le sens qu'elle aurait dans le contexte d'une méta-classe mais en limitant sa portée aux instances citées dans le contexte.

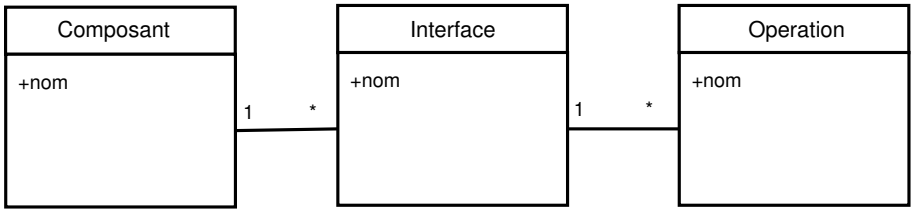

FIG. 6.3 – Un méta-modèle jouet

Pour mieux comprendre la syntaxe et la sémantique de CCL, voici un petit exemple. On supposera que le méta-modèle jouet (figure 6.3) parcouru par cette contrainte est composé de 3 méta-classes (Composant, Interface et Opération) et de 3 associations qui décrivent l'univers suivant : un composant affiche des interfaces, une interface a un nom et regroupe un ensemble d'opérations.

```
context Toto, Titi : Composant inv:
(self.interface->select(i|i.nom = "Maison")->size() = 1)
and
(self.interface->forAll(i|i.operation->size()<7))
```

Sémantiquement, la contrainte est interprétée comme si elle s'appliquait au méta-modèle. Elle manifeste donc que l'on ne peut pas produire de modèles de composants dans lesquels : un composant ne disposerait pas d'une et une seule interface de nom Maison et dont les composants comporteraient plus de 6 opérations par interface. Dans la mesure où nous réduisons la portée de cette contrainte à seulement deux instances de la méta-classe Composant (les composants Toto et Titi), seuls ces deux composants devront la respecter.

Le seul ajout au langage OCL est un marqueur qui permet de désigner l'ancienne version de l'artefact architectural auquel il s'applique : le marqueur @old. Considérons l'exemple suivant s'appliquant au méta-modèle jouet.

```
context C : Composant inv:
(self.interface->size()) < ((self@old.interface->size())+2)
```

Cette contrainte précise que, d'une version à sa suivante, le composant C ne peut pas augmenter de plus d'une unité le nombre de ses interfaces. Notons qu'elle ne dit rien sur les modifications éventuellement subies par les anciennes interfaces du composant. Si nous voulions imposer, qu'au plus une seule de ces anciennes interfaces ne puisse être altérée d'une version à une autre, nous devrions comparer les structures de toutes ces interfaces avant et après évolution. Ce type de contraintes serait pénible à écrire avec les opérations OCL usuels et le seul marqueur @old. J'ai donc introduit les opérations de collection qui suivent :

- modified() :Collection(T) : cette opération retourne tous les éléments de la collection à laquelle elle s'applique et qui ont subi une modification entre l'actuelle et la précédente version (seulement pour ceux existant dans les deux versions). Cette comparaison se fait sur la base d'un identifiant associé aux éléments de ces collections.
- added() :Collection(T) : cette opération fait de même pour les éléments qui n'existaient pas dans la précédente version,
- et deleted() :Collection(T) : cette opération fait de même pour les éléments qui ont disparus de la nouvelle version.

La contrainte totale précédente s'écrirait donc :

```
context C : Composant inv:
(self.interface->added()->size()) < 2) and
(self.interface->deleted()->isEmpty) and
(self.interface->modified()->size() < 2)
```

Maintenant que le langage CCL a été présenté, je vais présenter les différents profils existants. Un profil est composé du langage CCL et d'un méta-modèle sur lequel seront écrites les contraintes. Un méta-modèle détaille les abstractions architecturales manipulées par le langage de modélisation (ADL Acme, UML, etc ;) ou d'implantation (EJB, CCM, etc.) dont on souhaite contraindre les modèles. Ces contraintes, décrivant des AD, seront à vérifier sur les modèles documentés dans ces langages lors d'une étape d'un cycle de vie. Comme je souhaitais supporter un nombre suffisant de langages pour couvrir l'intégralité du cycle de vie d'un logiciel à base de composants, j'ai à ce jour défini 5 profils (donc 5 méta-modèles) : xAcme, UML, CCM, EJB et Fractal [18]. Je ne présenterai dans la suite que deux d'entre eux : xAcme et CCM. Le premier a le mérite de réaliser une synthèse acceptable des abstractions trouvées dans des langages à haut niveau d'abstraction que sont les ADL. Le second est un représentant de la classe des technologies d'implantation.

6.5.2.2 Un premier exemple de profil : xAcme pour les ADL

Un ADL doit, en principe, pouvoir décrire une architecture logicielle sous la forme des trois C : les Composants, les Connecteurs et les Configurations [102]. Les composants représentent les unités de calcul et de stockage de données dans un système logiciel. L'interaction entre ces composants est encapsulée par les connecteurs. Dans la configuration, l'architecte instancie un nombre de composants et un nombre de connecteurs. Il les lie entre eux afin de construire son système. Une étude approfondie a révélé que, si une partie des ADL répond effectivement à cette description, un nombre conséquent ne la respecte pas. Rapide [92], Darwin [96] ou Koala [156] par exemple, n'explicitent pas la notion de connecteur. D'autres permettent la des-

cription hiérarchique des composants. Dans ce cas, les composants peuvent être perçus comme des boîtes blanches et peuvent donc avoir une structure interne explicite. Dans certains ADL tels que Rapide, les composants sont, au contraire, considérés plutôt comme des boîtes noires. Dans UniCon [142], Wright [3] ou encore Acme [55], nous pouvons définir des connecteurs composites, alors que cela reste impossible dans les autres ADL. Ces divergences sont liées au fait que la plupart des ADL tentent de répondre à des objectifs particuliers. Darwin et Koala visent la reconfiguration dynamique des architectures. Rapide a, pour sa part, pour objectif la description architecturale de systèmes événementiels. SADL [115] se focalise sur le raffinement des architectures. C2SADEL modélise des architectures dans le style C2 [103], etc. La classe des ADL est donc assez hétérogène.

J'avais alors deux solutions pour couvrir l'ensemble des ADL : proposer, soit un profil propre à chaque ADL, soit un profil unique pour tous les ADL. La première solution a le mérite d'offrir un méta-modèle dédié à chaque ADL de la littérature. Ces méta-modèles peuvent donc incorporer toute la richesse des concepts structuraux d'un ADL précis et en conséquence permettre l'écriture de contraintes exploitant toutes les constructions de cet ADL cible. Le problème est qu'il m'aurait fallu plus d'une dizaine de profils pour couvrir les ADL les plus connus à ce jour. De plus, les ADL que je ne connaissais pas se retrouveraient sans profil et pour chaque nouvel ADL apparaissant il m'aurait fallu créer un nouveau profil. La seconde solution consiste à proposer un seul profil (donc un seul méta-modèle), plus générique, pour tous les ADL. Ce méta-modèle unique ne peut pas exploiter toutes les finesses de chaque ADL mais il peut être capable de répondre à la plupart des besoins. C'est cette dernière solution que j'ai retenue. J'ai donc créé un profil unique pour tous les ADL, dont le méta-modèle est celui de xArch [68] décrivant les concepts structuraux du langage xAcme. Ce langage est le résultat d'un projet qui vise à promouvoir un ADL, au format XML, regroupant les concepts communs aux ADL les plus connus (Acme, C2SADEL, Rapide, etc.). Il réalise ainsi une synthèse des abstractions communes à la plupart des ADL. J'ai donc décidé de l'utiliser comme référence pour élaborer le méta-modèle.

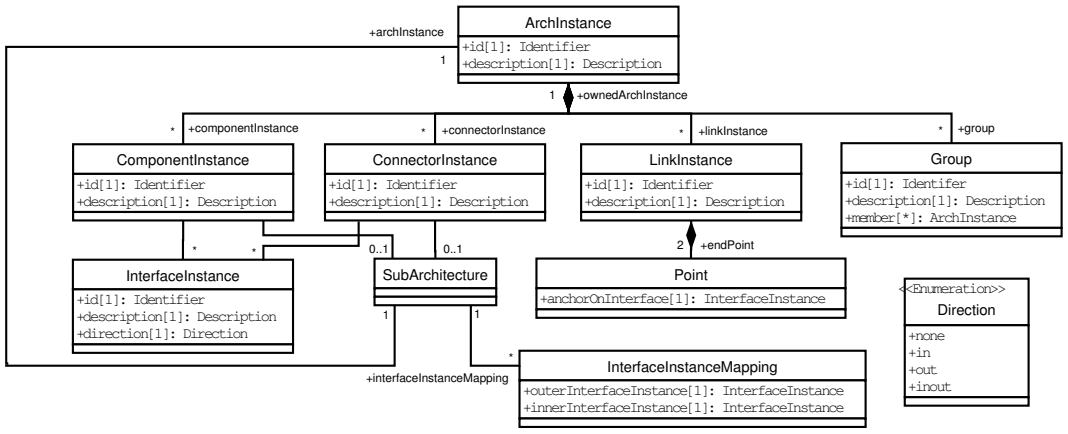

FIG. 6.4 – Le méta-modèle du profil xArch

La figure 6.4 représente le méta-modèle MOF du profil xAcme. Le concept le plus général est celui de ArchInstance. Cette abstraction peut représenter une instance de composant (ComponentInstance), une instance de connecteur (ConnectorInstance), un lien entre deux instances architecturales (LinkInstance), ou un groupe d'instances (Group). Une instance de composant ou de connecteur possède aucune ou plusieurs instances d'interfaces (InterfaceInstance). Un composant peut également définir une sous-architecture d'instances. La configuration de cette sous-architecture est décrite par des correspondances entre interfaces (InterfaceInstanceMapping).

6.5.2.3 Exemples de contraintes dans le profil xAcme

Dans cette section, je présente quelques exemples de contraintes architecturales écrites avec le profil ACL pour xAcme. Je commence par un exemple simple d'une contrainte qui impose l'existence d'au plus cinq interfaces par composant. Malgré sa simplicité, cette contrainte est d'une importance non négligeable. On peut trouver ce type de contraintes dans les manuels qualité de certaines entreprises de développement de logiciels. Cette contrainte peut être décrite sur tous les sous-composants[1] du composant MACS (introduit dans le chapitre 1) comme suit :

```
context MACS: ComponentInstance inv:
MACS.subArchitecture.archInstance.componentInstance
->forAll(c:ComponentInstance|c.interfaceInstance
->select(i:InterfaceInstance|i.direction = #in)
->size() <= 5)
```

Un autre type de contraintes peut être exprimé avec ACL. Dans la contrainte ci-dessous, j'ai décrit une règle utilisant une mesure de qualité pour les composants (appelés modules par ses auteurs [91]). Cette mesure est nommée CBM pour *Coupling Between Modules*.

```
context MACS: ComponentInstance inv:
MACS.subArchitecture.archInstance.componentInstance
->forAll(c|c.archInstance.linkInstance
->size() < 5)
```

Cette métrique est l'équivalent d'une métrique bien connue dans le monde de la conception objet, qui est CBO (*Coupling Between Objects* [24]). Elle représente le nombre de références distinctes inter-composants. Afin de préserver un certain niveau de maintenabilité du composant MACS et de limiter les dépendances entre ses sous-composants, une contrainte ACL est décrite. Elle stipule que CBM doit être bas (CBM < 5) [91].

Cette approche ressemble à celle présentée dans les travaux de Brito e Abreu. Avec certains auteurs, il propose d'utiliser OCL au niveau du méta-modèle UML pour décrire des métriques de conception orientée objets [8]. Dans d'autres travaux, ils présentent un méta-modèle pour CCM et utilisent OCL pour formaliser quelques métriques sur la structure des composants [59]. Les contraintes qu'ils définissent ne s'appliquent pas sur des éléments de conception (classes ou composants) bien précis, comme avec ACL. Leurs contraintes représentent des invariants qui s'appliquent à tous les modèles instances des méta-modèles UML et CCM (diagrammes de

[1] Pour l'instant, j'impose que seuls les sous-composants directs d'un composant hiérarchique sont accessibles.

classe UML ou composants CCM). Ils proposent, également, une manière de formaliser avec OCL des métriques pour les bases de données objets-relationnelles [9].

Le troisième exemple de contrainte ACL introduit un invariant architectural général qui prévient l'existence d'un connecteur reliant directement les sous-composants de MACS à un composant externe.

```
context MACS:ComponentInstance inv:
MACS.subArchitecture.archInstance.linkInstance
->forAll(l|MACS.subArchitecture.archInstance.componentInstance
.interfaceInstance->includes(l.point.anchorOnInterface))
```

La contrainte ci-dessous décrit une contrainte qui garantit le style architectural façade dans le composant MACS (AD2 dans la section 1.2.1).

```
context MACS:ComponentInstance inv:
let boundToGestionDonnees:Bag = MACS.archInstance.linkInstance
->select(l|l.point.anchorOnInterface.direction = #in
and l.point.anchorOnInterface.id = 'GestionDonnees') in
...
```

La variable *boundToGestionDonnees* contient un multi-ensemble composé de tous les liens qui ont comme source l'interface *GestionDonnees*.

```
...
((boundToGestionDonnees->size() = 1)
and (boundToGestionDonnees.point.anchorOnInterface
->select(i|i.direction = #in).componentInstance
->select(c|c.id = 'AdminDonnees')->size() = 1))
```

La taille de la collection récupérée dans la variable précédente doit être égale à 1. De plus, le seul composant qui doit être lié à cette interface est le composant *AdminDonnees*. Ce dernier fera donc l'objet d'un composant façade.

La contrainte suivante concerne les composants répliqués énoncés dans la décision AD3 de la section 1.2.1. Elle permet de garantir cette réplication lors de l'évolution.

```
context MACS:ComponentInstance inv:
let startingIntf:InterfaceInstance = MACS.interfaceInstance
->select(i|i.id = 'AuthentificationUtilisateur') in
let startingComponent:ComponentInstance = MACS
.subArchitecture.archInstance.componentInstance
->select(c|c.interfaceInstance = startingIntf) in
...
```

La variable *startingIntf* contient une référence vers l'interface du début de la chaîne des composants répliqués. Cette variable est utilisée pour déterminer le composant du début de cette chaîne (la variable *startingComponent*).

```
...
let endingIntf:InterfaceInstance = MACS.interfaceInstance
->select(i|i.id = 'ArchivageServeurCentral') in
let endingComponent:ComponentInstance = MACS
.subArchitecture.archInstance.componentInstance
->select(c|c.interfaceInstance = endingIntf) in
...
```

Les deux variables *endingIntf* et *endingComponent* contiennent respectivement l'interface de la fin de la chaîne des composants répliqués et le composant définissant cette interface.

```
....
let paths:OrderedSet = MACS.subArchitecture
.getPaths(startingComponent,endingComponent) in
paths->size() = 2 and paths->first()->excludesAll(paths->last())
```

Les variables *startingComponent* et *endingComponent* sont utilisées pour récupérer l'ensemble des chemins reliant ces deux composants sous forme d'un ensemble ordonné de composants. Le nombre de chemins distincts doit être égal à 2.

Dans cette contrainte, on utilise une opération particulière du méta-modèle xArch[2]. Cette opération (getPaths(c1 :ComponentInstance,c2 :ComponentInstance)) retourne un ensemble ordonné de tous les chemins entre deux composants passés en paramètre. Les chemins retournés sont aussi représentés sous forme d'ensembles ordonnés de composants. Les chemins retournés ne contiennent pas les composants passés en paramètre. Cette contrainte vérifie l'existence de deux chemins distincts entre le composant attaché à l'interface AuthentificationUtilisateur et le composant attaché à l'interface ArchivageServeurCentral.

6.5.2.4 Un second exemple de profil pour les composants CORBA

Les technologies de composants fournissent, à la fois, un modèle abstrait de développement d'applications à base de composants et une infrastructure concrète pour leur déploiement. Ces infrastructures fournissent l'environnement nécessaire à l'exécution de ces applications, en terme de services transactionnels, de sécurité, de répartition, etc. Les modèles abstraits proposés par ces technologies permettent de définir une application sous la forme de composants fournissant un certain nombre de services (interfaces) et explicitant leurs dépendances avec leur environnement. Pour définir le profil pour CORBA, seuls les concepts présents dans son modèle abstrait étaient utiles.

La figure 6.5 représente le méta-modèle MOF pour le langage de description structurelle des composants CCM. Un ComponentAssembly définit des connexions entre ports et/ou interfaces. Un port peut représenter une facette, un réceptacle ou un ou plusieurs événements. Une facette définit l'ensemble des services fournis par le composant. Le réceptacle précise les services requis par le composant. Les événements peuvent être émis, consommés ou publiés pour plusieurs clients. Un port peut être représenté sous la forme d'interfaces requises ou fournies, qui exportent un certain nombre d'opérations. Un composant peut définir des attributs. Un composant peut être de différents types : session, entité, service, process ou aucune de ces catégories. Par contre, CCM est un modèle de composants plat. Un composant ne peut être conçu sur la base d'autres composants.

[2]Cette opération n'est pas illustrée dans la Figure 6.4. D'autres opérations de même type que celle-ci seront discutées avec plus de détails dans le prochain chapitre (voir section 7.3.7).

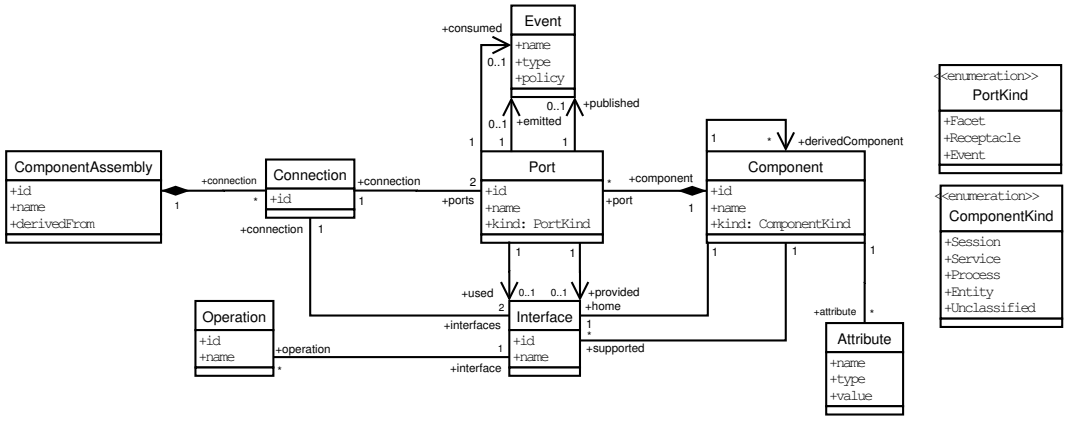

FIG. 6.5 – Le méta-modèle du profil CORBA

6.5.2.5 Exemples de contraintes dans le profil pour les composants CORBA

Les même contraintes présentées précédemment sont écrites ci-dessous avec le profil ACL pour CCM.

– Contrainte 1 : Nous pouvons définir au maximum cinq interfaces fournies par sous-composant.

```
context MACS:ComponentAssembly inv:
MACS.connection.port.component
->forAll(c|c.port.provided->size() <= 5)
```

– Contrainte 2 : CBM doit être bas (CBM < 5)

```
context MACS:ComponentAssembly inv:
MACS.connection.port.component
->forAll(c|c.port.connection->size() < 5)
```

– Contrainte 3 : Un connecteur ne doit pas traverser la structure interne d'un composant. Cette contrainte n'a pas d'équivalent en CCM parce qu'il n'existe pas de concept de composant hiérarchique dans cette technologie.

– Contrainte 4 : patron architectural façade. Cette contrainte est organisée de la même manière que dans le profil ACL pour xAcme. Elle vérifie l'existence d'un composant unique lié à l'interface *GestionDonnees*.

```
context MACS:ComponentAssembly inv:
let boundToGestionDonnees:Bag = MACS.connection
->select(con|con.port.provided.name = 'GestionDonnees')
in
(boundToGestionDonnees->size() = 1)
and (boundToGestionDonnees.port.provided.component.name
->includes('AdminDonnees')))
```

– Contrainte 5 : Composants répliqués. Cette contrainte navigue dans le méta-modèle CCM, mais elle est structurée de la même manière que la contrainte précédente. En

d'autres termes, on récupère d'abord les composants du début et de la fin de la chaîne des composants répliqués. Ensuite, on doit vérifier qu'il n'existe que deux chemins distincts entre ces composants.

```
context MACS:ComponentAssembly inv:
let startingIntf:Interface = MACS.connection.port.interface
->select(i|i.name = 'AuthentificationUtilisateur') in
let startingComponent:Component = startingIntf.component in
let endingIntf:Interface = MACS.connection.port.interface
->select(i|i.name = 'ArchivageServeurCentral') in
let endingComponent:Component = endingIntf.component in
let paths:OrderedSet =
MACS.getPaths(startingComponent,endingComponent) in
paths.size() = 2 and paths->first()->excludesAll(paths->last())
```

Nous remarquons que la différence entre les contraintes ci-dessus et les contraintes de la section 6.5.2.3 est minime. Elle concerne principalement les abstractions architecturales contraintes et les navigations, dans les méta-modèles, entre ces abstractions. Plus de détails sur ce point seront donnés dans le chapitre suivant.

6.5.3 Description des NFT et des NFS

Je viens de présenter le langage ACL qui permet de documenter des décisions architecturales. Il me reste à proposer un moyen pour documenter les liens unissant ces décisions (que j'appelle des AD) aux besoins de qualité (que j'appelle des NFP). Dans cette approche, un tel lien est appelé Tactique Non Fonctionnelle (NFT). L'ensemble des NFT d'un composant constitue une Stratégie Non Fonctionnelle (NFS). Dans un premier temps, je vais détailler la structure adoptée pour documenter les NFT. Je donnerai, dans un deuxième temps, un exemple de ce type de documents.

6.5.3.1 La structure des NFT

Les définitions introduites précédemment sont illustrées par la figure 6.6. Les décisions architecturales (AD) sont construites selon un modèle hiérarchique. Une AD peut donc être construite sur la base d'autres AD. La décision AD1 a été construite sur les décisions AD2 et AD3. Une décision architecturale est décrite par une contrainte écrite en ACL. Une NFT est la donnée d'un couple (AD, NFP). Elle manifeste que le développeur a fait le choix AD dans le but de satisfaire la totalité ou une partie du besoin NFP. Une décision peut apparaître dans plusieurs NFT. Ainsi la décision AD3 se retrouve dans deux NFT (NFT1 et NFT2). Une même décision peut donc être liée à plusieurs NFP. Ainsi AD3 participe à l'obtention des besoins NFP1 et NFP2. Inversement, une même NFP peut être associée à plusieurs décisions architecturales. Ainsi NFP2 est liée aux décisions AD3 et AD6 (via respectivement NFT2 et NFT3).

