Une assistance pour l'évolution des logiciels à base de composants

Régis Fleurquin — Chouki Tibermacine

Laboratoire VALORIA
Centre de Recherche Yves Coppens
Université de Bretagne Sud
F-56000 Vannes cedex
{Regis.Fleurquin, Chouki.Tibermacine}@univ-ubs.fr

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. Nous présentons une approche qui aide à diminuer les coûts associés à ces deux étapes pour des applications conçues à l'aide de composants. Elle propose de documenter formellement, à chaque étape du cycle de vie, les liens unissant les attributs qualité d'une application et les choix architecturaux qui leur sont dédiés. Lors d'une évolution, un outil d'assistance 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 qu'il réalise.*

ABSTRACT. *Among all the stages which constitute the maintenance process, the comprehension of an application before its evolution, and regression testing after its evolution are by far the most expensive. We present an approach which helps to reduce the costs associated with these two stages for component-based software. It proposes to formally document at each stage of the software life cycle the links between quality attributes of an application and the architectural decisions implementing them. During an evolution, a tool uses this documentation, for guaranteeing on the one hand its update, and for notifying the developer of the possible effects of architectural changes on quality attributess he made on the other hand.*

MOTS-CLÉS : *maintenance, qualité, ADL, composants, contrat d'évolution, contraintes architecturales.*

KEYWORDS: *maintenance, quality, ADL, components, evolution contract, architectural constraints.*

1. Introduction

La première loi de Lehman, issue de constatations sur le terrain, stipule qu'un logiciel doit évoluer faute de quoi il devient progressivement inutile (Lehman *et al.*, 1985). Bien qu'ancienne cette loi ne s'est jamais démentie. La maintenance coûte même de plus en plus chère. Estimée dans les années 1980 et 1990 à environ 50 à 60 % (Lientz *et al.*, 1981; McKee, 1984) des coûts associés aux logiciels, de récentes études évaluent désormais ce coût entre 80 et 90 % (Erlikh, 2000; Seacord *et al.*, 2003). La réactivité, toujours plus grande, exigée des applications informatiques toujours 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 ressource (section 2). L'étape de compréhension de l'architecture, à elle seule, compte par exemple pour plus de 50 % du temps de maintenance (Bennett, 1996) ; la faute à des documentations incomplètes, ambiguës et rarement mises à jour, que même les outils actuels de rétro-ingénierie encore limités à la génération de vues trop proches du code ne parviennent pas à faire revivre. 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 vient tardivement et génère d'inévitables et coûteux allers et retours entre la phase de test et celle de développement.

Après quelques définitions et rappels sur l'évolution des logiciels (section 2), nous présentons une approche qui aide à diminuer les coûts associés à ces deux étapes pour des applications conçues à l'aide de composants (section 3). Cette approche propose de documenter formellement, à chaque étape du cycle de vie, les liens unissant les exigences qualité au sens de la norme ISO 9126 (ISO, 2001) (fiabilité, maintenabilité, etc.) aux choix architecturaux qui visent à leur obtention. Nous montrons que cette documentation si elle est exploitée automatiquement selon des règles strictes, non seulement facilite l'étape de compréhension en garantissant la présence d'informations architecturales essentielles et à jour sous une forme non ambiguë, mais également qu'elle aide à prévenir au plus tôt la régression des propriétés qualité. L'élément-clé de cette approche est le langage utilisé pour établir cette documentation : le langage ACL (section 4). Ce langage a été conçu pour s'insérer naturellement dans un cadre de développement multilangage. Il se décline donc en autant de « profils » que le processus de développement compte de langages de développement. Une plate-forme logicielle supportant cette approche a également été développée (section 5).

2. Un état de l'art sur l'évolution des logiciels

Nous rappelons dans cette section quelques définitions. Nous insistons en particulier sur les liens qu'entretiennent les concepts de maintenance et d'évolution. Ces deux termes sont, en effet, souvent employés concurremment dans la littérature sans qu'il soit possible de positionner clairement l'un par rapport à l'autre. Nous présentons ensuite les phases et étapes, très souvent méconnues, qui rythment la vie d'un logiciel après sa première mise en service. Dans la mesure où notre approche se focalise sur les étapes de compréhension et de vérification d'une action d'évolution, nous nous focalisons, ensuite, sur les enjeux et problèmes posés par ces deux étapes.