Une NFP est un des besoins de qualité formulés dans le document de spécification d'un composant logiciel. Une NFP doit être un besoin atomique. Une NFP est dite atomique si elle n'est associée qu'à un et un seul attribut qualité dont la portée est unique. Ce modèle accepte

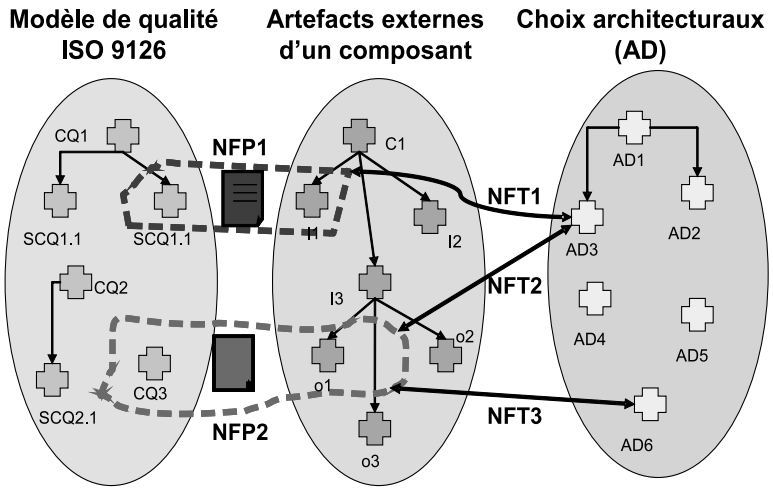

FIG. 6.6 – Expression des NFS

actuellement comme attribut qualité, les caractéristiques et les sous-caractéristiques du modèle de qualité ISO/IEC 9126 [67] (par exemple, la maintenabilité, la portabilité, etc.). Un attribut qualité est porté par un artefact architectural externe. Un artefact externe représente un concept architectural public d'un composant ; c'est-à-dire un artefact visible par les autres composants. Dans mon approche, les artefacts externes sont les composants, les interfaces et les opérations. Une NFP est donc un triplet composé : d'un artefact architectural externe cible, d'un attribut qualité et du texte provenant du document de spécification non fonctionnelle qui décrit ce besoin. Voici deux exemples de NFP qui pourrait être construites depuis la documentation d'un composant : (Maintenabilité, composant C, "le composant C doit supporter facilement d'autres formats de fichier") et (Performance, opération o() de l'interface I, "L'opération o() doit garantir un temps de réponse inférieur à 10ms"). Le développeur a donc pour obligation de découper la spécification non fonctionnelle d'un composant en autant de NFP que nécessaire.

En l'état je ne fais aucune hypothèse sur le format d'écriture des besoins dans la documentation du composant. Ils peuvent être formulés sous forme de textes libres (c'est le cas le plus fréquent) ou en usant d'un langage dédié à l'expression de certaines propriétés non fonctionnelles (par exemple QML [48] pour certains aspects de la qualité de service). L'absence d'un langage d'expression de propriétés non fonctionnelles digne de ce nom restreint le niveau de formalisation d'une NFP et donc d'une NFT. Je suis ici tributaire des avancées dans un domaine encore ouvert et en devenir, objet de nombreux travaux. Face à une telle situation et dans le but de gérer le maximum de cas possibles, la seule contrainte que j'impose, pour des raisons

de stockage et d'affichage, est que le format d'expression des besoins permette un stockage sous la forme de chaînes de caractères. Bien sûr, cette flexibilité se paye par l'absence de toute structure pour la troisième composante du triplet définissant une NFP ; absence préjudiciable à la mise sur pied de mécanismes de lien NFT plus puissants.

6.5.3.2 Le stockage des NFS

Très concrètement, dans cette approche, une NFS qui détaille l'ensemble des NFT prend la forme d'un document XML respectueux d'un schéma XML reprenant la structure que je viens de décrire. Le listing ci-dessous représente l'exemple d'une NFS (un contrat d'évolution).

```
<nonfunctional-strategy id="000001">
  <nonfunctional-tactic id="000100">
    <description>
      Cette tactique garantit la NFP portabilite
      par le biais du patron de conception Facade
    </description>
    <nonfunctional-property id="001000" name="Portabilite"
       characteristic="Portabilite" extern-arch-artifact="MACS">
      <description>
        Le composant doit etre portable sur
        differents environnement. Il peut servir
        differents types d'applications clientes.
      </description>
    </nonfunctional-property>
    <architecture-decision id="010000">
      <description>
        Patron de conception Facade
      </description>
      <formalization id="100000" profile="CCM">
        <!--Ici on retrouve la contrainte-->
      </formalization>
    </architecture-decision>
  </nonfunctional-tactic>
</nonfunctional-strategy>
```

Cet exemple illustre une NFS composée d'une seule NFT. Cette NFT représente le couple (AD, NFP) où AD est le choix du patron de conception Façade. La NFP ici est formée du triplet : i) l'attribut qualité Portabilité, ii) l'artefact architectural externe auquel est associé cet attribut et qui représente le composant nommé MACS et iii) la description textuelle du besoin qualité impliquant cette NFP. La dernière partie de cette NFS contient la contrainte qui formalise la décision architecturale (patron de conception Façade). Dans cet exemple, la contrainte est écrite avec le profil ACL pour le modèle de composants CORBA (CCM).

6.6 En résumé

Dans ce chapitre, j'ai présenté une approche par contrats qui vise, au delà de la simple documentation d'une architecture logicielle pour sa meilleure compréhension, à automatiser certaines vérifications durant l'évolution. Ces vérifications peuvent, à la demande du développeur, assister l'activité d'évolution et lui notifier l'impact des changements architecturaux sur les décisions architecturales. Cette assistance permet d'alerter des conséquences sur les besoins qualité associées à ces décisions. Elle oriente donc le développeur durant l'évolution vers une certaine direction de telle manière à ce que la déviation des spécifications de qualité initiales soit minimale (voir Figure 6.7). Cela réduit bien évidemment le nombre de tests effectués a posteriori et durant lesquels on détecte cette déviation. La réduction de ce nombre de tests contribue à la minimisation du coût global de la maintenance.

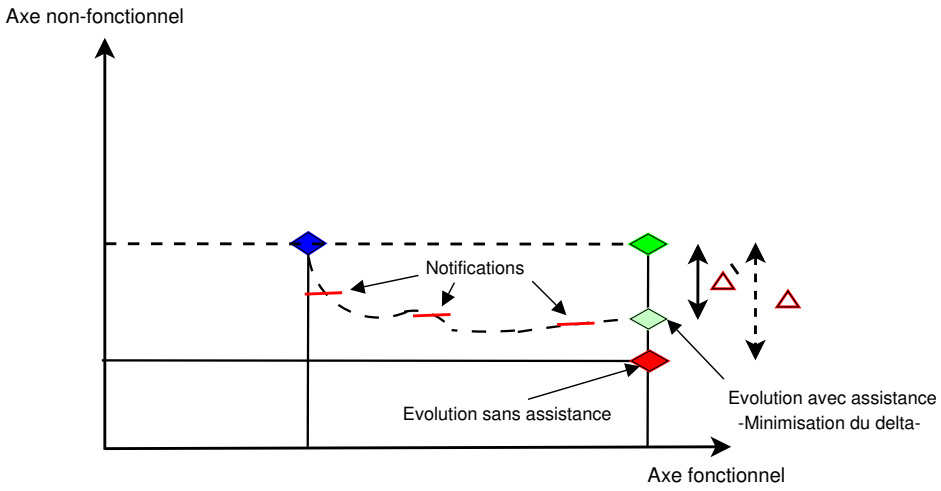

FIG. 6.7 – Assistance à l'évolution dirigée par la qualité

Supposons maintenant que l'évolution architecturale vise, pour certaines modifications, plutôt l'aspect non-fonctionnel. Dans ce cas, comme ce fut proposé dans la section 6.3.2, l'algorithme d'assistance impose la modification des spécifications non-fonctionnelles et des contrats d'évolution. Il garantit ainsi une assistance prenant en compte l'évolution des spécifications non-fonctionnelles. Sur la Figure 6.8, la ligne épaisse avec pointillés (en haut de la figure) représente le nouvel axe théorique d'évolution avec spécifications non-fonctionnelles constantes.

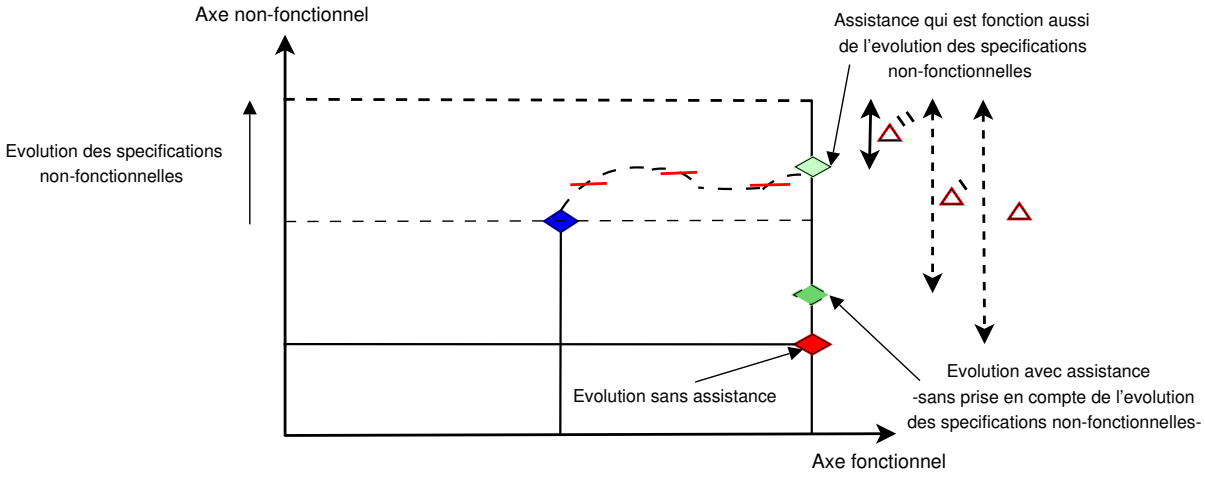

FIG. 6.8 – Assistance avec évolution des spécifications non-fonctionnelles

L'évolution de l'architecture atteint ainsi le premier point (en haut) à droite, au lieu du deuxième. Dans ce dernier cas, nous supposons que l'assistance à l'évolution a été effectuée sans prise en compte de l'évolution des spécifications non-fonctionnelles. Nous concluons que les contrats d'évolution évoluent avec les spécifications de besoins non-fonctionnels et garantissent ainsi une assistance cohérente et à jour vis à vis de ces derniers. Cet aspect est important dans la documentation logicielle et contribue considérablement dans la réduction des coût de maintenance [25].

Chapitre 7

Traçabilité des décisions architecturales dans un processus de développement

Sommaire

7.1 Introduction au besoin de tracer les décisions architecturales

Les contrats d'évolution ne sont pas simplement utilisés pour préserver la qualité d'un logiciel à base de composants durant l'évolution de son architecture, ils permettent également, par le biais de leur langage de description ACL, de tracer les décisions architecturales tout au long du cycle de développement. Ceci répond à la problématique 1 introduite dans le chapitre 1 :

les décisions architecturales doivent être explicites dans toutes les étapes du cycle de vie pour qu'elles ne soient pas altérées lors du passage d'une étape à une autre. En effet, les contrats d'évolution sont définis et enrichis tout au long du cycle de vie d'un logiciel. Dès qu'une décision est prise et justifiée au niveau architectural, elle est documentée et introduite dans le contrat. A toute étape du cycle de vie, on dispose donc de toutes les décisions architecturales formalisées dans les étapes en amont.

Dans ce chapitre, je présenterai comment cette problématique a pu être résolue à l'aide du langage ACL. Tout d'abord, deux besoins en terme d'expression de décisions et leur traçabilité sont introduits dans la section suivante. Ensuite, la solution à ces besoins est détaillée dans les sections 7.3 et 7.4.

7.2 Besoins de la traçabilité des décisions architecturales

Dans un processus de développement d'un logiciel à base de composants, il est possible d'établir, à l'étape de conception, une description d'architecture dans un ADL donné. A partir de cette description d'architecture, nous pouvons définir, à l'étape de conception de composants, un diagramme de composants UML 2. Ce diagramme peut être raffiné pour obtenir à la fin une implémentation à base de composants EJB ou CCM. Ces transitions peuvent être réalisées par transformations successives dans un contexte d'ingénierie dirigée par les modèles.

Dans la Figure 7.1, ce processus est illustré sur une partie du système MACS (voir chapitre 1). Dans ce processus de développement, nous disposons d'une description d'architecture définie à l'aide de l'ADL Acme. A partir de cette description, nous définissons un diagramme de composants UML 2. Ce modèle nous permet par la suite d'obtenir une implémentation en composants CORBA.

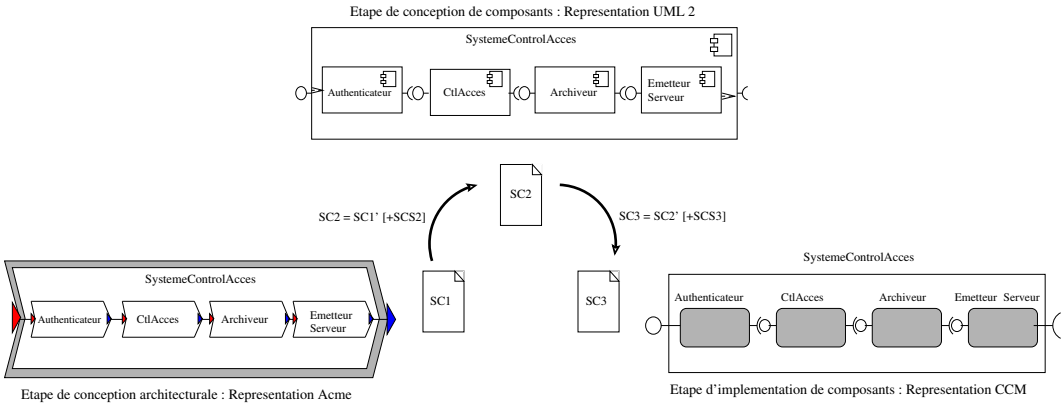

FIG. 7.1 – Préservation des décisions dans un processus de développement

Supposons que, lors de la première étape de conception architecturale, une spécification de contraintes, formalisant les décisions architecturales prises, a été définie. Cette spécification est

nommée SC1 dans la Figure 7.1. Lors de la deuxième étape, de nouvelles contraintes peuvent apparaître. Ces contraintes sont nommées SCS2. Dans cette étape, SC1 doit également pouvoir être évaluée. La vue dont on dispose de SC1 dans l'étape 2 est nommée ici SC1'. La spécification de contraintes de l'étape de conception de composants (étape 2) est donc l'union de SC1' et SCS2. De même, pour l'étape 3, SC3 est égal à l'union de SCS3 (les contraintes spécifiques à l'étape 3) et de SC2' (les contraintes SC2 vues dans l'étape 3).

Afin que les décisions architecturales puissent être préservées deux besoins émergent :

1. les contraintes SCS2 et SCS3 doivent pouvoir être exprimées. Plus généralement, les contraintes architecturales doivent pouvoir être exprimées à toutes les étapes du cycle de vie ;
2. les contraintes SC1 et SC2 doivent pouvoir être évaluées facilement dans les étapes 2 et 3 respectivement. Plus généralement, les contraintes architecturales exprimées dans les étapes en amont à une étape donnée doivent pouvoir être facilement évaluées dans cette étape-ci.

7.3 Expression des contraintes à divers niveaux

Afin de satisfaire le premier besoin énoncé ci-dessus, une famille de langages de contraintes (ACL) a été proposée. Chaque profil ACL peut être utilisé pour formaliser les décisions à une étape donnée du processus de développement. Dans l'exemple précédent, les contraintes SC1 peuvent être décrites à l'aide du profil ACL pour Acme. Ce dernier est composé du méta-modèle Acme et de CCL. Dans la deuxième étape, les contraintes SCS2 peuvent être exprimées à l'aide du profil ACL pour UML 2 (CCL + méta-modèle de composants UML 2). Les contraintes SCS3 de la dernière étape peuvent être exprimées à l'aide du profil ACL pour CCM. Ceci est illustré dans la Figure 7.2.

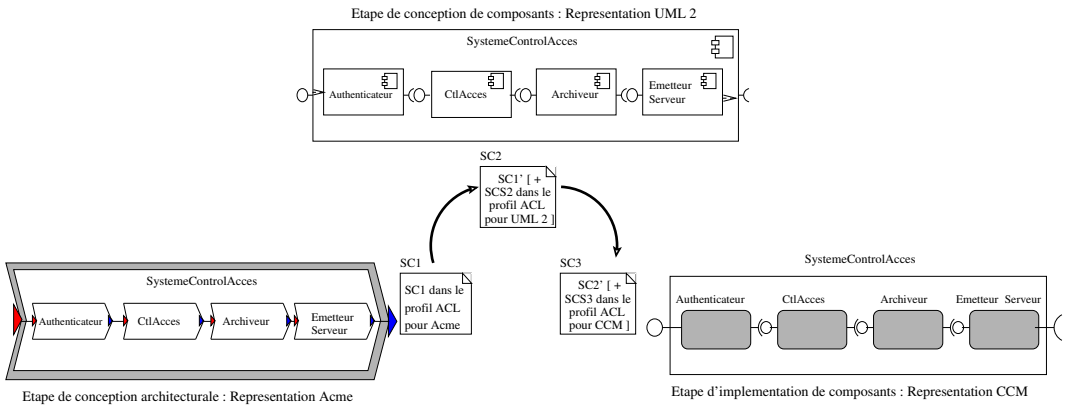

FIG. 7.2 – Expression des contraintes à divers niveaux

La satisfaction du deuxième besoin sera abordée plus tard dans ce chapitre. Tout d'abord,

je vous présente la manière avec laquelle ces différents profils ACL sont évalués. La méthode d'évaluation qui va être introduite dans les sous-sections suivantes va être utilisée pour répondre au deuxième besoin.

7.3.1 Problématique de l'évaluation des contraintes

Comme énoncé ci-dessus, ACL n'est pas un simple langage mais une collection de langages. Chaque élément de cette collection (profil ACL) définit en fait un langage. La difficulté est de fournir autant d'évaluateurs différents qu'il y a de profils (actuellement 5). Il est plus judicieux, de par les ressemblances que présentaient les méta-modèles de chacun des profils, de définir : un méta-modèle façade auquel on associe un unique évaluateur et autant de traducteurs d'un profil vers ce langage intermédiaire. Ce méta-modèle façade forme avec le langage CCL, le profil standard ArchMM. Avec ce mode de fonctionnement, pour évaluer un contrat sur un composant, on traduit le descripteur d'origine de ce composant dans un modèle conforme à ArchMM. Ce modèle est une instance d'ArchMM générée par transformation depuis le descripteur d'architecture du composant. Les décisions architecturales sont également traduites dans le profil ArchMM. Elles sont ensuite évaluées sur le modèle ArchMM généré précédemment.

Toute la difficulté était de trouver un méta-modèle façade suffisamment riche et abstrait pour fournir un espace de traduction sans perte depuis chacun des 5 profils. J'ai donc bâti ArchMM sur une étude comparative de deux types de méta-modèles. Le premier concerne les langages de description d'architecture et le second, les technologies de composants. Une étude sur les ADL Acme, Koala [156], xADL [31], et d'autres, a été menée[1]. Pour la deuxième catégorie de méta-modèles j'ai comparé les technologies de composants : EJB (*Enterprise JavaBeans*) de Sun Microsystems [145] et CCM (*CORBA Component Model*) de l'OMG [119]. J'ai jugé intéressant d'étendre mon étude sur le modèle de composants hiérarchique Fractal du consortium ObjectWeb et sur la partie du méta-modèle UML concernée par les composants.

Dans une première étape, il m'a fallu élaborer un méta-modèle pour chaque niveau (conception ou implémentation). Cette tâche était relativement simple dès lors que des standards dans les deux niveaux ont émergé. Je veux dire, par standard, une norme de fait, comme CCM dans les technologies de composants et Acme (par son extension XML, ADML) pour les ADL.

Le méta-modèle est présenté dans la prochaine section. Je présenterai, dans les sous-sections suivantes, les abstractions architecturales représentées dans les ADL, UML 2 et les technologies de composants.

7.3.2 Abstractions architecturales dans les ADL

Les abstractions architecturales introduites par les ADL ont été présentées dans le chapitre précédent dans la section 6.5.2.2. La plupart des ADL permettent la description des types et des instances de composants et de connecteurs. Leur description peut contenir des descriptions d'interfaces, de ports ou de rôles. Une configuration ou une topologie d'une application consiste en un ensemble de liens (de type 1-1, N-1, 1-N, N-M) entre les éléments précédents. Il est également possible de hiérarchiser la description et donc de définir des composants ou des connecteurs composites. Ceci est réalisé à l'aide de liens hiérarchique entre composants ou

[1] Je me suis basé également sur un autre état de l'art établi il y a quelques années par Medvidovic et Taylor [102].

connecteurs et leurs sous-éléments. Un exemple de méta-modèle MOF d'un ADL a été fourni dans le précédent chapitre. Il concerne xAcme, un ADL générique et englobant les abstractions architecturales communes aux autres ADL.

7.3.3 Abstractions architecturales dans UML 2

UML fournit une notation unifiée pour la modélisation et un standard supporté par plusieurs outils du génie logiciel. Dans sa spécification 2.0 [122] (adoptée par l'OMG), il fournit les éléments nécessaires pour décrire à la fois des architectures abstraites et des implémentations à base de composants[2]. Un composant est considéré comme un élément de modélisation à part entière (tout comme une classe) et non pas comme une entité déployable et existante à l'exécution. Cependant, UML est un langage de modélisation à objectif général. Par conséquent, il existe une multitude de concepts présents dans son méta-modèle qui rendent ce dernier relativement complexe. L'objectif visé (comme énoncé précédemment) est de pouvoir définir le contrat d'évolution le plus facilement et rapidement possible. Pour ce faire, le méta-modèle doit comporter des abstractions simples et connues par le développeur.

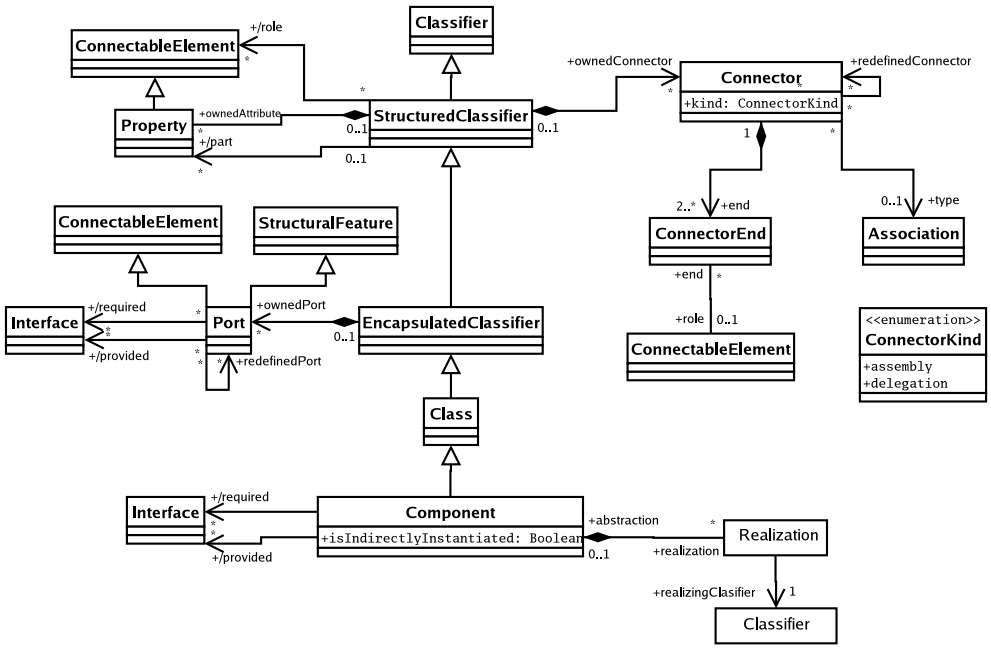

FIG. 7.3 – Le méta-modèle de composants UML2.0

La figure 7.3 présente un extrait du méta-modèle UML2.0 pour la description de composants. Un composant hérite de la méta-classe Class. Il peut donc définir des opérations et participer dans des généralisations. Il hérite également de EncapsulatedClassifier. Il peut ainsi

[2]Chapter 8 : Components. Spécification de la super-structure UML 2.0 [122], page 133.

posséder des ports, typés par des interfaces fournies et requises. La méta-classe Encapsulated-Classifier hérite de StructuredClassifier. Par conséquent, un composant peut avoir une structure interne et définir des connecteurs.

7.3.4 Abstractions architecturales dans les technologies de composants

Sur la base des profils UML[3] pour EJB [138] et CCM [124], j'ai établi mes propres méta-modèles pour ces deux technologies de composants. J'ai défini mes propres méta-modèles dès lors que les profils UML ne constituent pas la solution idéale, car trop complexes pour la simple description de contraintes. Dans le précédent chapitre, le méta-modèle MOF de CCM a été présenté. Ce méta-modèle englobe les abstractions architecturales fournies dans EJB. Cependant, CCM est un modèle de composants plats.

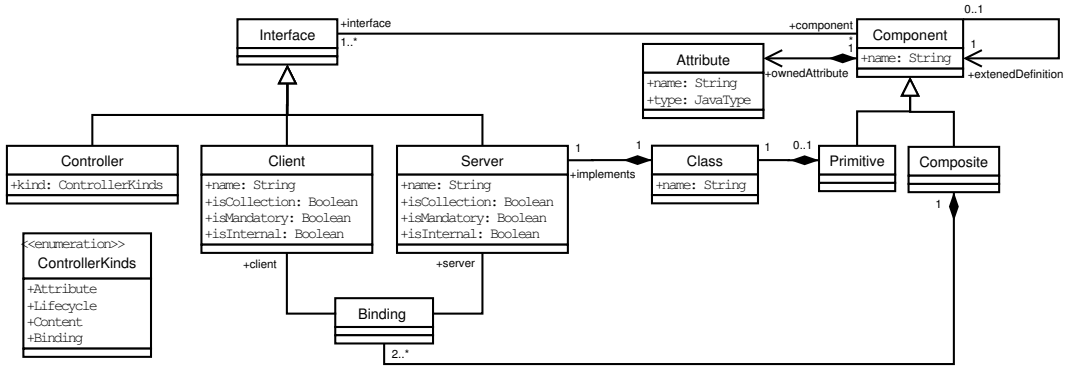

FIG. 7.4 – Un méta-modèle MOF de Fractal

La figure 7.4 représente un méta-modèle MOF pour le modèle de composants Fractal. A la différence de CCM, ce modèle de composants est hiérarchique. En effet, on distingue dans ce modèle de composants les composants primitifs des composants composites (hiérarchiques). Un composant primitif est directement implémenté par un module logiciel bas niveau (par exemple, une classe JAVA pour l'implémentation Julia du modèle Fractal). Un composite décrit un certain nombre de liens (*Bindings*) entre ses sous-composants. Un composant fournit et requiert un certain nombre d'interfaces (de type serveur et client, respectivement). Il est possible de définir des interfaces obligatoires ou optionnelles. Ceci signifie qu'elles sont obligatoirement ou optionnellement implémentées par le composant. Elles peuvent également être des collections d'interfaces. Dans ce cas, elles communiquent avec plusieurs interfaces (de types opposés -client ou serveur-) à la fois. Un composant dispose d'un certain nombre d'interfaces

[3]Un profil UML est une extension bien formée du méta-modèle UML. Par extension bien formée, je signifie une extension conforme aux spécifications du méta-modèle UML. Plus concrètement, c'est une extension du méta-modèle UML basée sur les stéréotypes, les valeurs marquées (*Tagged Values*) et les contraintes.

non-fonctionnelles, appelées interfaces contrôleurs, qui gèrent son contenu, son cycle de vie, etc. Il peut définir également un certain nombre d'attributs qui maintiennent son état.

7.3.5 Synthèse des abstractions architecturales

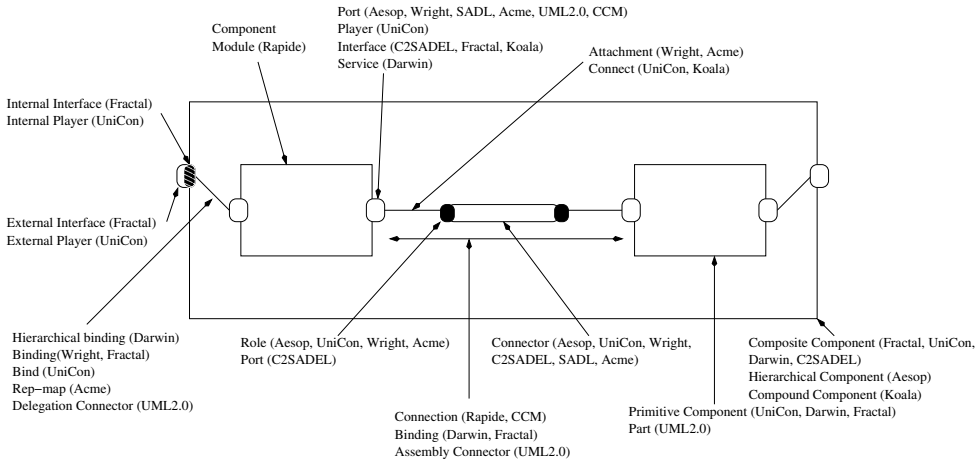

FIG. 7.5 – Représentation graphique des abstractions architecturales

Une synthèse des différents éléments architecturaux supportés par ces différents méta-modèles est présentée sur la figure 7.5. Il faut noter également les points suivants. Dans certains de ces modèles, les notions de genre de composant, port, interface, connecteur ou rôle sont présentes. Nous pouvons définir avec un langage ou une technologie donnée plusieurs variétés des éléments ci-dessus. Par exemple, un port dans CCM peut être une facette, un réceptacle, etc. Dans certains ADL, on distingue les interfaces internes des interfaces externes. Une interface interne est l'interface du composant composite sur laquelle est relié un connecteur hiérarchique. Une interface externe est une interface sur laquelle est relié un connecteur d'assemblage de même niveau, ou une interface d'un sous-composant sur laquelle est relié un connecteur hiérarchique. Certains proposent des attachements que nous appelons de type MN (un lien entre un composant et un connecteur de même niveau) et des attachements de type H (un lien entre un composant composite et un connecteur hiérarchique).

7.3.6 Description du profil standard ArchMM

Suite à l'étude précédente, j'ai élaboré un profil standard constitué de CCL et d'un méta-modèle nommé **ArchMM**. Ce méta-modèle unifie et abstrait les concepts architecturaux représentés par tous les autres méta-modèles de composant.

La figure 7.6 présente le méta-modèle ArchMM dans un format MOF [120]. Dans ce méta-modèle, un système est décrit par un certain nombre de composants (ComponentInstance dans xArch et Component dans CCM) qui représentent les unités de calcul et de données. Le mode

de communication et de coordination entre ces composants est encapsulé par des connecteurs (ConnectorInstance dans xArch et Connector dans CCM). La configuration du système (SubArchitecture dans xArch et ComponentAssembly dans CCM) est représentée par un assemblage de composants par le biais de connecteurs. Ce méta-modèle met en avant les concepts suivants :

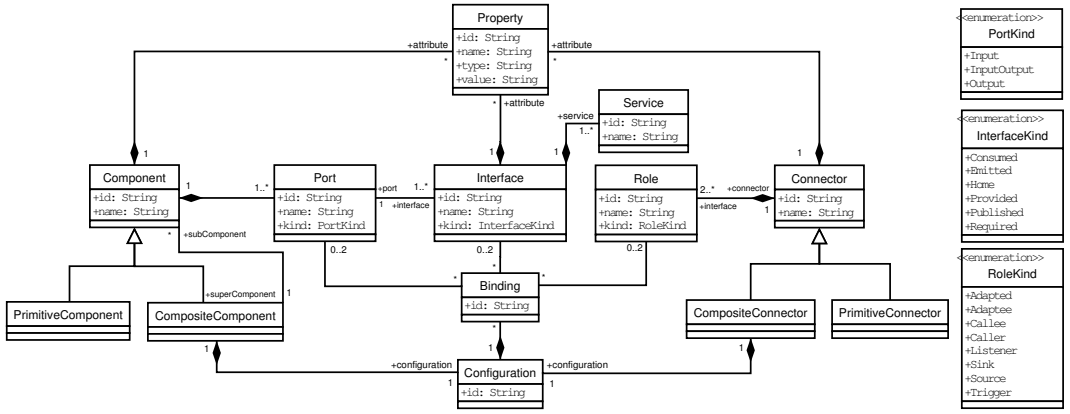

FIG. 7.6 – ArchMM : le méta-modèle façade

– Un composant définit un ou plusieurs ports. Ces ports représentent le point d'interaction du composant avec son environnement. Un port peut être de genre : i) Input, lorsqu'il reçoit des appels en entrée, ii) Output, lorsqu'il reçoit des appels en sortie, iii) ou InputOutput, lorsqu'il peut recevoir des appels en entrée et en sortie. Un port peut supporter une ou plusieurs interfaces de différents genres : requises par le composant (used interfaces dans le modèle de composants CORBA et in interfaces dans xArch), fournies (provided interfaces dans CCM et out interfaces dans xArch), etc. Les événements dans CCM et dans d'autres ADL sont considérés comme des interfaces dans ArchMM. Ces interfaces peuvent être donc également de genre : emitted, published ou consumed. Les interfaces définissent un ensemble de services, qui correspondent aux opérations CCM.
– Un connecteur définit un certain nombre de rôles. Ces derniers correspondent aux interfaces de connecteurs dans xArch. Un rôle peut être de genre : i) Adapted ou Adaptee, lorsqu'il s'agit d'un connecteur de genre adaptateur, ii) Callee ou Caller dans le cas d'un connecteur de type invocation de méthode, iii) Listener ou Trigger, lorsqu'il s'agit d'un connecteur de type diffusion d'événement, iv) et Sink ou Source dans le cas d'un connecteur de type pipe. Les connecteurs peuvent définir deux rôles. L'un d'eux décrit son interface d'entrée et l'autre, son interface de sortie. Un connecteur peut être vide. Il représente alors un lien simple entre deux composants (binding en Fractal).
– Une configuration représente plusieurs attachements (*bindings*, à ne pas confondre avec les bindings de Fractal). Ces liens, qui correspondent aux Connections en CCM et aux LinkInstance ou InterfaceInstanceMapping dans xArch, sont définis entre : i) des interfaces de deux composants (requises ou fournies), ii) une interface d'un composant et un rôle de connecteur, iii) le port d'un composant hiérarchique (composite) et l'interface de

l'un de ses sous-composants, iv) ou, les rôles de connecteurs. Il est à noter que, pour être le plus générique possible, dans ce méta-modèle, une interface ou un rôle peut participer dans plusieurs attachements.
- Une description hiérarchique associe une structure interne explicite à un composant ou à un connecteur composites. Cette structure interne est décrite par une configuration de sous-composants et connecteurs. Je suppose que les délégations effectuées des ports des composants aux ports de leurs sous-composants sont réalisées à l'aide de connecteurs dits de délégation. De cette manière, aucun lien direct entre les interfaces des ports de composants et les interfaces de leurs sous-composants n'est autorisé.
- Des propriétés sont associées aux composants, aux connecteurs et aux interfaces. Ces propriétés sont nommées et ont des valeurs typées. Les attributs des composants CCM et les propriétés xAcme, par exemple, sont projetés vers ces propriétés.

Certaines abstractions existent dans plusieurs méta-modèles, comme les composants, les interfaces requises ou fournies. Cependant, certains des concepts présents dans ArchMM n'existent pas dans d'autres méta-modèles. Ces concepts, comme les ports, les bindings, les connecteurs ou les rôles, peuvent être inférés pendant la transformation vers ArchMM. Par exemple, dans CCM, il n'existe pas de connecteurs. Pendant une transformation d'une description CCM vers ArchMM, des connecteurs sont générés avec deux rôles. Dépendant du type des ports des composants à connecter (Facet, Receptacle ou Event), les rôles générés sont, respectivement, Callee, Caller et Listener ou Trigger. La distinction entre événements de type Published ou Emitted est faite au niveau interface dans ArchMM.

7.3.6.1 Quelques exemples d'AD écrites dans le profil standard

Le premier exemple concerne une contrainte typique d'évolution. Cette contrainte implique une comparaison entre deux versions consécutives d'une description d'architecture. Cette contrainte stipule que, lors d'une évolution, seule une interface fournie peut être ajoutée entre deux versions consécutives d'un composant et que les autres interfaces ne doivent pas être supprimées ou modifiées. Cette contrainte peut être exprimée, dans le profil standard d'ACL, comme suit :

```
context MACS:CompositeComponent inv:
((self.port.interface
->select(i:Interface|i.kind = #Provided)->added()->size() < 2) and
(self.port.interface->modified()->isEmpty) and
(self.port.interface->deleted()->isEmpty))
```

Je reprends dans la suite de cette section les contraintes qui ont été présentées dans les sections 6.5.2.3 et 6.5.2.5. Cette deuxième contrainte représente un prédicat fixant le nombre maximum d'interfaces fournies par sous-composant à quatre. Les contraintes sont définies, comme fait précédemment, au composant MACS.

```
context MACS:CompositeComponent inv:
MACS.subComponent->forAll(c:Component|c.port.interface
->select(i:Interface|i.kind = #Provided)->size() <= 5)
```

La troisième contrainte concerne la métrique CBM. Elle formalise une règle qui stipule que cette mesure doit être inférieure à 5.

```
context MACS: CompositeComponent inv:
MACS.subComponent->forAll(c|c.port.interface
.binding->size() < 5)
```

Dans la contrainte ci-dessous, aucun connecteur ne doit traverser la structure interne d'un composant.

```
context MACS: CompositeComponent inv:
MACS.configuration.role.connector
->forAll(c:Connector|MACS.subComponent.port.interface
.binding->includes(c.role.binding))
```

La cinquième contrainte concerne le patron architectural façade. Elle est formalisée dans le profil standard ACL comme suit :

```
context MACS: CompositeComponent inv:
let boundToGestionDonnees:Bag = MACS.configuration.binding
->select(b|b.interface.kind = #Provided
and b.interface.name = 'GestionDonnees')
in
((boundToGestionDonnees->size() = 1)
and (boundToGestionDonnees.interface
->select(i|i.kind = #Provided).port.component
->select(c|c.name = 'AdminDonnees')->size() = 1))
```

Dans la dernière contrainte, nous utilisons le profil standard ACL pour forcer l'existence permanente des composants répliqués (AD3 dans la section 1.2.1 du chapitre 1).

```
context MACS: CompositeComponent inv:
let startingIntf:Interface = MACS.port.interface
->select(i|i.name = 'AuthentificationUtilisateur') in
let startingComponent:Component = MACS.subComponent
->select(c|c.port.interface = startingIntf) in
let endingIntf:Interface = MACS.port.interface
->select(i|i.name = 'ArchivageServeurCentral') in
let endingComponent:Component = MACS.subComponent
->select(c|c.port.interface = startingIntf) in
let paths:OrderedSet = MACS.configuration
.getPaths(startingComponent,endingComponent) in
paths.size() = 2 and paths->first()->excludesAll(paths->last())
```

7.3.7 Vers une simplification de l'expression des contraintes

Les méta-modèles des différents profils, ainsi qu'ArchMM ont été étendus afin de faciliter la description de certains types de contraintes. La Figure 7.7 représente l'extension faite à la méta-classe Configuration du méta-modèle ArchMM. Ceci nous permet l'expression de certaines contraintes impliquant des propriétés de graphe, comme des graphes connexes, des graphes réguliers, des graphes simples, etc. En outre, nous avons la possibilité d'invoquer des requêtes sur les prédécesseurs et les successeurs d'un noeud, les degrés entrant ou sortant d'un

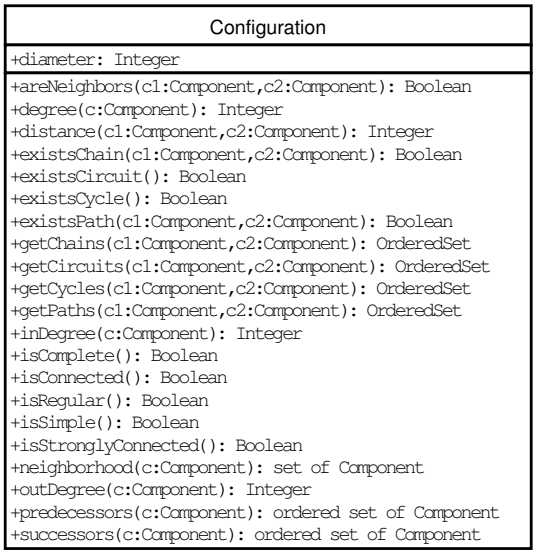

FIG. 7.7 – Propriétés de graphe ajoutées au langage ACL

noeud, la distance entre deux noeuds ou le diamètre d'un graphe. En effet, une description d'architecture ou une configuration de l'assemblage de composants peut être considérée comme un graphe où les noeuds sont les composants et où les arêtes sont les connecteurs entre ces composants.

La contrainte ci-dessous illustre l'utilisation de certaines propriétés de graphe dans la formalisation d'une décision architecturale. Cette décision représente le choix du style architectural pipeline pour la structure du composant de la figure 7.1. Cette contrainte est formalisée avec le profil standard ACL.

```
-- Chaque sous-composant doit avoir des ports en entree et en sortie
context SystemeControleAcces:CompositeComponent inv:
SystemeControleAcces.subComponent.port
->forAll(p:Port|(p.kind = #Input)
or (p.kind = #Output))
and
-- Chaque connecteur doit definir deux roles (l'un de type puits
-- et l'autre de type source)
SystemeControleAcces.configuration.binding.role.connector->AsSet()
->forAll(con:Connector|(con.role->size() = 2)
and ((con.role.kind = #Source)
or (con.role.kind = #Sink)))
and
-- Chaque connecteur relie deux composants (l'entree reliee
-- a puits et la sortie a source
SystemeControleAcces.configuration.binding.role.connector->AsSet()
->forAll(con:Connector|con.role
->forAll(r:Role|SystemeControleAcces.subComponent
->exists(com:Component|com.port
```

```
->exists(p:Port|(r in SystemeControleAcces.configuration.binding)
and ((p.kind = #Input) and (r.kind = #Sink))
or ((p.kind = #Output) and (r.kind = #Source)))))) and
--Le graphe doit etre connexe
SystemeControleAcces.configuration.isConnected and

--Le graphe doit contenir un nombre d'aretes egal au nombre
--de noeuds - 1
SystemeControleAcces.configuration.binding.role.connector
->AsSet()->size() = SystemeControleAcces.subComponent->size()-1
and
--Le graphe doit representer une liste
SystemeControleAcces.subComponent->forAll(com:Component|
(com.port->size() = 2)
and (com.port->exists(p:Port|p.kind = #Input))
and (com.port->exists(p:Port|p.kind = #Output)))
```

Les propriétés de graphes sont définies dans ce profil comme attributs et opérations de la méta-classe Configuration du méta-modèle ArchMM.

7.4 Transformation des contraintes des différents profils

Disposant de ce mécanisme d'évaluation des contraintes ACL passant par un profil façade, une approche simple a été proposée [151] afin de satisfaire le deuxième besoin de préservation des décisions architecturales dans le processus de développement, citée ci-dessus. Cette approche consiste à évaluer, à une étape donnée du processus, toutes les contraintes définies dans les étapes en amont. Ceci est illustré dans la Figure 7.8.

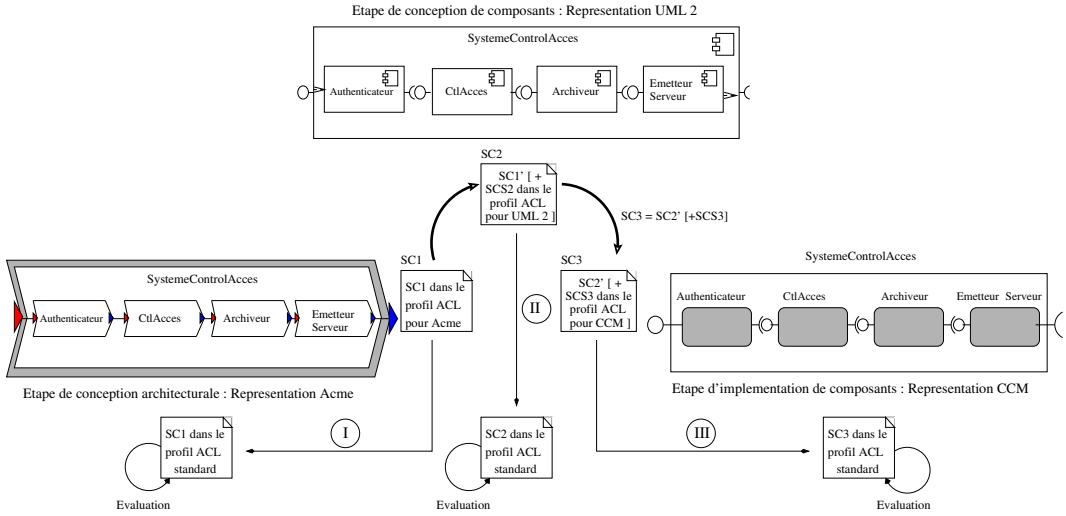

FIG. 7.8 – Transformation des contraintes vers le profil standard

Durant la première étape, les contraintes définies dans le profil ACL pour Acme sont transformées vers le profil standard ACL pour être évaluées. Ceci est représenté par la flèche (I). Les contraintes résultantes sont évaluées sur une description intermédiaire de l'architecture conforme à ArchMM. Cette description, comme j'ai énoncé précédemment, est le résultat de la transformation de la description d'architecture Acme.

Lors de la deuxième étape, les contraintes définies dans le profil ACL pour UML 2 sont transformées vers le profil standard ACL. Les contraintes définies en amont (celles exprimées dans le profil ACL pour Acme) sont transformées également vers le profil standard ACL. Cette transformation est illustrée par la flèche (II). Tout comme dans la première étape, les deux catégories de contraintes sont évaluées sur une description intermédiaire de l'architecture.

Durant l'étape d'implémentation, toutes les contraintes sont transformées (depuis le profil ACL pour CCM, le profil pour UML et celui pour Acme) vers le profil standard pour être évaluées (flèche (III)).

Grâce à l'approche proposée, la transformation des contraintes des différents profils ACL vers le profil standard est rendue simple. En effet, seules les abstractions architecturales sont traitées. La partie CCL est commune à tous les profils et reste globalement inchangée lors de la transformation [153].