2.1. *Maintenance* versus *évolution*

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 semble dégager un certain consensus, il n'en est rien du second. Nous indiquons donc, ce que désigne successivement ces deux termes en insistant tout particulièrement sur les divergences qu'ils suscitent.

2.1.1. *La 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 restreindre à 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 œuvre 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, devrait être 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 en conséquence 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.

2.1.2. *L'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 beaucoup plus positive. Certains laboratoires et revues internationales 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 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 1960. 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, revisitées quelques 20 années plus tard (Lehman, 1997). 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 (Robles *et al.*, 2005), le terme évolution désignait l'étude de la dynamique, au fil du temps, des propriétés (Ramil *et al.*, 2001) 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. Clairement cette école positionnait, à l'époque, l'évolution comme une discipline nouvelle (Figure 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 fonctions discrètes obtenues pour chaque variable mesurée. Le lecteur intéressé trouvera une synthèse des travaux entrepris dans le domaine dans (Kemerer *et al.*, 1999).

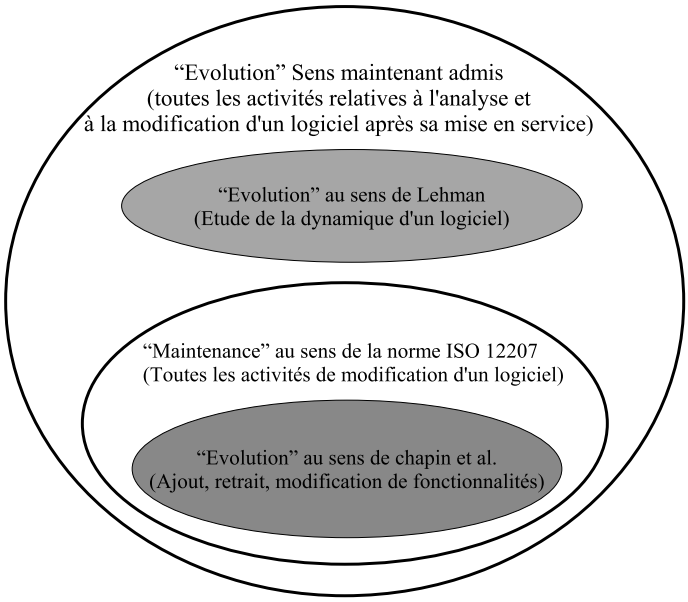

Figure 1. *Les écoles de défi nition du terme évolution*

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 repésentants de cette école (Chapin *et al.*, 2001). 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 dans le temps des propriétés d'un logiciel, des processus et des organisations. C'est la définition que nous adoptons dans la suite de cet article.

Le terme évolution étant maintenant précisé, nous allons rappeler quelles sont les différentes phases traversées par un logiciel après sa mise en service et les activités menées lors de ces phases.

2.2. *Le processus d'évolution*

Dans la littérature, on distingue 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 vastes. De grandes périodes de la vie d'un logiciel appelées phases, durant lesquelles le logiciel semble évoluer d'une manière bien particulière, émergent alors. Nous allons présenter dans cette section ces deux visions. Nous présenterons en premier lieu la vision macroscopique puis la vision microscopique.

2.2.1. *Le macroprocessus d'évolution*

La vision macroscopique est assez peu connue. Dans chacune des phases de la vision macroscopique, les actions d'évolution 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 œuvre pour chacune d'entre elles se ressemblent.