7.4.1 Concepts transformés dans les profils

Afin de définir des règles de transformation entre les différents profils ACL et le profil standard, des projections entre les concepts dans un profil donné (source) et les concepts dans le profil standard (cible) ont été mis en place. Les concepts projetés entre les différents profils peuvent être classifiés dans les trois catégories suivantes :

- **Abstractions architecturales :** Dans cette catégorie, nous retrouvons les méta-classes et leurs propriétés. Par exemple, un composant, son nom, son type, etc.
- **Relations entre abstractions :** Dans cette deuxième catégorie, nous projetons des navigations simples entre méta-classes. Par exemple, les rôles d'un connecteur.
- **Patrons de navigation :** Dans cette catégorie, nous définissons des transformations entre navigations complexes et récurrentes dans un méta-modèle. Par exemple, toutes les interfaces requises des sous-composants d'un composant donné.

Chaque projection permet de transformer une partie d'une contrainte écrite dans un profil en une partie de contrainte dans le profil standard.

7.4.2 Méthode de transformation des contraintes

La méthode de transformation des contraintes, illustrée dans la figure 7.9, est constituée des étapes suivantes :

1. **Compilation de la contrainte :** Dans cette première phase, une contrainte source sous la forme textuelle est transformée en un arbre syntaxique abstrait.
2. **Sérialisation de l'arbre syntaxique :** Lors de cette phase, l'arbre syntaxique abstrait résultant de la phase précédente est écrit dans un document XML (le XML Schema de ces documents est donné dans l'annexe D). Toutes les informations dans l'arbre syntaxique ne sont pas sérialisées dans le document XML. Seules les feuilles de l'arbre (les unités

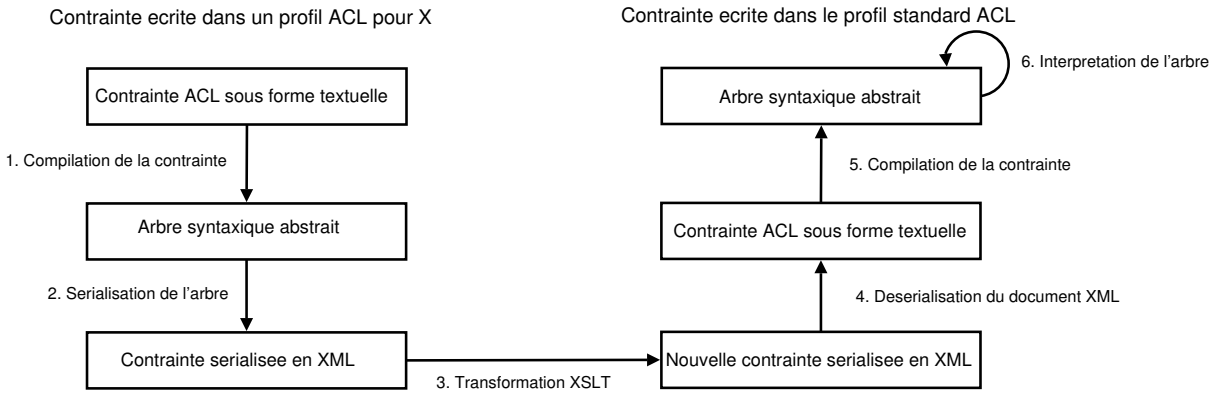

FIG. 7.9 – Transformation XSL des contraintes entre profils

lexicales dans la contrainte) sont rendues persistantes. En plus, pour certaines unités, des informations relatives au type sont ajoutées dans les documents XML. Nous verrons, plus tard dans ce chapitre, l'utilité de ces informations.

3. **Transformation XSL du document XML :** Un ensemble de feuilles de style a été défini pour chaque profil ACL. Ces feuilles de styles contiennent l'implémentation XSLT des règles de projection discutées ci-dessus. Elles permettent de transformer un document XML contenant la contrainte source en un autre document contenant la contrainte cible.
4. **Déserialisation du document XML :** Dans cette étape, la contrainte sous sa forme textuelle est extraite du document XML résultant de la précédente étape.
5. **Compilation de la contrainte :** Cette étape précède l'étape d'évaluation de la contrainte sur la description de l'architecture.

7.4.3 Exemples de contraintes transformées

La contrainte introduite dans la section 7.3.7 représente le résultat de la transformation d'une contrainte écrite dans le profil ACL pour xAcme vers le profil standard. Celle-ci est donnée dans le listing ci-dessous.

```
context SystemeControleAcces:ComponentInstance inv:
SystemeControleAcces.subArchitecture.archInstance
.componentInstance.interfaceInstance
->forAll(i:InterfaceInstance|(i.direction = #in)
or (i.direction = #out))
and
SystemeControleAcces.subArchitecture.archInstance.connectorInstance
->forAll(con:ConnectorInstance|
```

```
(con.interfaceInstance->size() = 2)
and ((con.interfaceInstance.direction = #in)
or (con.interfaceInstance.direction = #out)))
and
SystemeControleAcces.subArchitecture.archInstance.connectorInstance
->forAll(con:ConnectorInstance|con.interfaceInstance
->forAll(iCon:InterfaceInstance|SystemeControleAcces.subArchitecture
.archInstance.componentInstance
->exists(com:ComponentInstance|com.interfaceInstance
->exists(iCom:InterfaceInstance|(iCon in SystemeControleAcces
.subArchitecture.archInstance.linkInstance
.point.anchorOnInterface)
and ((iCom.direction = #in)
and (iCon.direction = #out))
or ((iCom.direction = #out)
and (iCon.direction = #in))))))
and
SystemeControleAcces.subArchitecture.isConnected()
and
SystemeControleAcces.subArchitecture.archInstance.connectorInstance
->size() = SystemeControleAcces.subArchitecture.archInstance
.componentInstance->size() - 1
and
SystemeControleAcces.subArchitecture.archInstance.componentInstance
->forAll(comp:ComponentInstance|
(comp.interfaceInstance->size() = 2)
and (comp.interfaceInstance
->exists(i:InterfaceInstance|i.direction = #in))
and (comp.interfaceInstance
->exists(i:InterfaceInstance|i.direction = #out)))
```

Dans l'un de ces invariants, on utilise l'opération isConnected(). Cette opération est associée dans ce profil au type SubArchitecture. Ce type a la même sémantique que le type Configuration dans le profil standard ACL.

Les concepts de cette contrainte, qui sont transformés vers le profil standard ArchMM, sont résumés dans le tableau 7.1

Cette même contrainte s'écrit dans le profil ACL pour CCM de la manière suivante :

```
context SystemeControleAcces:ComponentAssembly inv:
SystemeControleAcces.connection.port
->forAll(p:Port|(p.provided->size() = 1)
or (p.used->size() = 1))
and
SystemeControleAcces.connection->forAll(con:Connection|
(con.port.provided->size() = 1)
and (con.port.used->size() = 1))
and
SystemeControleAcces.isConnected()
and
SystemeControleAcces.connection->size()
= SystemeControleAcces.connection.port.component->AsSet()
```

```
->size() - 1
and
SystemeControleAcces.connection.port.component->asSet()
->forAll(com:Component |
com.port->size() = 2
and com.port->exists(p:Port|p.provided->size() = 1)
and com.port->exists(p:Port|p.used->size() = 1)
```

Dans le tableau 7.2 sont identifiés les concepts de cette contrainte écrite dans le profil ACL pour CCM et qui sont transformés vers le profil standard d'ACL.

Lors de la transformation, les patrons de navigation sont d'abord recherchés dans la contrainte source, pour appliquer les règles de transformation correspondantes. Si un patron correspond, la portion de la contrainte est transformée par son équivalent dans le profil standard. Si aucun patron n'est trouvé, les relations entre abstractions sont recherchées. Dans ce cas, si une relation est trouvée, elle est remplacée par ce qui lui est associée dans le profil standard. Si aucune relation ne correspond, lors de l'analyse, la recherche des abstractions commence. Chaque abstraction trouvée est remplacée par le concept qui lui correspond dans ArchMM. Si aucun de ces cas n'est trouvé dans la contrainte source, aucune règle n'est appliquée et l'unité syntaxique de la contrainte reste inchangée. C'est le cas, par exemple, de l'identificateur du composant

Nature du concept	**Concepts profil xAcme**	**Concepts profil standard**
Abstractions architecturales	ComponentInstance	CompositeComponent
	InterfaceInstance	Port
	ConnectorInstance	Connector
	SubArchitecture	Configuration
	in (component)	Input
	out (component)	Output
	in (connector)	Source
	out (connector)	Sink
Relations entre abstractions	componentInstance.interfaceInstance	component.port
	connectorInstance.interfaceInstance	connector.role
	interfaceInstance.direction (component) interfaceInstance.direction (connector)	port.kind role.kind
Patrons de navigation	componentInstance.subArchitecture .archInstance.componentInstance	component.subComponent
	componentInstance.subArchitecture .archInstance.connectorInstance	component.configuration.binding .role.connector
	componentInstance.subArchitecture .archInstance.linkInstance .point.anchorOnInterface	compositeComponent.configuration .binding.port− >union(compositeComponent .configuration.binding.role)

TAB. 7.1 – Projection de concepts entre le profil xAcme et le profil standard

Nature du concept	**Concepts profil CCM**	**Concepts profil standard**
Abstractions architecturales	ComponentAssembly	CompositeComponent
Relations entre abstractions	port.provided port.used	port.kind='Input' port.kind='Output'
	componentAssembly.connection.port	compositeComponent.subComponent.port
Patrons de navigation	componentAssembly.connection	compositeComponent.configuration .binding.role.connector
	componentAssembly.connection .port.component	compositeComponent.subComponent

TAB. 7.2 – Projection de concepts entre le profil CCM et le profil standard

(SystemeControleAcces), ou les opérations de collections comme asSet() ou size().

Vous remarquez que certaines des règles mentionnées ci-dessus sont ambiguës. Elles ont la même condition d'application, comme interfaceInstance.direction dans le tableau 7.1. Ce qui distingue ces deux règles est leur contexte. Dans cet exemple, le contexte de la première règle est *component* et la seconde *connector*. Comme précisé précédemment, lors de la sérialisation de l'arbre syntaxique des contraintes, des informations de types sont stockées. Ces informations concernent le contexte de l'unité syntaxique en question. Selon la valeur de cette information (component ou connector, dans l'exemple), la règle de transformation à appliquer est choisie.

Certaines vérifications de type sont faites pour les opérations de collections, lors de la transformation. Si une opération de collection est retrouvée et que celle-ci ne correspond pas au type de l'unité syntaxique générée à laquelle elle s'applique, cette opération est remplacée par son équivalent ou supprimée. Par exemple, si on génère à partir d'un multi-ensemble (BAG) de composants un ensemble simple (SET), et que l'opération à transformer est asSet(), celle-ci est remplacée par une unité syntaxique vide. En effet, le rôle de cette opération est de transformer un multi-ensemble en un ensemble simple. Cette opération n'a donc aucun sens lorsqu'elle est appliquée à un ensemble simple.

Dans cette approche, il est intéressant de recourir à des techniques de simplification des contraintes [56, 21]. En effet, les contraintes, générées après transformation vers le profil standard, contiennent parfois beaucoup de redondance (qui n'influent pas la sémantique de la contrainte). Des contraintes, dans ce cas, plus simples (et sémantiquement équivalentes) peuvent remplacer ces dernières. Il est évident que les techniques de simplification des contraintes sont utilisées pour une meilleure compréhension de celles-ci. Ici, la simplification vise simplement à obtenir une évaluation optimale (avec un temps d'exécution minimal) des contraintes.

La méthode de transformation de contraintes proposée ci-dessus ne constitue pas la solution optimale pour une transformation haut niveau de contraintes architecturales. En effet, les étapes coûteuses de compilation, de sérialisation et de désérialisation ne constituent que des étapes supplémentaires qui ajoutent un coût au développement et à l'exécution de l'évaluation des décisions architecturales. En plus, XSLT est un langage bas niveau dédié uniquement à la transformation de documents semi-structurés. Il est plus judicieux de formaliser les projections, entre les différents profils et le profil standard, à l'aide d'un langage de règles ou un langage haut niveau dédié à la transformation de modèles [51].

7.4.4 Transformation des descriptions d'architecture

Comme précisé précédemment, après transformation des contraintes et avant leur évaluation, les descriptions d'architecture auxquelles sont associées ces contraintes sont transformées. Ces descriptions sont donc traduites vers un format intermédiaire conforme à ArchMM. Dans l'annexe C, le XML Schema d'ArchMM est fourni. Tout comme la transformation des contraintes, la transformation des descriptions d'architecture est réalisée avec le langage XSL. Donc, l'approche proposée ne prend en compte que les descriptions d'architecture basées sur XML.

Il est évident que certains concepts présents dans ArchMM ne sont pas présents dans les méta-modèles des différents ADL et technologies de composants. Des concepts avec des valeurs par défaut sont donc générés lors de la transformation. Par exemple, lorsqu'on transforme une instance d'interface xAcme, un port est généré ayant comme propriétés celles de l'inter-

face. D'autres exemples d'abstractions architecturales générées lors de la transformation vers ArchMM ont été discutées dans la section 7.3.6.

7.5 En résumé

Les décisions architecturales doivent être explicites tout au long du processus de développement d'un logiciel, car elles forment la documentation de ce dernier. Ces décisions doivent être formelles également pour qu'elles puissent être vérifiées dans toutes les étapes de ce processus. Cette traçabilité des décisions architecturales garantit la non-altération des propriétés de qualité que ces décisions implémentent, durant le développement ou l'évolution du modèle produit à une étape donnée.

Afin de rendre explicites et formelles ces décisions architecturales, j'ai présenté, dans ce chapitre, une approche basée sur une famille de langages de contraintes et une méthode de transformation. Cette approche de traçabilité des décisions architecturales propose d'utiliser ACL afin de formaliser les décisions dans les différentes étapes du processus de développement. Elle fournit également une méthode basée sur la transformation, à une étape donnée, de toutes les décisions formalisées en amont vers un profil ACL générique appelé le profil standard. Ce profil fournit un méta-modèle générique englobant les abstractions architecturales qui existent dans les ADL existants, UML 2 et les technologies de composants.

Il est évident que pour l'expression de certaines contraintes, l'utilisation d'ACL s'avère assez compliquée. Nous avons simplifié ceci par l'ajout d'une librairie d'attributs et d'opérations liés aux propriétés de graphes. L'une des ouvertures de cette thèse est l'étude et la proposition de librairies de modèles de contraintes récurrents, comme les styles architecturaux et les patrons de conception (voir chapitre 10).

Chapitre 8

Prototypes pour l'implémentation des approches

Sommaire

8.1 Introduction à l'implémentation des approches

Afin de valider le travail présenté dans ce mémoire, j'ai développé un certain nombre d'outils prototypes. Ces outils permettent globalement :

- d'assister à l'édition des contraintes architecturales ACL,
- d'assister à la construction des contrats d'évolution,
- d'évaluer les contrats d'évolution lors de l'évolution à la demande du développeur,
- de notifier le développeur si une évolution affecte les décisions architecturales et les propriétés non-fonctionnelles correspondantes,
- de mettre à jour les contrats après les réactions des développeurs selon l'algorithme d'assistance,

Par ordre chronologique, *ACE* (*Architectural Constraint Evaluator*) a d'abord été développé comme extension du compilateur OCL (OCL Compiler [36]). ACE permet d'évaluer les contraintes architecturales écrites dans le langage ACL. Au commencement, ACE n'interprétait que les contraintes écrites dans le profil ACL pour Fractal. L'outil a été, ensuite, amélioré pour prendre en compte le profil standard. Puis, *AURES* (*ArchitectuRe Evolution aSsistant*) a été

développé, au dessus d'ACE, pour automatiser l'assistance à l'évolution. Pour des raisons de séquencement logique entre les chapitres, je présente d'abord AURES puis, je raffine son architecture, pour présenter ACE.

8.2 AURES : un outil pour l'assistance à l'évolution

Afin de valider l'approche présentée dans le chapitre 6, un outil baptisé AURES a été développé. Ce prototype permet l'édition, la validation et l'évaluation des contrats d'évolution. De plus, il permet d'assister le développeur lors de l'activité d'évolution. Je présenterai dans les sections suivantes la structure et le fonctionnement de cet outil. L'outil a été développé en JAVA avec les JDK 1.4, puis 5.0 sous l'environnement Eclipse. Une partie de l'outil concernant les technologies XML, XML Schema et XSLT a été développée sous l'environnement $< oXygen/ >$[1].

8.2.1 Architecture d'AURES

Au sein de l'équipe où cette thèse a été effectuée, nous projetons, dans un avenir proche, de construire les outils en composants. Le choix de la technologie d'implémentation n'est pas encore fait. Par contre, il est fort probable qu'il soit développé dans le modèle de composants Fractal. J'ai choisi, dans ce mémoire, de présenter l'architecture d'AURES à l'aide de composants. Il me semble que le lecteur s'est déjà habitué, à travers les chapitres précédents, à ce genre de notations. La Figure 8.1 montre l'architecture de cet outil. Elle est composée des éléments suivants :

EC Editor : Ce composant permet l'édition de contrats d'évolution. Il utilise le format XMI des méta-modèles des différents profils pour guider le développeur à éditer ses contraintes architecturales. Il demande au développeur de compléter l'information nécessaire (relative aux propriétés non-fonctionnelles) afin de compléter et de générer le contrat.

EC Validator : Ce composant valide un contrat avec une description architecturale introduite. Si l'évaluation du contrat retourne la valeur fausse, le développeur est contraint de modifier ou bien le contrat, ou la description architecturale. Autrement, ce composant produit une archive composée de la description architecturale, du contrat d'évolution ainsi qu'un fichier de configuration XML. Ces archives sont gérées par le composant gestionnaire de versions (VersionHandler).

Evolution Assistant : Lorsqu'une nouvelle version de l'architecture est soumise, le contrat d'évolution de la version précédente est récupéré grâce au composant gestionnaire de versions. Ce contrat est évalué sur la nouvelle description architecturale. Si l'évaluation retourne la valeur vraie, la nouvelle description architecturale est associée au contrat, ensuite elle est stockée par le gestionnaire de versions. Sinon, les AD altérées et par conséquent les propriétés non-fonctionnelles qui lui sont associées sont notifiées au développeur. Cette notification spécifie également les éléments architecturaux concernés.

[1] Site Web d'oXygen : http ://www.oxygenxml.com/

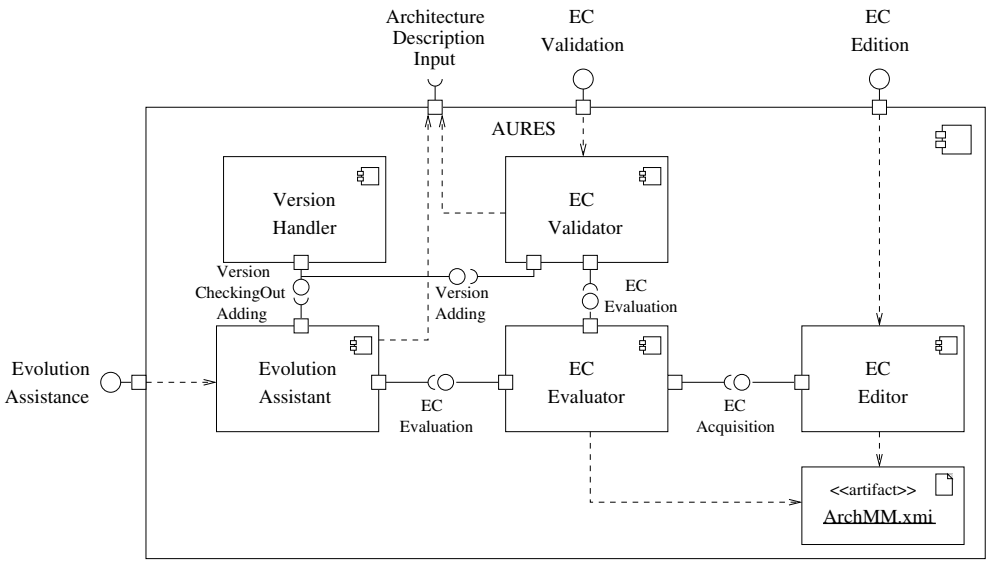

FIG. 8.1 – Un prototype pour l'assistance à l'évolution

Le développeur a le choix entre modifier la description architecturale, directement à travers l'interface d'AURES et sauvegarder les changements, ou bien en utilisant l'environnement de conception de l'ADL ou la technologie de composants en question. Ensuite, la description architecturale ou la configuration de composants peut être chargée dans AURES. L'outil permet également de modifier le contrat tout en préservant la règle 1 de l'algorithme d'assistance stipulant qu'à chaque NFP dans le contrat, il existe au moins une AD qui lui est associée. L'assistant d'évolution implémente l'algorithme d'assistance présenté dans le chapitre 6.

EC Evaluator : Afin d'évaluer les contrats d'évolution, ce composant utilise le modèle intermédiaire et le méta-modèle ArchMM. Le modèle intermédiaire est une instance de ArchMM, produite par le sous-composant *Description Transformer* (voir la Figure 8.2). Ceci est obtenu à partir de la description architecturale originale (dans l'ADL original). Ce composant lance l'interprétation d'un ensemble de fichiers XSLT pour effectuer des transformations XML des descriptions. Le prototype présenté dans cette thèse supporte des architectures décrites dans l'ADL Fractal et dans xAcme. Cependant, comme précisé précédemment, le modèle intermédiaire rend l'outil facilement extensible.

L'évaluateur de contrats possède ACE comme composant noyau. Ce composant permet d'évaluer les contraintes architecturales (la partie AD des contrats). Il sera détaillé dans la section suivante. Un autre composant (*Coor-dinatorReportGenerator*) permet de générer le rapport d'erreurs au cas où l'architecture ne serait pas conforme au contrat. Le composant évaluateur de contrats est utilisé à la fois par le composant *EC Validator* et *Evolution Assistant*. Quand il est invoqué par le premier, seules les contraintes impliquant une version de l'architecture sont évaluées. Par contre, lorsqu'il est invoqué par

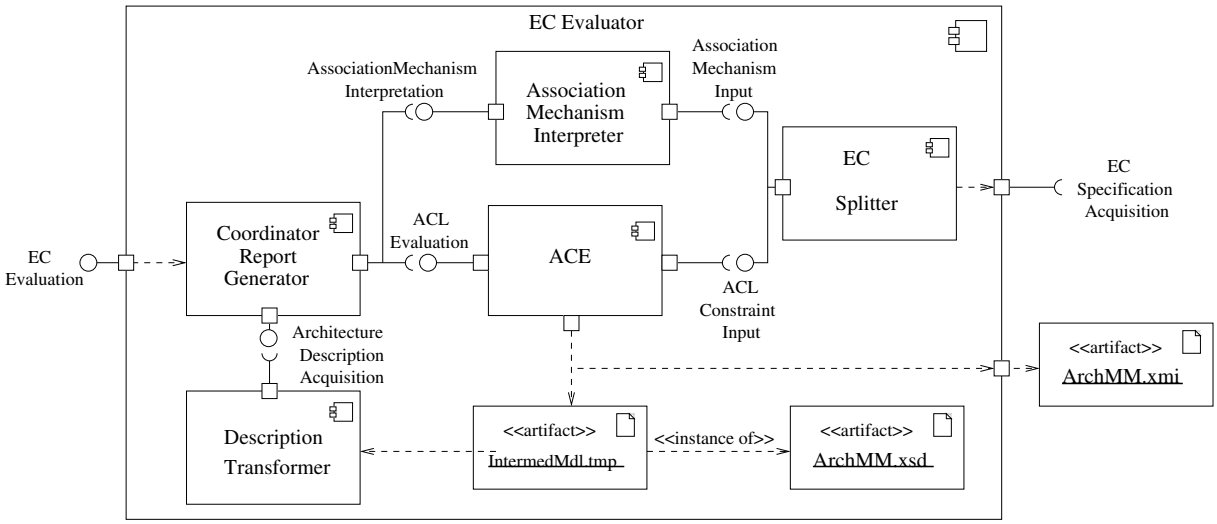

FIG. 8.2 – L'évaluateur de contrats d'évolution

le second, toutes les contraintes sont évaluées. La dernière description architecturale est récupérée avec son contrat. Elle est transformée vers le modèle intermédiaire pour être évaluée.