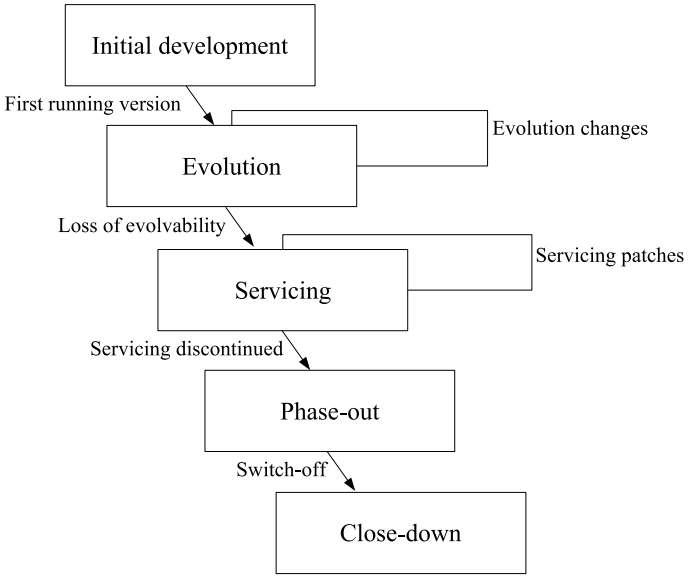

Figure 2. *Le macroprocessus d'évolution de Rajlich*

Le modèle de Rajlich (Rajlich, 1999) est le modèle le plus connu relevant de cette vision. Il affirme qu'un logiciel passe dans sa vie par 5 phases (Figure 2) : développement initial, évolution, service, fin de vie et 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.

La phase d'évolution est celle pendant laquelle le logiciel va subir l'essentiel de ses mises à jour. Son architecture, bien que se dégradant rapidement, autorise encore des modifications d'autant que l'expertise nécessaire à ce genre d'action est encore présente au sein des équipes de développement. C'est dans cette phase que notre approche de maîtrise de l'évolution prend place. Son objectif est bien sûr de prolonger au maximum cette phase ; c'est-à-dire de repousser le plus loin possible le moment où les coûts des modifications deviennent tels que celles-ci peuvent ne plus être conduites.

Lors de la phase d'évolution, les mises à jour vont se succéder en respectant généralement le même processus. C'est ce processus, qualifié de microscopique, que nous allons maintenant décrire.

2.2.2. *Le microprocessus d'évolution*

La norme ISO 12207 donne une description consensuelle sur le plan microscopique de l'ensemble des processus et activités à suivre lors d'un acte de mise à jour d'un logiciel. Des activités citées par ce texte, nous n'évoquerons pas ici les activités qualifiées de « support » par cette norme (gestion de configuration, assurance qualité, etc.), pour nous limiter à celles directement liées au processus de modification.

La réception d'une demande de changement est le point de départ de toute action d'évolution (Figure 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 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 (Chapin *et al.*, 2001). Celle que nous avons 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'archi-

tecture 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 des modèles, du code et des documentations associées. On use ici d'outils, méthodes et procédures identiques à ceux utilisés lors du développement de la version initiale de l'application. 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.

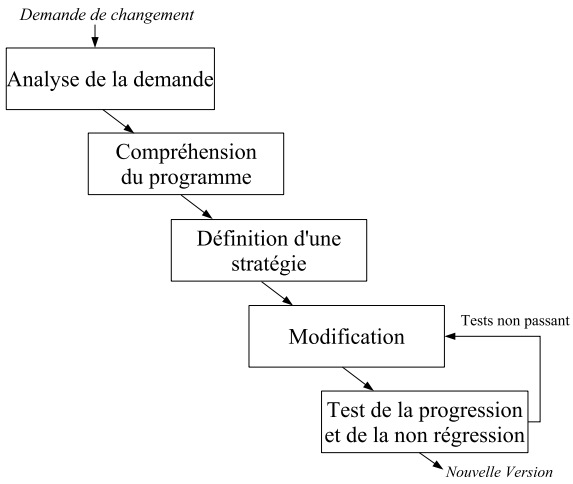

Figure 3. *Le microprocessus d'évolution*

Nous allons maintenant nous intéresser aux deux étapes qui nous concernent et qui sont les plus coûteuses : la compréhension et la vérification.

2.3. *Les étapes de compréhension et de vérification*

Nous présentons dans cet article une approche qui aide à diminuer les coûts associés aux étapes de compréhension et de vérification pour des applications conçues à l'aide de composants. Nous allons donc présenter en détail ces deux étapes : leurs enjeux et les moyens actuellement disponibles pour les mener à bien.