8.2.2 Fonctionnement d'AURES

Dans la Figure 8.3, une capture d'écran de l'interface graphique d'AURES est présentée. Sur le menu de gauche, on retrouve la liste de projets déjà ouverts. AURES, tout comme Eclipse gère les différentes version d'une architecture et de son contrat sous la forme de projets. Un projet représente une architecture donnée. Il mémorise plusieurs versions de sa description, de son contrat et un fichier de configuration. En sélectionnant un projet, la dernière version de cette architecture et son contrat sont chargés. La description de l'architecture est chargée dans l'espace au milieu de l'interface. Le contrat d'évolution est affiché sur la droite de la figure. Un contrat est visualisé tactique par tactique (NFT par NFT). L'interface permet de mettre à jour l'architecture directement et de sauvegarder les changements. Elle permet également de mettre à jour le contrat. On peut ajouter, supprimer ou modifier des propriétés non-fonctionnelles ou des contraintes architecturales (les boutons sous forme de plus et de croix). On peut également ajouter ou supprimer des tactiques. Le bouton *evaluate* permet de valider un contrat avec une description d'architecture. Il permet également d'évaluer un contrat sur une nouvelle description d'architecture, représentant l'évolution de cette dernière. Dans les deux cas, un rapport est généré. Il peut s'agir d'un rapport d'erreur ou d'une notification comme quoi l'évolution respecte le contrat.

Considérons un des scénarios d'évolution présentés dans le chapitre 1. Celui-ci concerne l'ajout d'un composant représentant un service de notification. Une fois que cette nouvelle description architecturale est introduite, le composant assistant d'évolution évalue le contrat

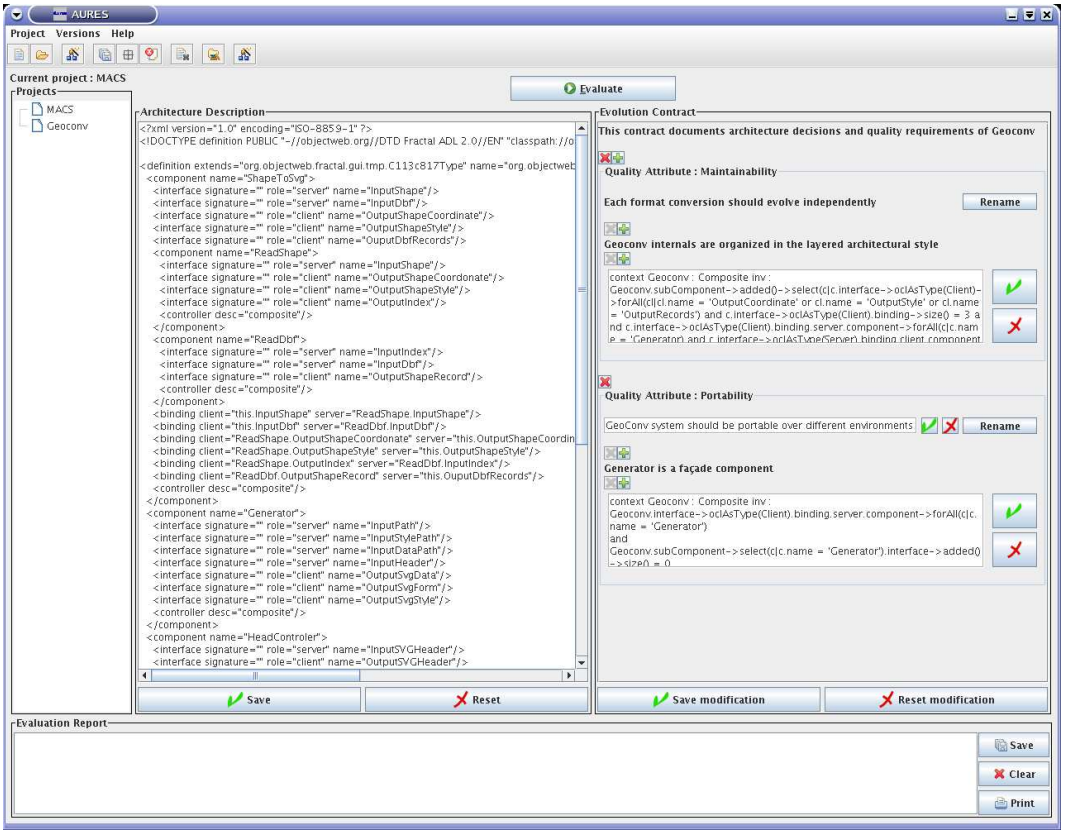

FIG. 8.3 – Capture d'écran de l'outil AURES

correspondant. Il notifie par le biais d'un rapport affiché et pouvant être stocké (voir les boutons dans le bas de la Figure 8.3). Ce rapport notifie à l'architecte de l'application, que le patron façade n'est plus respecté, et que l'attribut qualité de portabilité a été affecté (voir la Figure 8.4). L'architecte doit donc modifier la nouvelle architecture, ou bien maintenir son choix. Dans ce dernier cas, le contrat doit être mis à jour. Cet exemple simple montre l'assistance fournie par AURES à un développeur lors de l'évolution d'une architecture donnée.

8.3 ACE : un outil pour l'évaluation des AD

Le second outil développé avant AURES permet d'évaluer les contraintes architecturales. A la base, l'outil fournit une interface graphique et offre bien plus de fonctionnalités que le simple fait d'être un sous-composant d'AURES. Dans cette section, l'évaluation des contraintes avec ACE est présentée. Les autres fonctionnalités sont également introduites en parallèle.

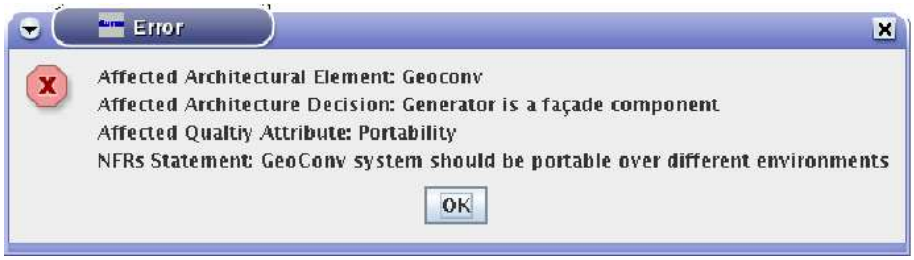

FIG. 8.4 – Capture d'écran du rapport d'erreur

8.3.1 Architecture d'ACE

Dans sa version actuelle, ACE permet l'évaluation des contraintes architecturales dans les profils ACL pour xAcme à l'étape de conception et pour Fractal à l'étape d'implémentation. L'architecture d'ACE est illustrée dans les figures 8.5 et 8.6. Pour des raisons d'espace, l'architecture a été découpée en deux parties. Le trait en pointillé représente la section de découpage. Cette architecture est composée des éléments architecturaux suivants :

Constraint Editor : Une fois le profil choisi (méta-modèle xArch ou Fractal), ce composant utilise le format XMI de ce méta-modèle pour guider le développeur dans l'édition de ces contraintes. Ce même composant est utilisé dans AURES.

Constraint Transformer : Une fois les contraintes écrites utilisant l'un des deux profils, elles sont transformées vers le profil standard pour être évaluées. Ce composant génère donc des contraintes qui naviguent dans ArchMM. Cette transformation est effectuée selon la méthode proposée dans le chapitre précédent. Ce composant compile d'abord les contraintes pour obtenir leurs arbres syntaxiques (AST). Ensuite, il sérialise les AST en documents XML. Il lance, ensuite, l'exécution d'un ensemble de feuilles de style XSLT pour transformer les documents XML générés en d'autres documents XML. Ces derniers représentent les contraintes dans le profil standard. Ces documents sont ensuite désérialisés pour extraire les contraintes dans leur forme textuelle. Les feuilles de styles XSLT représentent les projections entre les méta-modèles xArch et ArchMM, ainsi qu'entre Fractal et ArchMM.

Model Transformer : Ce module, déjà présenté dans AURES, génère une description intermédiaire conforme à ArchMM à partir d'une description d'architecture xAcme ou Fractal. Le méta-modèle ArchMM a été défini sous la forme de XML Schema (fourni dans l'annexe C). Le format intermédiaire est conforme à ArchMM. Il est donc basé sur XML. Le transformateur de descriptions exécute un ensemble de feuilles de style XSLT pour transformer les descriptions d'architecture xAcme et Fractal en des descriptions intermédiaires. Les feuilles de styles utilisées ici et les feuilles de styles utilisées par le transformateur de contraintes ne sont pas les mêmes. En effet, les documents XML transformés ne sont pas conformes au même XML Schema. Dans le premier cas, les documents XML sont conformes aux XML Schemas des méta-modèles xArch et Fractal,

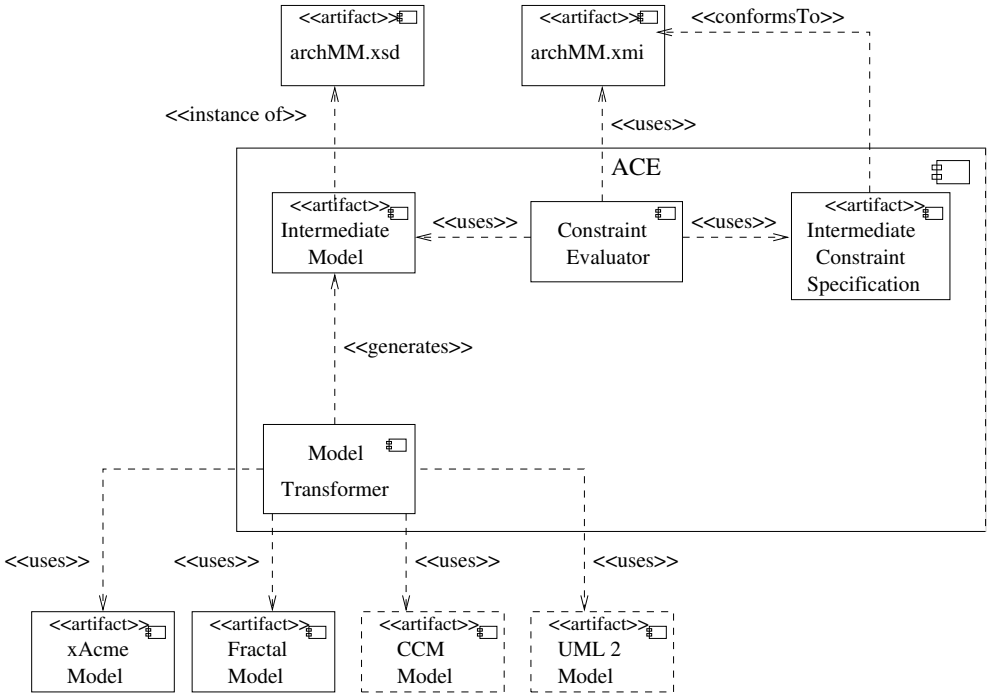

FIG. 8.5 – Un prototype pour l'édition et l'évaluation des contraintes -Partie 1-

alors que dans le dernier cas, les documents sont conformes au XML Schema fourni dans l'annexe D.

Constraint Evaluator : Ce composant est le coeur d'ACE. Il utilise le modèle intermédiaire et ArchMM pour évaluer les contraintes dans le profil standard (voir la Figure 8.5). Il est composé d'un compilateur pour ACL qui permet de générer des arbres syntaxiques abstraits pour les contraintes. Comme expliqué précédemment, ce compilateur a été construit au dessus d'OCL Compiler. Le méta-modèle est utilisé pour vérifier si les contraintes naviguent correctement dans sa structure. L'apparition d'erreurs syntaxiques est notifiée au développeur. Si aucun erreur n'est détectée, le processus d'évaluation commence. Si l'évaluation retourne la valeur vraie, la description architecturale est considérée conforme aux contraintes. Dans le cas contraire, le développeur doit corriger les contraintes ou la description architecturale. Cette dernière doit être conforme aux contraintes pour que ACE la valide. Comme précisé précédemment, à ce niveau, seules les contraintes impliquant une version sont évaluées (pas celles avec le @old, added() ou les autres opérations).

8.3.2 Fonctionnement d'ACE

Supposons une contrainte définie lors de l'étape de conception avec le profil pour xAcme. Pour que cette contrainte soit évaluée lors de l'implémentation (dans ce cas, sur une description

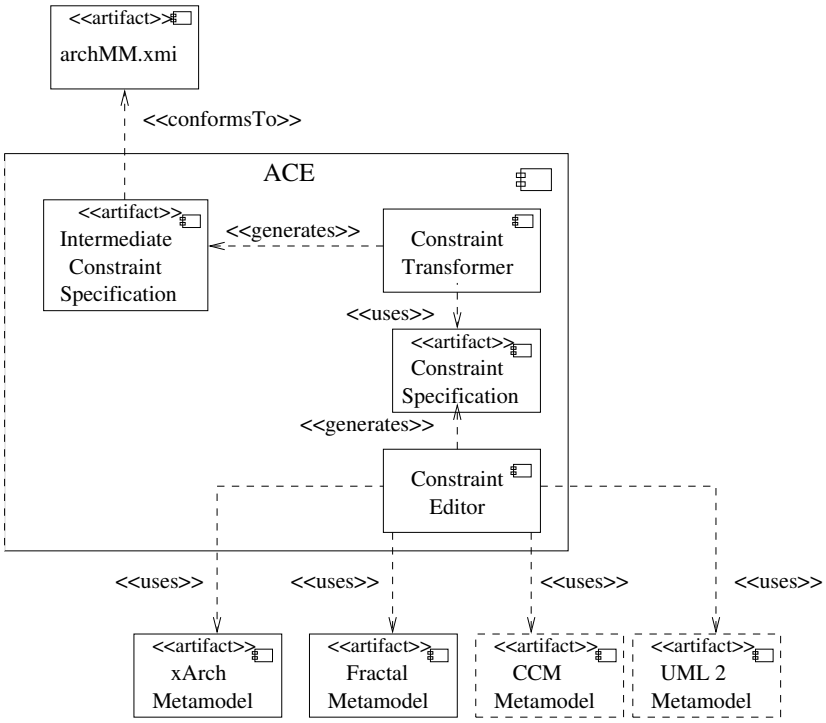

FIG. 8.6 – Un prototype pour l'édition et l'évaluation des contraintes -Partie 2-

en Fractal), les étapes suivantes sont effectuées :

1. transformation de la contrainte spécifiée dans le profil ACL pour xAcme vers le profil standard ;
2. transformation de la description architecturale en Fractal dans le modèle intermédiaire ;
3. évaluation de la contrainte transformée sur le modèle intermédiaire.

Sur la Figure 8.7, on retrouve une capture d'écran de l'outil ACE. Dans cette interface, on retrouve de simples zones de textes pour le chargement et éventuellement l'édition de contraintes (zone en haut à droite) et les descriptions d'architecture (zone en haut à gauche). En bas de la figure, on retrouve les résultats d'évaluation des contraintes architecturales sur les descriptions d'architecture.

Durant une activité d'évolution, une nouvelle version d'une architecture peut être introduite. Les contraintes sont évaluées sur cette nouvelle description transformée dans le modèle intermédiaire. Dans ce cas, toutes les contraintes (même celles impliquant les deux versions) sont évaluées. Si l'évaluation retourne la valeur vraie, la nouvelle architecture introduite est validée ; sinon, le développeur en est averti.

Selon ce schéma de conception, l'outil ACE peut être facilement étendu pour supporter d'autres ADL ou des technologies de composants (comme CCM ou UML 2, voir les boîtes avec pointillés dans les Figure 8.5 et 8.6). Il évalue toujours les contraintes sur la description

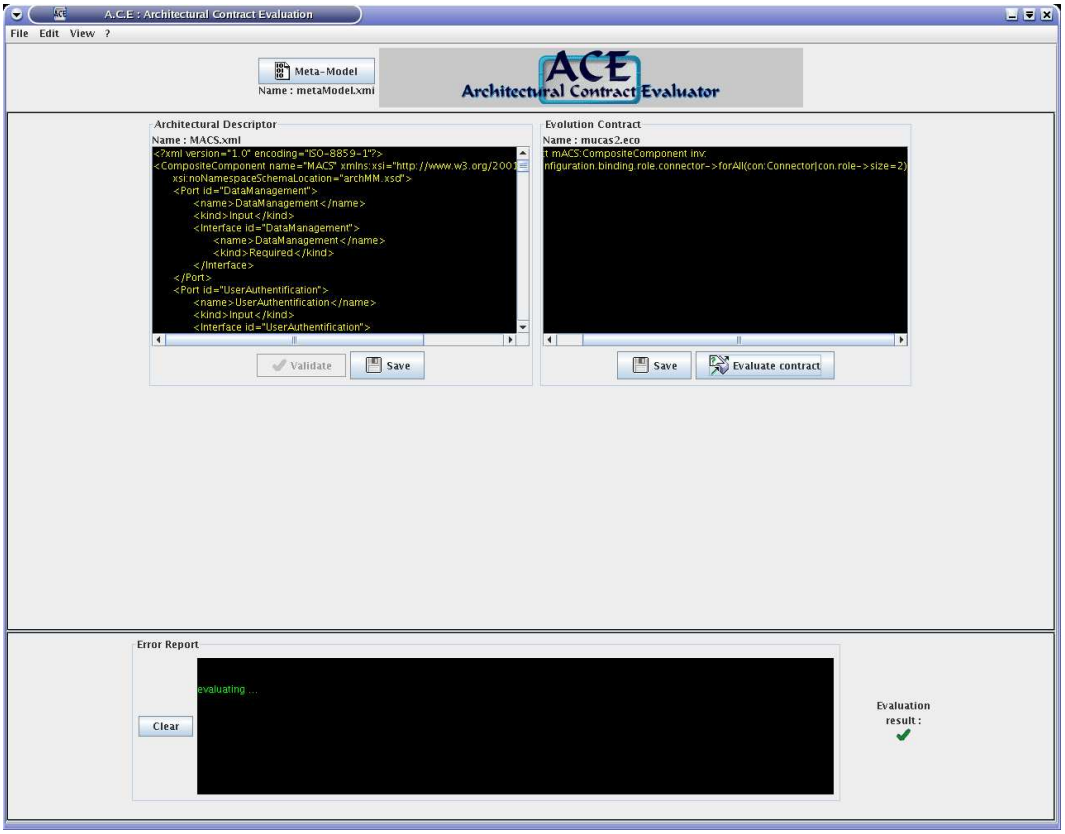

FIG. 8.7 – Capture d'écran de l'outil ACE

intermédiaire. Pour chaque nouvel ADL ou technologie de composants, il suffit de définir un méta-modèle (sous la forme XMI et XML Schema), ainsi que des projections (XSLT) entre ce méta-modèle et ArchMM, à la fois pour les descriptions d'architecture et les contraintes. Dans un avenir proche, je pense proposer un moyen pour décrire les projections entre un méta-modèle et ArchMM de manière unique. Ensuite, ces projections seront exploiter pour transformer, à la fois, les descriptions d'architecture et les contraintes. L'introduction d'un nouveau méta-modèle et de projections permet de générer un nouveau profil ACL. Seuls les transformateurs de contraintes et de descriptions doivent être étendus pour prendre en compte ses nouveaux formats. Les changements à faire sont donc minimes.

8.4 En résumé

Dans ce chapitre, des outils pour implémenter le travail ont été présentés. L'outil ACE implémente l'approche présentée dans le chapitre 7. Il contribue à la résolution de la problématique de préservation des décisions architecturales tout au long du processus de développement d'un logiciel à base de composants. En effet, ACE permet d'évaluer des contraintes, formalisant

ces décisions écrites dans différents profils ACL. Il transforme ces contraintes en un profil standard évalué par cet outil. Il transforme également les descriptions d'architecture ou les configurations de composants vers une représentation intermédiaire. C'est sur cette représentation que les contraintes écrites dans le profil ACL standard sont évaluées. Dans sa version actuelle, ACE permet d'évaluer des décisions architecturales prises, au niveau conception, sur des descriptions architecturales xAcme. Il permet également d'interpréter des décisions prises, au niveau implémentation, sur des descriptions de composants Fractal.

AURES, l'autre outil présenté dans ce chapitre, exploite les fonctionnalités d'ACE pour l'évaluation des décisions architecturales écrites dans différents profils ACL. Il permet d'éditer les contrats d'évolution, qui contiennent la description des décisions architecturales. De plus, les contrats contiennent les liens de ces décisions avec les propriétés non-fonctionnelles (attributs qualité) qu'elles garantissent. Ces contrats sont interprétés par AURES lors de l'évolution d'une description d'architecture ou d'une configuration de composants. Cette interprétation de contrats permet de notifier au développeur qu'un changement architectural affecte potentiellement les décisions architecturales évaluées dans le contrat. En suivant les liens définis dans ces contrats, l'outil notifie également l'altération probable des propriétés non-fonctionnelles correspondantes. Le développeur a le choix entre modifier la description architecturale, le contrat, ou laisser ces modifications telles qu'elles. L'outil garantit le respect des règles de l'algorithme d'assistance. AURES implémente l'approche présentée dans le chapitre 6 et répond donc à la problématique de préservation des décisions architecturales et des propriétés non-fonctionnelles lors de l'évolution.

Dans cette partie, le coeur de cette thèse a été détaillé. Une approche pour la documentation des architectures logicielles sous la forme de contrats a été proposée. Ces contrats permettent d'exprimer de manière formelle les décisions de conception prises par les développeurs au niveau architectural. Ils permettent également d'expliciter les raisons derrière ces décisions. Cet aspect couvre les propriétés non-fonctionnelles (et plus particulièrement) les attributs qualité requis par les documents de spécification.

Les contrats d'évolution servent, bien plus qu'une simple documentation. Lors de l'évolution, ces contrats sont utilisés afin de notifier les développeurs des conséquences d'un changement sur les décisions architecturales et sur les propriétés non-fonctionnelles correspondantes. Ceci permet de mieux contrôler l'évolution et guider le développeur, lors d'une évolution, vers une version de l'architecture qui ne dérive pas de manière importante des besoins non-fonctionnels initiaux. Cette assistance vise à minimiser le coût de maintenance à travers la réduction des tests de non-régression effectués sur l'aspect non-fonctionnel.

Les décisions architecturales dans ces contrats sont formalisées sous la forme de contraintes. Ces contraintes sont exploitées pour tracer les décisions architecturales tout au long du processus de développement (avant l'évolution). Cette traçabilité est garantie grâce à un langage fournissant plusieurs profils, et une méthode de transformation de contraintes. L'approche présentée est extensible. Elle permet de prendre en compte de nouveaux ADL ou technologies de composants avec un effort minimal.

Ces différentes approches ont été implémentées à travers un environnement (AURES) et un outil (ACE). Ces prototypes sont assez simples d'usage et peuvent être facilement étendus.

Quatrième partie

Conclusion

Ce mémoire retrace le travail effectué dans le cadre d'une thèse sur la contractualisation de l'évolution. L'évolution se situe ici au niveau des architectures de logiciels à base de composants. Cette partie est structurée comme suit. Dans le premier chapitre, les apports de cette thèse en terme de concepts, puis en terme de langages et de techniques, seront présentés. Chaque point sera détaillé en se basant sur les différentes parties du travail présenté dans ce mémoire. Ensuite, ces apports seront répartis sur les différentes publications et communications établies pendant la thèse. Dans le second chapitre, les améliorations possibles de ce travail seront introduites. Ces améliorations nécessitent un travail important et requérant un temps supplémentaire que le temps imparti à cette thèse n'a pas permis. Enfin, un chapitre épilogue clôturera ce mémoire.

Chapitre 9

Apports de la thèse

Sommaire

Cette thèse apporte des contributions tant sur le plan conceptuel que sur le plan ingénierie. En premier lieu, je présenterai les principaux concepts introduits par ce travail, au monde du génie logiciel. Ensuite, je détaillerai l'apport de cette thèse sur le plan de l'ingénierie.

9.1 Sur le plan conceptuel

Sur ce plan, les travaux de cette thèse contribuent aux méthodologies du développement logiciel sur plusieurs aspects. Ces aspects varient de la proposition de concepts, comme l'explicitation et le maintien de liens entre décisions architecturales et propriétés non-fonctionnelles, à la proposition du concept de contrat d'évolution ; jusqu'à l'usage des contrats pour une assistance à l'évolution dirigée par la qualité ; et enfin la traçabilité des décisions tout au long du cycle de vie d'un logiciel à base de composants.

9.1.1 Explicitation et maintien des liens entre AD et NFP

Beaucoup de travaux dans la littérature affirment que les décisions architecturales sont déterminées principalement par les propriétés non-fonctionnelles (attributs qualité) exprimées

dans les spécifications de besoins [11], [77], [26]. Il existe, dans la littérature, des travaux qui relient de manière explicite les décisions architecturales aux propriétés non-fonctionnelles [155], [25] et [116]. Ces liens sont dans le cas de [155] et [25] purement informels et se limitent à la simple compréhension d'une architecture, comme une simple documentation. Aucun de ces travaux n'a exploré la voie de la formalisation de ces liens, à part [116]. Cependant, ces liens ne sont exploités que lors de la conception et ne sont pas maintenus lors de l'évolution. Les liens entre des décisions architecturales exprimées de manière formelle et les propriétés non-fonctionnelles ont permis l'automatisation de la vérification de certaines propriétés d'un logiciel avant son évolution afin de fournir une certaine assistance. Ces liens, formant le contrat, sont maintenus et mis à jour lors de l'évolution.

Cet apport a été publié dans [44], [150] et [152].

9.1.2 Notion de contrat pour l'évolution

Dans cette thèse, le concept de contrat d'évolution a été introduit. Ce contrat représente un accord entre l'architecte d'un système et la personne qui fait évoluer l'architecture. Cet accord oblige, d'une part, l'architecte à documenter formellement les liens discutés ci-dessus. En contre partie, il bénéficie de la préservation ultérieure de la qualité de son architecture. D'autre part, lorsqu'il applique ses changements, le développeur est obligé de se conformer à cet accord, afin de bénéficier de la qualité initialement établie.

Ce concept de contrat d'évolution est original dans le sens où il n'existe pas, dans la littérature, de concepts ayant le même objectif. Comme précisé précédemment, les contrats d'évolution dans [106], permettent de détecter les incohérences dans une hiérarchie de classes lors de l'évolution. Ces contrats sont rédigés en partie par les concepteurs et en autre partie par celui qui fait évoluer le document de conception. Ils sont mis à jour à chaque évolution. Les contrats d'évolution introduits dans cette thèse sont entièrement rédigés par l'architecte et sont mis à jour, de temps à autre, par le développeur effectuant l'évolution. Comme détaillé dans le chapitre 5, l'objectif des deux contrats n'est plus le même.

Cette contribution a été publié dans [44] et [152].

9.1.3 Assistance à l'évolution dirigée par la qualité

Dans la littérature, il n'existe pas d'approche pour assister l'évolution à l'aide d'informations sur la qualité. Dans cette thèse, une approche pour l'assistance à l'évolution dirigée par des informations sur les décisions architecturales et sur les besoins qualité a été proposée. Le seul travail existant [46], discuté dans le chapitre 5, ne se base pas sur la notification. Il utilise la formalisation des propriétés non-fonctionnelles pour sélectionner la meilleure implémentation pour un composant dont les spécifications ont changé. On suppose qu'il existe déjà une librairie d'implémentations de composants. L'approche permet de choisir celui qui s'approche le plus de l'évolution de la spécification non-fonctionnelle.

L'approche d'assistance, présentée dans cette thèse, ne permet pas de choisir un composant donné pour une évolution particulière. L'évolution prise en compte est d'ordre architectural. Elle peut être la conséquence d'une évolution des spécifications fonctionnelles ou des spécifications non-fonctionnelles. L'assistance permet de notifier l'impact de l'évolution sur

les décisions architecturales et sur la qualité. Elle ne propose aucune solution existante dans une librairie. Par contre, elle permet d'alerter de l'impact sur la qualité qu'a une évolution des spécifications, aussi bien fonctionnelles que non-fonctionnelles.

Ce travail a été validé dans les publications [153] et [43].

9.1.4 Traçabilité des décisions architecturales

Dans cette thèse, j'ai proposé une approche qui vise à maintenir les décisions tout au long du cycle de vie d'un logiciel. Par exemple, une décision prise lors de la conception est maintenue lors de l'implémentation. L'avantage de ceci est la préservation des propriétés non-fonctionnelles, garanties par ces décisions, tout au long du développement. Ce développement est souvent réalisé par des équipes différentes d'une étape à une autre. Il est donc nécessaire d'expliciter les décisions prises à une étape donnée lors des étapes en aval. La préservation de ces décisions est faite de manière automatique dans l'approche proposée pour la traçabilité des décisions.

Cette contribution a été publiée dans [151].

9.2 Sur le plan de l'ingénierie

Sur ce plan, un langage de contraintes architecturales à deux niveaux a été proposé. Cette structuration en deux niveaux a permis de simplifier la transformation de contraintes écrites avec ce langage. Cette thèse a introduit également un méta-modèle générique pour les architectures logicielles et les technologies de composants. De plus, une nouvelle approche pour la documentation d'architectures logicielle a été présentée.

9.2.1 Langages de contraintes architecturales multi-niveaux

Dans cette thèse, le langage ACL a été proposé pour formaliser les décisions architecturales. Ce langage, dit bicéphale, s'articule autour de deux niveaux d'expression. Le premier niveau concerne l'expression basique de prédicats de premier ordre. Il est assuré par une version légèrement modifiée d'OCL. Le second niveau représente les abstractions architecturales (composants, interfaces, etc) contraintes par le premier niveau. Il englobe un ensemble de méta-modèles définis en MOF. Chaque couple, composé de CCL et d'un méta-modèle MOF, est considéré comme un profil ACL. Ce profil est considéré comme langage de contraintes à part entière. Chaque définition d'un nouveau méta-modèle introduit un nouveau langage de contraintes architecturales.

Cet apport a été validé par plusieurs publications. Dans ces anciennes versions (versions de base), ce langage de contraintes a été publié dans [44]. La nouvelle version de ce langage (incluant plusieurs profils et le support aux contraintes de pure évolution) a été publié dans [151].

9.2.2 Simplification de la transformation de contraintes

D'une étape à une autre du cycle de vie, les contraintes architecturales peuvent être exprimées avec différents profils ACL. Dans une approche d'ingénierie logicielle dirigée par

les modèles, ces contraintes doivent être transformées afin d'accompagner les modèles (descriptions d'architecture) auxquelles elles sont associées. Utilisant ACL, les contraintes, écrites d'une étape à une autre, ont en grande partie CCL en commun. Grâce à la séparation entre les deux niveaux, seules les projections entre méta-modèles sont définies pour implémenter les transformations. Ceci simplifie considérablement le travail de l'implémentation des transformations entre contraintes.

Cette contribution a été le sujet de la publication : [153].

9.2.3 Un méta-modèle générique pour les architectures et les composants

Dans cette thèse, un méta-modèle, appelé ArchMM, a été proposé. ArchMM englobe les abstractions architecturales présentes dans les langages de description d'architecture les plus référencés dans la littérature. Il encapsule également les concepts architecturaux présents dans le langage UML et les technologies de composants d'implémentation EJB, CCM et Fractal.

Ce travail a été publié dans [149] et révisé dans [151].

9.2.4 Nouvelle approche pour la documentation logicielle

Le travail présenté dans ce mémoire a fourni également une nouvelle approche pour la documentation d'architectures logicielles. Cette documentation peut être exploitée dans la compréhension d'une architecture avant son évolution. En effet, cette documentation permet de rendre explicite un certain savoir sur les justificatifs derrière des décisions architecturales (*rationale*). Naturellement, dans l'étape de compréhension (lors de l'évolution), ce sont les informations les plus recherchées [155], [25]. Comme précisé précédemment, cette documentation doit être formelle afin qu'elle soit non ambiguë. De ce fait, elle permet également l'assistance lors de l'évolution.

Cette approche de documentation a été validée dans les deux publications [152] et [43].

Cette thèse a plusieurs apports tant sur le plan conceptuel que sur le plan de l'ingénierie. Ces différents apports ont été validé par de multiples publications. Récemment, le langage ACL a été appliqué dans le domaine du déploiement de composants. Il a été utilisé pour la description des contraintes sur les ressources système et réseau et des contraintes sur la localisation où des composants vont être déployés. Ce travail a pour objectif de préserver les décisions architecturales et les contraintes de ressources et de localisation lors du déploiement. Les composants sont déployés dans des plate-formes, dites dynamiques. Ces plate-formes ont pour caractéristique que les noeuds où sont déployés les composants, peuvent s'arrêter et donc deviennent inaccessibles. Ils peuvent ensuite se reconnecter de manière aléatoire. Cette extension de la thèse, investiguée en collaboration avec des collègues, travaillant dans ce domaine (déploiement de composants), a été validée dans [62].

Chapitre 10

Améliorations possibles

Sommaire

Les approches présentées dans ce mémoire peuvent être améliorées sur différents plans. Ces améliorations constituent des pistes de recherches qui nécessitent plus de réflexion et d'investigation. Le temps imparti à la réalisation de cette thèse n'a pas permis d'approfondir mes recherches dans ces voies-ci. Les pistes d'amélioration concernent d'abord la simplification de la rédaction des contrats. Cette extension possible sera détaillée dans la sous-section suivante. Des améliorations pour obtenir une assistance à l'évolution plus raffinée seront ensuite introduites. Une piste de recherche sur la transformation directe des contraintes entre profils sera abordée. Cette amélioration aide à la traçabilité des décisions architecturales dans le processus de développement d'un logiciel à base de composants. La prise en compte de l'aspect comportemental d'une architecture sera présentée à la fin de ce chapitre.

10.1 Simplification de la rédaction des contrats

Pour le moment les outils développés permettent d'assister l'écriture de contraintes en proposant les navigations possibles dans les méta-modèles sur lesquels sont écrites ces contraintes. Il est envisageable d'étendre l'outil avec des fonctionnalités afin d'assister les développeurs non

formés à ACL dans l'écriture des contraintes. Cette extension permettra, entre autre, de proposer les différentes parties d'une contrainte ACL au développeur afin qu'il puisse les remplir.

L'autre volet qu'on pourra explorer pour simplifier la rédaction des contrats passe par la proposition de librairies. Ces librairies peuvent être des catalogues de décisions architecturales, comme les patrons de conception et d'architecture les plus utilisés. Il peut s'agir également de catalogues de tactiques non-fonctionnelles.

10.1.1 Catalogue des patrons de conception et d'architecture

Il est envisageable de proposer un catalogue représentant les patrons de conception. En effet, il est possible de proposer une librairie de contraintes ACL représentant les règles structurelles que doivent respecter les patrons de conception. En s'inspirant de la classification du GOF[1] [50] et en les adaptant au monde des composants, nous pouvons définir une liste de contraintes réutilisables. Ces dernières peuvent être référencées par le développeur afin de préciser qu'un certain nombre d'éléments de son architecture sont organisés sous la forme d'un patron de conception donné. Ceci lui simplifiera considérablement l'écriture du contrat. Nous pouvons envisager également de proposer un catalogue pour les patrons et les styles d'architecture. Comme le catalogue précédent, ce dernier permet de charger directement la liste de contraintes implémentant un patron d'architecture donné. J'ai imaginé, ci-dessous, et illustré un modèle de tels catalogues.

- **Description :** "Un système en couche est organisé hiérarchiquement. Chaque couche fournit des services à la couche au dessus et sert comme client à la couche au dessous" [141].
- **Propriétés non-Fonctionnelles :** Ce style assure la maintenabilité et la réutilisabilité.
- **Utilisation :**
  - **Signature :** Ce style est représenté au niveau méta-modèle sous la forme de l'opération suivante :
    layeredStyle(Collection couches, Integer maxConnecteurs)
    où :
    couches est une collection d'ensembles de composants.
    maxConnecteurs représente le nombre maximal de connecteurs entre les couches.
  - **Position dans les différents méta-modèles :** Cette opération peut être représentée dans les différents modèles des ADL, des technologies de composants et aussi dans ArchMM.

    Méta-modèle xAcme : Cette opération peut être placée dans la méta-classe *SubArchitecture*. Elle est interprétée dans le contexte de la sous-architecture d'une instance d'un composant.

    ArchMM : Cette opération peut être placée dans la méta-classe *Configuration*. Elle est interprétée dans le contexte de la configuration d'un composant composite.

    Méta-modèle CCM : Cette opération peut être placée dans la méta-classe *ComponentAssembly*. Elle est interprétée dans le contexte d'un assemblage de composants.

[1] *Gang Of Four* représente les quatre auteurs du livre sur les patrons de conception.

– **Implémentation :** Les contraintes assurant ce patron sont les suivantes :
  – Il existe au plus *maxConnecteurs* connecteurs entre les couches,
  – Il n'existe pas de connecteur entre couches non-adjacentes.
  Ces contraintes sont implémentées dans le profil standard ACL de la manière suivante :

```
-- Ici placer les contraintes éreprsentant le style
context X:...
```

Le style architectural illustré ci-dessus est le style en couches. Il peut être donc utilisé directement dans une contrainte par le biais de l'opération *layeredStyle(...)*. Les composants constituant les différentes couches sont passés en paramètre, ainsi que le nombre maximal de connecteurs qui peuvent être définis entre deux couches adjacentes. Par exemple, pour un composant donné (C), on peut écrire la contrainte suivante :

```
context C:ComponentInstance inv:
C.subArchitecture.layeredStyle(...)
```

Cette contrainte stipule que le composant C est organisé en architecture en couches. Pour des raisons de simplicité, la signature exacte de la contrainte n'est pas fournie. La contrainte ci-dessus sera évaluée grâce aux différentes contraintes sous-jacentes.

10.1.2 Guide pour les NFT récurrentes

Le précédent catalogue peut identifier les différents attributs qualité (propriétés non-fonctionnelles) garantis par les décisions architecturales. Il peut donc guider le développeur au moment où les contrats sont écrits, en lui proposant les grandes lignes pour faire des décisions architecturales qui correspondent aux attributs qualité requis. Les différentes AD sont classifiées par attribut. Le développeur aura à choisir celle (ou celles) qui correspondent le plus à ces besoins parmi les AD cataloguées.

Il est intéressant d'investiguer une voie de recherche visant à proposer des catalogues génériques paramétrés. En effet, il est possible d'imaginer des catalogues pouvant avoir des paramètres et être considérés comme des types. Il sera donc possible d'instancier un contrat à partir d'un catalogue en précisant les paramètres nécessaires. Ce contrat contiendra les éléments du catalogue choisis grâce aux paramètres. Ces paramètres peuvent représenter également les éléments architecturaux impliqués dans les contraintes sélectionnées.

10.2 Assistance plus raffinée à l'évolution

Pour le moment, l'assistance permet de notifier au développeur les informations sur les décisions architecturales affectées et les attributs qualité potentiellement altérés. Dans cette section, les différentes améliorations possibles à cette thèse seront discutées. Ces extensions possibles sont illustrées dans les Figures 10.1, 10.2 et 10.3.

10.2.1 Formalisation des relations entre ACG et AD

Au stade actuel du travail, l'environnement d'assistance à l'évolution reçoit simplement en entrée une nouvelle description d'architecture. Aucune information supplémentaire n'est fournie. L'élaboration d'un modèle pouvant expliciter à la fois le changement architectural et son objectif aide à la notification avec des informations plus pertinentes.

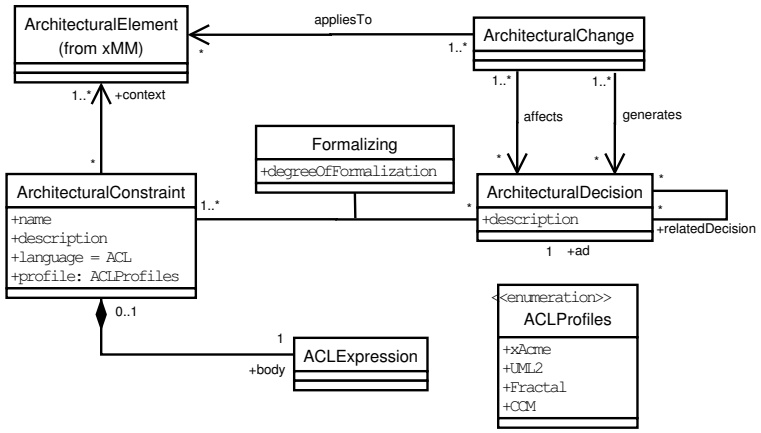

FIG. 10.1 – Documentation plus fine -Partie 1-

Supposons que le contrat d'évolution du composant MACS contient deux NFT qui partagent en commun l'AD4 (pipeline). La première NFT est liée à l'attribut qualité maintenabilité et la deuxième à l'attribut performance. Considérons le scénario d'évolution consistant à ajouter un lien entre les deux composants non-adjacents CtlAccesDup et EmetteurServeur. Ce scénario affecte réellement les deux NFT. Le système d'assistance notifie donc l'affectation possibles des deux attributs qualité (maintenabilité et performance). Normalement, le scénario d'évolution vise à améliorer les performances du composant MACS (en transmettant directement certaines données, dont l'archivage n'est pas important, vers le serveur central). L'information notifiée concernant l'attribut qualité performance n'est pas donc réellement pertinente. Dès lors l'établissement d'un lien entre un changement architectural et une décision architecturale, tel qu'illustré sur la Figure 10.1, aide à fournir des informations pertinentes lors de l'assistance à l'évolution. Les deux types de liens imaginables entre ces éléments sont les liens d'affection (*affects* dans la Figure 10.1) et les liens de génération (*generates*). Le premier type de liens concerne les ACG affectant positivement ou négativement une AD. Le second est lié aux ACG qui génèrent des décisions architecturales.

10.2.2 Formalisation des relations entre AD

Afin de faciliter l'expression des décisions architecturales avec ACL, il est intéressant de proposer des moyens de réutilisation des décisions. Il sera donc possible de référencer une décision architecturale existante à partir d'une autre décision. Celle-ci hérite les propriétés de l'autre. La relation d'héritage qu'on peut imaginer ici représente une simple relation de conjonction logique. Ceci est dû au fait que les décisions sont formalisées comme contraintes ACL, qui sont des expressions booléennes. Il faudra donc proposer des mécanismes de haut niveau pour établir des références aux contraintes ACL.

10.2.3 Formalisation des relations entre AD et AC

Pour le moment, seul l'aspect structurel d'une architecture a été abordé dans cette thèse. L'aspect comportemental n'est pas pris en compte. Il existe certaines décisions architecturales qui adressent en partie l'aspect structurel et également l'aspect comportemental. Disposant d'ACL, ces décisions ne sont formalisées qu'à un certain degré. Il est intéressant, dans ce cas, d'avoir un moyen pour préciser ce degré de formalisation (voir Figure 10.1). Ceci permet d'évaluer une contrainte ACL donnée et de savoir à quel degré une décision a été altérée. Il se peut que d'autres notations soient utilisées pour compléter la proportion manquante de formalisation d'une décision.

10.2.3.1 Raffinement des relations entre AD et NFP

Les mêmes remarques peuvent être faites par rapport au lien entre AD et NFP proposé dans cette thèse. Il est intéressant d'investiguer la possibilité d'ajouter un degré de satisfaction entre une AD et une NFP. Il est évident qu'une décision architecturale n'assure pas systématiquement une propriété de qualité [11]. Il est donc utile d'identifier à quel degré une AD garantit une NFP. Ceci est illustré sur la figure 10.2. Ce degré est représenté par la classe-association *Satisficing*.

Il se peut que le degré de satisfaction aie une valeur numérique ou un pourcentage. Dans ce cas, on peut imaginer la somme des valeurs des degrés, dans différentes NFT impliquant une même NFP, égale à l'unité (100 % dans le cas du pourcentage, par exemple). Le degré de satisfaction ne doit pas forcément avoir que des valeurs numériques. Il est possible de proposer un type énuméré, contenant des valeurs comme : élevée, moyenne, basse, etc.

10.2.4 Formalisation des relations entre NFT

Comme dans les approches existantes de documentation non-formelles des décisions architecturales [155], [25], il est possible de formaliser des relations entre NFT. Ces relations représentent les différentes alternatives envisageables par le développeur en cas d'affectation d'une NFT lors de l'évolution. Ces alternatives permettent de rendre explicite l'ensemble (ou une partie) des choix architecturaux possibles auxquels le développeur était confronté. Lorsqu'il a pris sa décision, le développeur a fait un choix et a donc éliminé les autres alternatives. Il se peut que, lors d'une évolution, si sa décision est affectée, il veuille restituer l'un des choix éliminés.

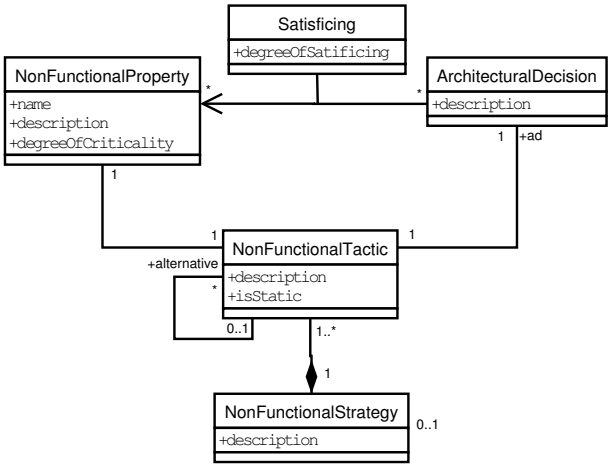

FIG. 10.2 – Documentation plus fine -Partie 2-

10.2.4.1 Formalisation des relations entre NFP

L'explicitation des relations entre NFP (et donc entre attributs qualité) constitue un aspect intéressant à investiguer. Par contre, un catalogue universel des liens entre attributs qualité ne peut être établi de manière permanente pour les modèles de qualité existants. En effet, il a été constaté dans [38] que ces dernières (les relations) peuvent changer lors du développement. Dans leur expérimentation, les auteurs ont démontré qu'au départ, certains attributs qualité n'étaient pas en conflit dans leur modèle générique de conflits et de coopération entre attributs. En d'autres termes, ces deux attributs peuvent évoluer indépendamment l'un de l'autre. Lors du développement d'un système particulier, il s'est avéré que ces attributs se chevauchent. Ceci permet de conclure qu'il sont inter-dépendants et qu'ils sont en conflit.

Ce qui est donc envisageable, est de créer, lors du développement, une liste de liens entre les attributs qualité implémentés dans le système. Nous pouvons ensuite exploiter ces liens lors de l'assistance. En effet, dans ce cas, il devient possible de suivre ces liens et de montrer au développeur la conséquence d'un changement, non pas sur une seule NFP, mais sur l'ensemble des NFP qui dépendent de celle directement affectée.

Théoriquement, deux types de liens entre NFP peuvent être identifiés. Ces types de liens sont illustrés sur la Figure 10.3. Supposons qu'on dispose de deux NFP (NFP1 et NFP2). Les types de relations entre NFP sont :

- Relations de type "collabore avec" (*collaborates with*) :
  - fortement couplée (relation de type 1) : si NFP1 est améliorée alors NFP2 est améliorée, et si NFP1 est affaiblie alors NFP2 est affaiblie également,
  - faiblement couplée (relation de type 2) : si NFP1 est améliorée alors NFP2 est améliorée, et si NFP1 est affaiblie alors NFP2 n'est pas affectée,
- Relations de type "se heurte contre" (*collides with*) :
  - fortement couplée (relation de type 3) : si NFP1 est améliorée alors NFP2 est affaiblie, et si NFP1 est affaiblie alors NFP2 est améliorée,

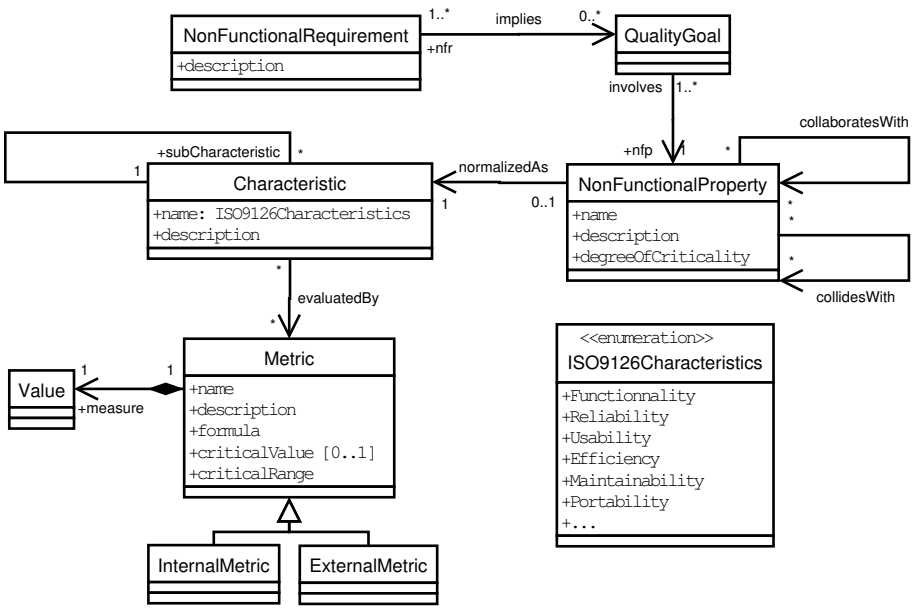

FIG. 10.3 – Documentation plus fine -Partie 3-

– faiblement couplée (relation de type 4) : si NFP1 est améliorée alors NFP2 est affaiblie, et si NFP1 est affaiblie alors NFP2 reste inchangée.

Les relations de type 1 et 3 (fortement couplée) sont les plus intéressantes dans le cadre de cette thèse. En effet, on peut suivre les liens représentant ces relations afin de mieux assister l'évolution. Lors de l'évolution, si une NFP est affectée, les relations de type 1 permettent d'alerter le développeur des NFP qui collaborent avec celle-ci et qui lui sont fortement couplées. Ces NFP sont également potentiellement altérées. Lors de l'évolution, si une NFP est affectée, les relations de type 3 permettent de notifier au développeur qu'une ou plusieurs autres NFP a ou ont été potentiellement améliorée(s). Il se peut que ceci aie un impact sur les spécifications du logiciel. Les relations 2 et 4 entre NFP (faiblement couplées) ne sont utiles que lorsque le développeur spécifie qu'un changement architectural vise à améliorer un attribut qualité donné (grâce à l'extension proposée dans la section 10.2.1). En effet, elles permettent de savoir quel impact a ce changement sur les attributs existants. Il existe logiquement un troisième type de relations entre NFP. Il s'agit des relations du type "n'affecte pas". Ces relations entre NFP ne sont pas d'un grand intérêt dans cette étude.

10.2.5 Assistance par quantification de la qualité

Les modèles de qualité existant (introduits dans le chapitre 2) spécifient une relation entre métriques et caractéristiques qualité (et donc NFP). Ceci est illustré sur la Figure 10.3. Il est donc possible de mesurer ces métriques avant et après l'évolution pour un attribut qualité potentiellement affecté. Ces valeurs sont donc notifiées au développeur afin qu'il aie une information

quantitative. Cette information permet de mieux raisonner sur un changement particulier. Il est évident que toutes les métriques ne peuvent pas être mesurées de manière statique. L'approche proposée dans cette thèse ne peut donc exploiter que les métriques pouvant être évaluées avant le déploiement et la mise en route de l'application. Ces métriques concernent principalement l'attribut qualité de maintenabilité.

Il devient donc possible, si l'attribut qualité de maintenabilité est affecté lors d'une évolution, de mesurer les métriques associées à cet attribut, comme les métriques de couplage [91]. Le développeur saura donc si le changement effectué sur l'architecture a un impact quantitatif important sur la maintenabilité.

Il est évident que certains de ces raffinements de la documentation proposés ci-dessus ont un coût supplémentaire. Celui-ci vient s'ajouter aux coûts de développement et de rédaction du contrat. Il est donc important, lors de l'investigation de ces améliorations, de mesurer les coûts associés à chacune. Ceci permet de voir si l'approche ainsi améliorée est rentable ou non. Par exemple, la définition des degrés de formalisation et de satisfaction peuvent avoir un coût associé à la rédaction des contrats. Lors du développement, l'établissement des différents liens entre NFP, peut également avoir un coût. Ceci semble être amorti au fur et à mesure de l'application des évolutions sur l'architecture. En effet, cette liste de liens n'évolue pas de manière considérable dans le temps.

10.3 Traçabilité par transformation horizontale des contraintes

Les transformations de contraintes proposées dans cette thèse sont dites verticales. En effet, elles sont transformées des différents profils vers le profil standard intermédiaire. La traçabilité des décisions passe inévitablement par ces transformations. Il est possible d'investiguer la manière de transformer directement les contraintes entre profils. Il existe deux pistes pour répondre à ce besoin :

1. envisager des transformations entre le profil standard et les différents autres profils. Il faudra dans ce cas définir des projections entre ArchMM et les autres méta-modèles. Ceci a l'inconvénient d'occasionner une double transformation pour passer d'un profil à un autre. Par contre, cela offre les avantages, discutés précédemment, au niveau de l'outil ACE. En effet, on aura toujours un seul évaluateur des contraintes pour tous les profils.
2. ignorer complètement ArchMM et le profil standard. Les transformations se font directement entre profils sans passer par le profil standard. Des projections auront donc à être définies entre les différents méta-modèles. Ceci ne constitue pas un travail facile, car ArchMM a été initialement proposé dès lors qu'entre certains méta-modèles il n'existait pas de projections directes. Il faudra donc trouver un moyen pour représenter les informations perdues lors des transformations (les abstractions qui existent dans le méta-modèle source et qui n'existent pas dans le méta-modèle cible). Par contre, cette approche a l'avantage de limiter à une seule le nombre de passes de transformation à appliquer.

Les transformations horizontales de contraintes s'inscrivent directement dans un contexte de développement de logiciels à base de composants à la façon de la MDA. A une étape donnée, les contraintes définies dans toutes les étapes en amont seront disponibles dans le format (dans le profil) correspondant au langage ou à la technologie utilisés dans cette étape.

10.4 Prise en compte de l'aspect comportemental

Le travail présenté dans cette thèse explore l'aspect structurel et statique d'une architecture logicielle. Il est intéressant d'étendre cette étude à l'aspect comportemental. Il devient donc nécessaire de proposer de nouveaux méta-modèles (ou d'étendre les méta-modèles existants) avec des abstractions liées au comportement d'un système. La même approche pourra donc être suivie. On disposera d'autant de profils ACL que de méta-modèles définis. Il est possible de proposer par la suite un méta-modèle générique (ou étendre ArchMM) avec les abstractions de comportement communes. L'évaluation des nouveaux profils passera par la consultation des descriptions de comportement définies à l'aide des ADL et des diagrammes de comportement UML, ou du code des composants d'implémentation.

Au sein de l'équipe, nous avons commencé à explorer cet aspect là, mais les résultats ne peuvent pas encore être présentés, car ils ne sont pas suffisamment matures. Le travail a été effectué sur l'ADL ArchWare, qui est un ADL spécialisé dans la description de l'aspect dynamique et comportemental des architectures. Un premier méta-modèle a été développé pour cet ADL, mais l'évaluateur de contraintes spécifiées sur ce dernier n'a pas encore été implémenté. Beaucoup de travail reste à faire dans cette voie-ci. Les liens entre contraintes comportementales et propriétés non-fonctionnelles doivent être également explorés. Cette approche viendra s'ajouter à celle présentée dans cette thèse en la complétant sur l'aspect dynamique.

Chapitre 11

Épilogue

Clements et al. [25] présentent une approche pour la documentation des architectures logicielles. Les principes de cette documentation ont été discutés dans le chapitre 2. Considérons les contrats d'évolution comme la documentation d'une architecture. Ces sept principes de documentation peuvent être perçus de la manière suivante :

- "Écrire du point de vue du lecteur" : Les contrats d'évolution contiennent des informations qui s'adressent à la personne responsable de l'évolution. Ces informations concernent les décisions architecturales prises avant l'évolution et les besoins en qualité.
- "Éviter les répétitions inutiles" : Les NFT ne sont pas censées être dupliquées dans plusieurs contrats. Chaque décision doit être documentée une et une seule fois.
- "Éviter l'ambiguïté" : Le langage de description des contrats est formel. Le problème d'ambiguïté d'interprétation des contrats ne se pose pas.
- "Utiliser une organisation standard" : Les contrats ont une structure bien définie et fixée. Cette organisation est gérée de manière automatique par l'environnement d'assistance.
- "Enregistrer le raisonnement (*rationale*)" : A toutes les décisions sont associées les propriétés non-fonctionnelles correspondantes. Les raisons des choix faits sur l'architecture ont comme origine ces propriétés. Les contrats maintiennent donc bien le raisonnement ayant mené aux choix de conception.
- "Maintenir la documentation à jour, mais pas trop" : l'environnement d'assistance met à jour de manière automatique les contrats d'évolution. Cette mise à jour ne se fait que lorsqu'une évolution est validée, et que cette évolution entraîne l'ajout, la modification ou la suppression de certaines données dans le contrat (AD, NFP ou NFT).
- "Revoir la documentation" : Cet aspect là n'est pas encore considéré dans l'approche proposée dans cette thèse. Cette opération vise pour les différents intervenants dans le cycle de vie d'un logiciel à re-consulter la documentation afin de corriger certaines informations.

Comme illustré dans cette thèse, cette documentation sert bien plus qu'à la simple compréhension d'une architecture, ce qui est le cas des documentations classiques ciblées par les principes ci-dessus. Elle permet l'assistance automatique à l'évolution, et donc l'orientation de l'évolution vers un état se rapprochant le plus des spécifications initiales de qualité. Le langage de contraintes utilisé dans cette documentation a ouvert la voie à une approche pour

la traçabilité des décisions dans le cycle de développement de logiciels à base de composants. Des outils pour implémenter l'approche ont été développés. Ces outils doivent être d'avantage expérimentés sur des applications du monde réel afin de les valider. Une expérimentation et une validation industrielles de l'approche sont en cours depuis septembre 2005 afin d'évaluer le retour sur investissement de cette méthode et de définir un cadre méthodologique intégrant et affinant la méthode proposée aux processus de développement et aux Ateliers de Génie Logiciel du monde des composants. Les premiers retours de cette expérience ont clairement mis en évidence l'intérêt de l'approche pour des composants logiciels ayant eu des évolutions fréquentes occasionnées par de nombreux intervenants. Ceci constitue le travail de thèse d'une autre personne au laboratoire. C'est la raison pour laquelle je n'ai pas abordé ce sujet en détail.

Bibliographie

[1] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns : Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[2] Alcatel, Softeam, Thales, TNI-Valiosys, and Codagen Technologies Corp et al. Response to the mof 2.0 query/views/transformations rfp (ad/2002-04-10), revised submission, version 1.0. OMG Document : ad/2003-08-08, August 2003.

[3] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.

[4] Ilham Alloui and Flavio Oquendo. The archware architecture description language : Uml 1.5 profile for archware adl. Deliverable D1.4b, ArchWare European RTD Project, IST-2001-32360, June 2003.

[5] Carina Alves, Xavier Franch, Juan Pablo Carvallo, and Anthony Finkelstein. Using goals and quality models to support the matching analysis during cots selection. In *Proceedings of the 4th International Conference on COTS-Based Software Systems (ICCBSS'05)*, pages 146–156. Springer-Verlag, 2005.

[6] Colin Atkinson and Thomas Kühne. The role of metamodeling in mda. In *Proceedings of the Wokshop in Software Model Engineering (WISME@UML'2002), in conjunction with UML'02 Conference*, Dresden, Germany, October 2002.

[7] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Communications of the ACM*, 6(1) :1–17, 1963.

[8] Aline Lucia Baroni and Fernando Brito e Abreu. An ocl-based formalization of the moose metric suite. In *Proceedings of the Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'03), held in conjunction with ECOOP'03*, Darmstadt, Germany, July 2003.

[9] Aline Lucia Baroni, Coral Calero, Mario Piattini, and Fernando Brito e Abreu. A formal definition for object-relational database metrics. In *Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS'05)*, Miami, Florida, USA, May 2005.

[10] L. Bass, F. Bachmann, and M. Klein. Quality attribute design primitives and the attribute driven design method. In *Proceedings of the 4th International Conference on Product Family Engineering*, pages 169–186, Bilbao, Spain, October 2001. Springer-Verlag.

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.

[12] Keith Bennett. Software evolution : past, present and future. *Information and Software Technology*, 38(11) :671–732, 1996.

[13] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, San Francisco, California, USA, 1976. IEEE Computer Society Press.

[14] Jan Bosch and Peter Molin. Software architecture design : Evaluation and transformation. In *Proceedings of the International IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS'99)*, pages 4–10, March 1999.

[15] Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han (Daphne) Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, 31(10) :833–849, October 2005.

[16] Alan W. Brown. Model driven architecture : Principles and practice. *Software and System Modeling (SoSyM)*, 3(4) :314–327, 2004.

[17] E. Bruneton. Fractal adl tutorial, version 1.2. http ://fractal.objectweb.org/tutorials/adl/, 2004.

[18] Eric Bruneton, Coupaye Thierry, Matthieu Leclercq, Vivien Quéma, and Stefani Jean-Bernard. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering. Held in conjunction with ICSE'04*, Edinburgh, Scotland, may 2004.

[19] F. Buschmann, Meunier R., H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture : A System of Patterns*. John Wiley & Sons, 1996.

[20] Fabian Büttner. Transformation-based structure model evolution. In *Doctoral Symposium of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2005)*, Montego Bay, Jamaica, October 2005.

[21] Jordi Cabot and Ernest Teniente. Transforming ocl constraints : a context change approach. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation*, Dijon, France, April 2006. ACM Press.

[22] S. Jeromy Carrière, Steven G. Woods, and Rick Kazman. Software architectural transformation. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 13–23, Atlanta, Georgia, USA, 1999. IEEE Computer Society Press.

[23] N. Chapin, J. E. Hale, K. Md. Khan, J. F. Ramil, and W. G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution : Research and Practice*, 13 :3–30, 2001.

[24] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6) :476–493, 1994.

[25] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. Addison-Wesley, 2003.

[26] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, 2002.

[27] Antonio Coronato, Antonio d'Acierno, and Giuseppe De Pietro. Automatic implementation of constraints in component based applications. *Information and Software Technology*, 47(7) :497–509, November 2004.

[28] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 261–270. IEEE Computer Society Press, 1998.

[29] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements : From elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5) :328–350, 2004.

[30] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, California, USA, October 2003.

[31] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions On Software Engineering and Methodology*, 14(2) :199–245, 2005.

[32] Virginia C. C. de Paula and Thais V. Batista. Mapping an adl to a component-based application development environment. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, pages 128–142, Grenoble, France, April 2002. Springer-Verlag.

[33] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Extra-functional contract support in components. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE7)*, pages 217–232. Springer-Verlag, May 2004.

[34] Elisabetta Di Nitto and David Rosenblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 13–22, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[35] Massimiliano Di Penta. Evolution doctor : a framework to control software system evolution. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 280–283. IEEE Computer Society Press, 2005.

[36] Technische Universität Dresden. Ocl compiler web site. http ://dresden-ocl.sourceforge.net/, 2002.

[37] Eclipse. Eclipse modeling framework (emf). Eclipse Board Web Site : http ://www.eclipse.org/emf/, 2006.

[38] Alexander Egyed and Paul Grünbacher. Identifying requirements conflicts and cooperation : How quality attributes and automated traceability can help. *IEEE Software*, 21(6) :50–58, 2004.

[39] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay ? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1) :1–12, 2001.

[40] Len Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.

[41] Tata Consultancy Services et al. Revised submission for mof 2.0 query / views / transformations rfp, version 1.1. OMG Document : ad/2003-08-08, August 2003.

[42] Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 88–96, San José, California, USA, October 2000.

[43] Régis Fleurquin and Chouki Tibermacine. Une assistance pour l'évolution des logiciels à base de composants. In *Accepted for publication in L'Objet*. Hermes Editions, 2006.

[44] Régis Fleurquin, Chouki Tibermacine, and Salah Sadou. Le contrat d'évolution d'architectures : un outil pour le maintien de propriétés non fonctionnelles. In *Proceedings of LMO'05 Conference (Langages et Modèles à Objets)*, pages 209–222, Bern, Switzerland, March 2005. Hermes Editions.

[45] Xavier Franch. Systematic formulation of non-functional characteristics of software. In *Proceedings of the 3rd International Conference on Requirements Engineering (ICRE'98)*, pages 174–181, Colorado Springs, Colorado, USA, 1998. IEEE Computer Society Press.

[46] Xavier Franch and Pere Botella. Supporting software maintenance with non-functional information. In *Proceedings of the First IEEE Euromicro Conference on Software Maintenance and Reengineering (CSMR'97)*, pages 10–16, Berlin, Germany, March 1997. IEEE Computer Society Press.

[47] Phyllis G. Frankl, Gregg Rothermel, Kent Sayre, and Filippos I. Vokolos. An empirical comparison of two safe regression test selection techniques. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'03)*, pages 195–204, Roma, Italy, 2003.

[48] S. Frolund and J. Koistinen. Qml : A language for quality of service specification. Technical Report of Hewlett-Packard Laboratories HPL-9810, February 1998.

[49] Harald Gall, Mehdi Jazayeri, Klöch René R., and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, October 1997. IEEE Computer Society Press.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1995.

[51] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of omg mof 2.0 query / views / transformations submissions and recommendations towards the final standard. OMG Document ad/03-08-02. Object Management Group Web Site : http ://www.omg.org/docs/ad/03-08-02.pdf, 2003.

[52] David Garlan. Software architecture : a roadmap. In *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM Press.

[53] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, 1994.

[54] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44(1) :23–49, 2002.

[55] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[56] Martin Giese and Daniel Larsson. Simplifying transformations of ocl constraints. In *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2005)*, Montego Bay, Jamaica, October 2005.

[57] M. W. Godfrey and E. H. S. Lee. Secrets from the monster : Extracting mozilla's software architecture. In *Proceedings of the Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, Limerick, Ireland, June 2000.

[58] Miguel Goulao and Fernando Brito e Abreu. Bridging the gap between acme and uml 2.0 for cbd. In *Proceedings of Specification and Verification of Component-Based Systems (SAVSCB'03), workshop at ESEC/FSE 2003*, pages 75–79, Helsinki, Finland, September 2003.

[59] Miguel Goulao and Fernando Brito e Abreu. Formal definition of metrics upon the corba component model. In *Proceedings of the First International Conference on Software Architectures (QoSA'05) and the Second International Workshop on Software Quality (SOQUA'05)*, Erfurt, Germany, September 2005. LNCS 3712, Springer-Verlag.

[60] Paul Grünbacher, Alexander Egyed, and Nenad Medvidovic. Reconciling software requirements and architectures with intermediate models. *Journal of Software and Systems Modeling*, 3 :235–253, December 2003.

[61] Juha Gustafsson, Jukka Paakki, Lilli Nenonen, and A. Inkeri Verkamo. Architecture-centric software evolution by software metrics and design patterns. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 108–115. IEEE Computer Society, 2002.

[62] Didier Hoareau and Chouki Tibermacine. Component deployment evolution driven by architectural patterns and resource requirements. In *European Workshop on Software Architectures (EWSA'06)*, pages 236–243. Springer-Verlag, 2006.

[63] Lorin Hochstein and Mikael Lindvall. Combating architectural degenration : A survey. *Information and Software Technology*, 47(10) :693–707, July 2005.

[64] C Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with uml. In *Proceeding of the First Working IFIP Conference on Software Architecture (WICSA'01)*, San Antonio, Texas, USA, February 1999.

[65] IBM. Rational xde. http ://www-128.ibm.com/developerworks/rational/products/xde, 2006.

[66] IEEE. Ansi/ieee std 1471-2000, recommended practice for architectural description of software-intensive systems, 2000.

[67] ISO. Software engineering - product quality - part 1 : Quality model. International Organization for Standardization web site. ISO/IEC 9126-1. http ://www.iso.org, 2001.

[68] ISR. xarch web site. Institute for Software Research Web Site : http ://www.isr.uci.edu/architecture/xarch/, 2002.

[69] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva. Documenting component and connector views with uml 2.0. Technical report, CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, April 2004.

[70] Mehdi Jazayeri. Component programming - a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference (ESEC'05)*, pages 457–478. Springer-Verlag, 1995.

[71] Frédéric Jouault and Jean Bézivin. Km3 : a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. LNCS 4037, Springer-Verlag, 2006.

[72] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *In : Satellite Events at the MoDELS 2005 Conference*, pages 128–138, Montego Bay, Jamaica, 2006. Springer LNCS, Volume 3844.

[73] Mohamed Mancona Kandé, Valentin Crettaz, Alfred Strohmeier, and Shane Sendall. Bridging the gap between ieee 1471, an architecture description language, and uml. *Journal on Software and Systems Modeling*, 1(2) :113–129, 2002.

[74] Mohamed Mancona Kandé and Alfred Strohmeier. Towards a uml profile for software architecture descriptions. In *Proceedings of UML'2000 - The Third International Conference on the Unified Modeling Language : Advancing the Standard* -, York, United Kingdom, October 2000.

[75] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, and Steven G Woods. Experience with performing architecture tradeoff analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 54–63. ACM Press, 1999.

[76] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. Saam : A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 81–90. IEEE Computer Society Press, 1994.

[77] Rick. Kazman, Hoh Peter In, and Hong-Mei Chen. From requirements negotiation to software architecture decisions. *Information and Software Technology*, 47(8) :511–520, December 2004.

[78] Chris F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4) :493–509, 1999.

[79] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. A uml-based metamodeling language to specify design patterns. In *In Workshop In Software Model Engineering (WISME@UML'2002), held in conjunction with UML'02*, 2002.

[80] Tereza G. Kirner and Alan M. Davis. Nonfunctional requirements of real-time systems. *Advances in Computers*, 42 :1–37, 1996.

[81] Mark H. Klein, Rick Kazman, Len J. Bass, S. Jeromy Carrière, Mario Barbacci, and Howard F. Lipson. Attribute-based architecture styles. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'99)*, pages 225–244, San Antonio, Texas, USA, February 1999.

[82] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6) :42–50, 1995.

[83] Patricia Lago and Hans van Vliet. Explicit assumptions enrich architectural models. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 206–214. ACM Press, May 2005.

[84] J. Lamping. Typing the specialisation interface. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'93)*, pages 201–215. ACM Press, 1993.

[85] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *Proceedings of the third International Conference on the Unified Modeling Language (UML'00)*. Springer-Verlag, 2000.

[86] M. M. Lehman. Laws of software evolution revisited. *Lecture Notes in Computer Science*, 1149 :108–124, 1997.

[87] M. M. Lehman and Juan F. Ramil. Software evolution in the age of component-based software engineering. *IEE Proceedings - Software*, 147(6) :249–255, 2000.

[88] M.M. Lehman and L. Belady. *Program Evolution : Process of Software Change*. London : Academic Press, 1985.

[89] Fabien Leymonerie. *ASL : Un langage et des outils pour les styles architecturaux, Contribution à la description d'architectures dynamiques*. PhD thesis, Université de Savoie, France, 2004.

[90] Bennett P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communiactions of the ACM*, 24(11) :763–769, 1981.

[91] Mikael Lindvall, Roseanne Tesoriero, and Patricia Costa. Avoiding architectural degeneration : An evaluation process for software architecture. In *Proceedings of the Eighth IEEE Symposium on Software Mertrics (METRICS'02)*, pages 77–86, Ottawa, Ontario, Canada, June 2002.

[92] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, 1995.

[93] Yutao Ma, Jianxun Chen, and Jianghua Wu. Research on the phenomenon of software drift in software processes. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Lisbon, Portugal, September 2005.

[94] Nazim H. Madhavji and Josée Tassé. Policy-guided software evolution. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 75–82. IEEE Computer Society Press, 2003.

[95] J. Magee, N Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, Spain, September 1995.

[96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96), San Francisco, USA*, pages 3–14, 1996.

[97] Jeff Magee, Andrew Tseng, and Jeff Kramer. Composing distributed objects in corba. In *Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems (ISADS'97)*, page 257. IEEE Computer Society Press, 1997.

[98] Jasminka Matevska-Meyer, Wilhelm Hasselbring, and Ralf H. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *Proceeding of the Workshop on Component-Oriented Programming (WCOP'04)*, 2004.

[99] J. McCall, P. Richards, and G. Walters. Factors in software quality. Technical report, (RADC)- TR-77-369, Vols. 1–3, Rome Air Development Center, United States Air Force, Hanscom AFB, MA, 1977.

[100] J.R. McKee. Maintenance as function of design. In *Proceedings of AFIPS National Computer Conference*, pages 187–193, Reston, Virginia, USA, 1984.

[101] N Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural desing in the c2 style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'96)*, pages 24–32, San Francisco, California, USA, October 1996.

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

[103] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution*. PhD thesis, University of California, Irvine, 1999.

[104] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions On Software Engineering and Methodology*, 11(1) :2–57, 2002.

[105] Tom Mens. *A Formal Foundation of Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1999.

[106] Tom Mens and Theo D'Hondt. Automating support for software evolution in uml. *Automated Software Engineering Journal*, 7(1) :39–59, 2000.

[107] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2) :126–139, 2004.

[108] Bertrand Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 1997.

[109] Microsoft. Com : Component object model technologies. http ://www.microsoft.com/com/, 2005.

[110] Tommi Mikkonen. Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 115–124. IEEE Computer Society Press, 1998.

[111] Hafedh Mili and Ghizlaine El-Boussaidi. Representing and applying design patterns : What is the problem. In *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*. Springer-Verlag, 2005.

[112] Robin Milner. *Communicating and Mobile Systems : the π-calculus*. Cambridge University Press, 1999.

[113] Robert T. Monroe. Capturing software architecture design expertise with armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

[114] Mike Moore, Rick Kazman, Mark Klein, and Jai Asundi. Quantifying the value of architecture design decisions : Lessons from the field. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*, pages 557–563. IEEE Computer Society Press, 2003.

[115] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, April 1995.

[116] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements : A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6) :483–497, June 1992.

[117] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, held in conjunction with ICSE'00*, pages 35–46. ACM Press, 2000.

[118] OMG. Model driven architecture (mda), a technical perspective, document ormsc/01-07-01. Object Management Group Web Site : http ://www.omg.org/docs/ormsc/01-07-01.pdf, July 2001.

[119] OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. Object Management Group Web Site : http ://www.omg.org/docs/formal/02-06-65.pdf, June 2002.

[120] OMG. Meta object facility (mof) 2.0 final adopted specification, document ptc/03-10-04. Object Management Group Web Site : http ://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[121] OMG. Uml 2.0 ocl final adopted specification, document ptc/03-10-14. Object Management Group Web Site : http ://www.omg.org/docs/ptc/03-10-14.pdf, 2003.

[122] OMG. Uml 2.0 superstructure final adopted specification, document ptc/03-08-02. Object Management Group Web Site : http ://www.omg.org/docs/ptc/03-08-02.pdf, 2003.

[123] OMG. Xml metadata interchange (xmi) v2.0 specification, document formal/03-05-02. Object Management Group Web Site : http ://www.omg.org/docs/formal/03-05-02.pdf, 2003.

[124] OMG. Uml profile for corba components final adopted specification, document ptc/04-03-04. Object Management Group Web Site : http ://www.omg.org/docs/ptc/04-03-04.pdf, March 2004.

[125] OMG. Meta object facility (mof) 2.0 query/view/transformation final adopted specification, document ptc/05-11-01. Object Management Group Web Site : http ://www.omg.org/docs/ptc/05-11-01.pdf, November 2005.

[126] Flavio Oquendo. Uml profile for archware adl : Coping with uml 2.0. Deliverable D1.8, ArchWare European RTD Project, IST-2001-32360, June 2005.

[127] Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor, and David S. Rosenblum. Software architecture and component technologies : Bridging the gap. In *Proceedings of the Workshop on Compositional Software Architectures*, Monterey, California, USA, January 1998.

[128] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94*, Sorrento, Italy, May 1994. IEEE Computer Society Press and ACM Press.

[129] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.

[130] Vaclav Rajlich. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering*, 9 :235–248, 1999.

[131] J.F. Ramil and M.M. Lehman. Defining and applying metrics in the context of continuing software evolution. In *Proceedings of the 7th International Software Metrics Symposium (METRICS)*, pages 199–209, April 2001.

[132] Andreas Rausch. Software evolution in componentware using requirements/assurances contracts. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 147–156. ACM Press, 2000.

[133] Steven P. Reiss. Constraining software evolution. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM'02)*, pages 162–171, Montréal, Québec, Canada, 2002. IEEE Computer Society Press.

[134] G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, and I Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 165–174, September 2005.

[135] D.J. ROBSON, K.H. BENNETT, BJ. CORNELIUS, and M. MUNRO. Approaches to program comprhension. *Journal of Systems and Software*, 41 :79–84, 1991.

[136] Màrcia J.N. Rodrigues, Leonardo Lucena, and Thaís Batista. From acme to corba : Bridging the gap. In *Proceedings of the First European Workshop on Software Architectures (EWSA'04)*, St Andrews, UK, May 2004. Springer-Verlag.

[137] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transaction on Software Engineering*, 22(8) :529–551, 1996.

[138] RSC. Rational software corporation. uml/ejb (tm) mapping specification. Java Community Process, JSR-000026. http ://www.jcp.org/aboutJava/communityprocess/review/jsr026/, June 2001.

[139] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems : Software Technologies, Engineering Processes, and Business Practices*. SEI Series in Software Engineering. Pearson Education, 2003.

[140] S. Sendall and W. Kozaczynski. Model transformation : The heart and soul of model-driven software development. *IEEE Software*, 20(5) :42–45, 2003.

[141] M. Shaw and D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[142] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4) :314–335, 1995.

[143] J.M. Spivey. *The Z Notation : A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[144] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts : managing the evolution of reusable assets. In *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 268–285. ACM Press, 1996.

[145] Sun-Microsystems. Enterprise javabeans(tm) specification, version 2.1. http ://-java.sun.com/products/ejb, November 2003.

[146] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[147] Toufik Taibi and David Chek Ling Ngo. Formal specification of design patterns - a balanced approach. *journal of Object Technology*, 2(4), 2003.

[148] Antony Tang, Muhammad Ali Babar, Ian Gorton, and Jun Han. A survey of the use and documentation of architecture design rationale. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, Pittsburgh, Pennsylvania, USA, November 2005.

[149] Chouki Tibermacine. Un méta-modèle pour la description de contraintes architecturales sur l'évolution des composants. In *Proceedings of SE'05 workshop (Software Evolution), held in conjunction with LMO'05*, Bern, Switzerland, March 2005.

[150] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Nfrs-aware architectural evolution of component-based software. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 388–391, Long Beach, California, USA, November 2005. ACM Press.

[151] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, pages 121–130, Pittsburgh, Pennsylvania, USA, November 2005. IEEE Computer Society Press.

[152] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CB-SE'06)*, Vasteras, Sweden, June 2006. Springer LNCS.

[153] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Simplifying transformations of architectural constraints. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation*, Dijon, France, April 2006. ACM Press.

[154] Triskell. Kermeta : Triskell metamodeling kernel. Kermeta Web Site : http ://www.kermeta.org/, 2006.

[155] Jeff Tyree and Art Akerman. Architecture decisions : Demystifying architecture. *IEEE Software*, 22(2) :19–27, March/April 2005.

[156] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3) :78–85, March 2000.

[157] Daniel Varro, Gergely Varro, and Andras Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2) :205–227, 2002.

[158] W3C. Xml schema part 1 : Structures second edition. W3C Recommendation, available on :http ://www.w3.org/TR/xmlschema-1/, October 2004.

[159] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions ? *Communications of the ACM*, 20(11) :822–823, 1977.

[160] xAcme : Acme Extensions to xArch. School of Computer Science Web Site : http ://www-2.cs.cmu.edu/ acme/pub/xAcme/, 2001.

[161] Apostolos Zarras and Valérie Issarny. A framework for systematic synthesis of transactional middleware. In *Proceedings of Middleware'98*, 1998.

Annexe A

Syntaxe du langage CCL

```
constraint              := contextDeclaration
                           (stereotype name? ":" expression)+
contextDeclaration      := "context"
                           (classifierContext | operationContext)
classifierContext       := <name> ":" <typeName>
operationContext        := <typeName> "::" <name>
                           "(" formalParameterList? ")"
                               ( ":" <typeName> )?
formalParameterList     := formalParameter (";" formalParameter)*
formalParameter         := <name> ":" <typeName>
stereotype              := "archInv"
expression              := letExpression* logicalExpression
logicalExpression       := relationalExpression
                           ( logicalOperator
                             relationalExpression )*
relationalExpression    := additiveExpression
                           ( relationalOperator
                             additiveExpression )?
additiveExpression      := multiplicativeExpression
                           ( addOperator
                             multiplicativeExpression )*
multiplicativeExpression := unaryExpression
                           ( multiplyOperator unaryExpression )*
unaryExpression         := ( unaryOperator postfixExpression )
                           | postfixExpression
postfixExpression       := primaryExpression ( ("." | "->")
                           featureCall )*
primaryExpression       := literalCollection
                           | literal
                           | pathName timeExpression? qualifier?
                             featureCallParameters?
                           | "(" expression ")"
                           | ifExpression
featureCallParameters   := "(" ( declarator )?
                               ( actualParameterList )? ")"
letExpression           := "let" <name>
                           ( ":" pathTypeName )?
```

```
                        "=" expression "in"
ifExpression          := "if" expression
                         "then" expression
                         "else" expression
                         "endif"
literal               := <STRING> | <number> | "#" <name>
enumerationType       := "enum" "{" "#" <name> ( "," "#" <name>
                          )* "}"
simpleTypeSpecifier   := pathTypeName
                         | enumerationType
literalCollection     := collectionKind "{"
                         expressionListOrRange? "}"
expressionListOrRange := expression
                         ( ( "," expression )+
                           | ( ".." expression )
                         )?
featureCall           := pathName timeExpression? qualifiers?
                         featureCallParameters?
qualifiers            := "[" actualParameterList "]"
declarator            := <name> ( "," <name> )*
                         ( ":" simpleTypeSpecifier )? "|"
pathTypeName          := <typeName> ( "::" <typeName> )*
pathName              := ( <typeName> | <name> )
                         ( "::" ( <typeName> | <name> ) )*
timeExpression        := "@" <name>
actualParameterList   := expression ( "," expression )*
logicalOperator       := "and" | "or" | "xor" | "implies"
collectionKind        := "Set" | "Bag" | "Sequence" |
                         "Collection"
relationalOperator    := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator           := "+" | "-"
multiplyOperator      := "*" | "/"
unaryOperator         := "-" | "not"
typeName              := ( ["a"-"z"] | ["A"-"Z"] | "_" )
                         ( ["a"-"z"] | ["0"-"9"] |
                           ["A"-"Z"] | "_")*
name                  := ( ["a"-"z"] | ["A"-"Z"] | "_" )
                         ( ["a"-"z"] | ["0"-"9"] |
                           ["A"-"Z"] | "_")*
number                := ["0"-"9"] (["0"-"9"])*
string                := "'" ( (~["'","\\","\n","\r"])
                               | ("\\"
                                  ( ["n","t","b","r","f","\\","'","\""]
                                  | ["0"-"7"] ( ["0"-"7"] )?
                                  | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                                  )
                                 )
                         )*
                         "'"
```

Annexe B

XML Schema du contrat d'évolution

```
<!--XML Schema of an Evolution Contract-->
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Documentation -->
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      This is an XML Schema of an Evolution Contract
    </xsd:documentation>
  </xsd:annotation>
  <!-- Root element -->
  <xsd:element name="evolution-contract" type="ECType"/>
    <!-- Evolution contract -->
    <xsd:complexType name="ECType">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
          An evolution contract is composed of one or more
          Non-Functional Tactics.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="description" type="xsd:string"
                     minOccurs="0"/>
        <xsd:element name="nonfunctional-tactic" type="NFTType" minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="profil" type="xsd:string"/>
    </xsd:complexType>
    <!-- Non-Functional Tactics -->
    <xsd:complexType name="NFTType">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
          Non-Functional Tactics are composed of one Non-Functional
          Property and one Architecure Decision
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="nonfunctional-property" type="NFPType"/>
```

```
      <xsd:element name="architecture-decision" type="ADType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer"/>
  </xsd:complexType>
  <!-- Non-Functional Property -->
  <xsd:complexType name="NFPType">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        A Quality Attribute represents a quality charistic
        the architecture should gurantee. Examples include
        Maintainability and Portability
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="characteristic" type="xsd:string"/>
    <xsd:attribute name=" extern-arch-artifact" type="xsd:string"/>
  </xsd:complexType>
  <!-- Architecture Decision -->
  <xsd:complexType name="ADType">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        An Architectural Decision represents a design choice
        which guarantees an NFP. Examples include the layered
        architectural style or the pipeline style.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="formalization" type="ACType" minOccurs="1"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer"/>
  </xsd:complexType>
  <!-- Formalization is an Architecural Constraint -->
  <xsd:complexType name="ACType">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        An Architectural Constraint is a formal description
        of an AD.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer"/>
    <xsd:attribute name="profile" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

Annexe C

XML Schema du méta-modèle ArchMM

```
<!-- XML Schema of ArchMM -->
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd">
<xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
schemaLocation="http://www.ics.uci.edu/pub/arch/xarch/schemas/instance.xsd"/>
  <!-- TYPE: Identifier -->
    <xsd:element name="id" type="archinstance:Identifier"/>
  <!-- TYPE: Description -->
    <xsd:element name="description" type="archinstance:Description"/>
  <!-- TYPE: Name -->
    <xsd:element name="name" type="xsd:string"/>
  <!-- TYPE: Type -->
    <xsd:element name="type" type="xsd:string"/>
  <!-- TYPE: Value -->
    <xsd:element name="value" type="xsd:short"/>
  <!-- TYPE: ComponenentKind -->
    <xsd:simpleType name="ComponentKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Client"/>
        <xsd:enumeration value="Filter"/>
        <xsd:enumeration value="InterpretationEngine"/>
        <xsd:enumeration value="Layer"/>
        <xsd:enumeration value="Repository"/>
        <xsd:enumeration value="Server"/>
      </xsd:restriction>
    </xsd:simpleType>
  <!-- TYPE: ConnectorKind -->
    <xsd:simpleType name="ConnectorKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Adaptor"/>
        <xsd:enumeration value="Arbitrator"/>
        <xsd:enumeration value="DataAcess"/>
        <xsd:enumeration value="Distributor"/>
        <xsd:enumeration value="Event"/>
```

```
        <xsd:enumeration value="Linkage"/>
        <xsd:enumeration value="ProcedureCall"/>
        <xsd:enumeration value="Stream"/>
      </xsd:restriction>
    </xsd:simpleType>
  <!-- TYPE: PortKind -->
    <xsd:simpleType name="PortKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Input"/>
        <xsd:enumeration value="InputOutput"/>
        <xsd:enumeration value="Output"/>
      </xsd:restriction>
    </xsd:simpleType>
  <!-- TYPE: InterfaceKind -->
    <xsd:simpleType name="InterfaceKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Consumed"/>
        <xsd:enumeration value="Emitted"/>
        <xsd:enumeration value="Home"/>
        <xsd:enumeration value="Provided"/>
        <xsd:enumeration value="Published"/>
        <xsd:enumeration value="Required"/>
      </xsd:restriction>
    </xsd:simpleType>
  <!-- TYPE: RoleKind -->
    <xsd:simpleType name="RoleKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Adapted"/>
        <xsd:enumeration value="Adaptee"/>
        <xsd:enumeration value="Callee"/>
        <xsd:enumeration value="Caller"/>
        <xsd:enumeration value="Listener"/>
        <xsd:enumeration value="Sink"/>
        <xsd:enumeration value="Source"/>
        <xsd:enumeration value="Trigger"/>
      </xsd:restriction>
    </xsd:simpleType>
  <!-- TYPE: PrimitiveComponent -->
    <xsd:element name="PrimitiveComponent">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="kind" type="ComponentKind"
                       minOccurs="0"/>
          <xsd:element ref="Port" minOccurs="1"
                       maxOccurs="unbounded"/>
          <xsd:element ref="Property" minOccurs="0"
                       maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"
                       use="required"/>
      </xsd:complexType>
    </xsd:element>
  <!-- TYPE: CompositeComponent -->
    <xsd:element name="CompositeComponent">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="kind" type="ComponentKind"
                     minOccurs="0"/>
        <xsd:element ref="Port" minOccurs="1"
                     maxOccurs="unbounded"/>
        <xsd:element ref="Property" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="CompositeComponent" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="PrimitiveComponent" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="PrimitiveConnector" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="CompositeConnector" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="Configuration" minOccurs="0"
                     maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"
                     use="required"/>
    </xsd:complexType>
  </xsd:element>
<!-- TYPE: Port -->
  <xsd:element name="Port">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="name"/>
        <xsd:element name="kind" type="PortKind"/>
        <xsd:element ref="Interface" minOccurs="1"
                     maxOccurs="unbounded"/>
        <xsd:element ref="Binding" minOccurs="0"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string"
                     use="required"/>
    </xsd:complexType>
  </xsd:element>
<!-- TYPE: Interface -->
  <xsd:element name="Interface">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="name"/>
        <xsd:element name="kind" type="InterfaceKind"/>
        <xsd:element ref="Port" minOccurs="0"
                     maxOccurs="1"/>
        <xsd:element ref="Property" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="Service" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element ref="Binding" minOccurs="0"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string"
```

```
                    use="required"/>
    </xsd:complexType>
  </xsd:element>
<!-- TYPE: Role -->
  <xsd:element name="Role">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Binding" minOccurs="0"
                    maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="archinstance:Identifier"
                    use="optional"/>
      <xsd:attribute name="name" type="xsd:string"
                    use="required"/>
      <xsd:attribute name="kind" type="RoleKind"
                    use="required"/>
    </xsd:complexType>
  </xsd:element>
<!-- TYPE: Service -->
  <xsd:element name="Service">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="name"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="archinstance:Identifier"
                    use="required"/>
    </xsd:complexType>
  </xsd:element>
<!-- TYPE: Connector -->
  <xsd:element name="PrimitiveConnector">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="kind" type="ConnectorKind"
                    minOccurs="0"/>
        <xsd:element ref="Role" minOccurs="2"
                    maxOccurs="unbounded"/>
        <xsd:element ref="Property" minOccurs="0"
                    maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"
                    use="required"/>
    </xsd:complexType>
  </xsd:element>

<!-- TYPE: CompositeConnector -->
  <xsd:element name="CompositeConnector">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="kind" type="ConnectorKind"/>
        <xsd:element ref="Role" minOccurs="2"
                    maxOccurs="unbounded"/>
        <xsd:element ref="Property" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element ref="Configuration" minOccurs="1"
```

```
                    maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"
                    use="required"/>
      </xsd:complexType>
    </xsd:element>
  <!-- TYPE: Property -->
    <xsd:element name="Property">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="name"/>
          <xsd:element ref="type"/>
          <xsd:element ref="value"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="archinstance:Identifier"
                    use="required"/>
      </xsd:complexType>
    </xsd:element>
  <!-- TYPE: Binding -->
    <xsd:element name="Binding">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="side" minOccurs="2"
                    maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:attribute name="name" type="xsd:string"
                          use="required"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"
                    use="required"/>
      </xsd:complexType>
    </xsd:element>
  <!-- TYPE: Configuration -->
    <xsd:element name="Configuration">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="Binding" minOccurs="0"
                    maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"
                    use="required"/>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Annexe D

XML Schema des documents de sérialization des contraintes

```
<!--XML Schema of the Constraint Serialization XML Document-->
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <!-- This is the root node-->
  <xs:element name="acl-constraint">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="context"/>
        <xs:element ref="constraint" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- This describes the context of the constraint.-->
  <xs:element name="context">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="identifier"/>
        <xs:element ref="type"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- This is the name of the the constraint's context-->
  <xs:element name="identifier">
    <xs:complexType>
      <xs:attribute name="value" use="required"
                    type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <!-- This describes the type of one element
       in the constraint-->
  <xs:element name="type">
    <xs:complexType>
      <xs:attribute name="value" use="required"
```

```
                    type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <!-- One constraint is composed of one or more nodes -->
  <xs:element name="constraint">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="node"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- A node describes an atomic element. -->
  <xs:element name="node">
    <xs:complexType>
      <xs:attribute name="type" use="required"/>
      <xs:attribute name="value" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```