2.3.1. *L'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 sur sa structure un niveau

de connaissance suffisant. Il s'agit, non seulement de reconstruire une image de ses réelles aptitudes fonctionnelles et non fonctionnelles (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 facile 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. Typiquement pour générer, depuis un exécutable du code, ou depuis du code des modèles de conception (par exemple des diagrammes de classes ou de séquence UML). Les techniques de compréhension de programmes 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 (Robson *et al.*, 1991). Ces deux types de technique sont passives, elles n'altèrent en rien un système. Elles sont par contre partie prenante de cycles de réingénierie 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 parlants donc le plus souvent propriétaires, une information à très haute valeur ajoutée ; une information qui n'avait jamais été 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 (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 « pourquoi » (l'objectif recherché au travers d'un choix architectural) au « comment » (le choix architectural constaté) semble encore un vœu 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.

2.3.2. *L'étape de vérifi cation*

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 (Frankl *et al.*, 2003). 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 d'un cycle de modification qui sera suivi 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 homme. 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 notre connaissance, que de rares travaux dans le monde des composants. On citera en particulier ceux d'Occello *et al.* (Occello *et al.*, 2004).

Ces définitions données et ces quelques rappels méthodologiques effectués nous allons maintenant présenter une approche concourant à la maîtrise de l'évolution d'un logiciel lors d'un microprocessus d'évolution.

3. Une assistance à l'évolution architecturale dirigée par les exigences qualité

Nous présentons dans cette section, 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 également offre un cadre de vérification *a priori* de la non-régression selon la composante non fonctionnelle. Nous commencerons, dans un premier temps, par présenter nos hypothèses de travail. Ces hypothèses nous ont conduit à introduire le concept de « contrat d'évolution » et un algorithme d'assistance à l'évolution que nous décrirons ensuite. La section se conclura par un exemple d'évolution illustrant le fonctionnement de cet algorithme d'assistance.

3.1. *Importance des liens unissant exigences qualité et architecture*

3.1.1. *Exigences qualité et architecture des logiciels*

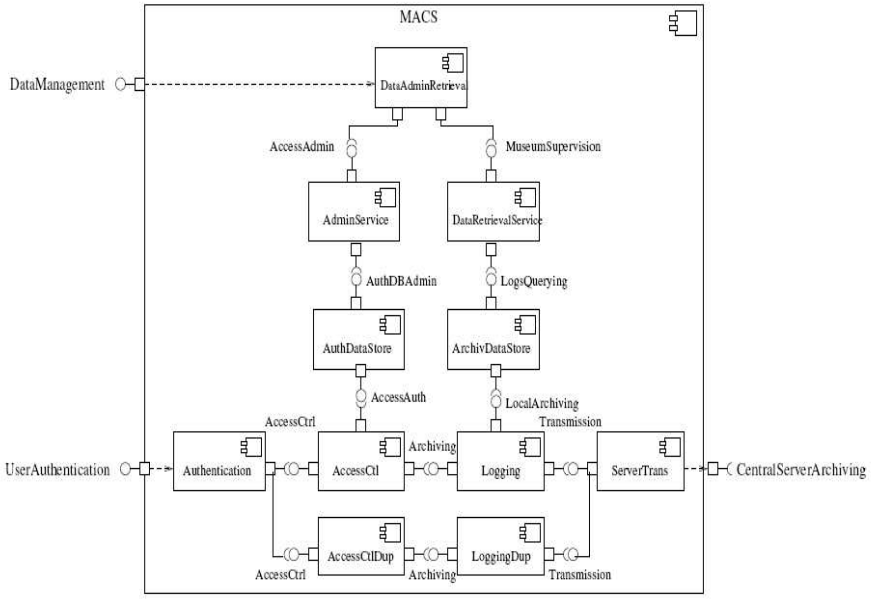

Figure 4. *Architecture d'un composant de contrôle d'accès à un musée*

Il est admis que ce ne sont pas les fonctions attendues d'un logiciel qui déterminent son architecture, mais bien les attributs qualité requis (Bass *et al.*, 2003). La figure 4 illustre ce propos. Elle présente l'architecture d'un composant gérant le contrôle d'accès à un bâtiment. Son architecture 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 certaines exigences qualité.

Ainsi, le composant `DataAdminRetrieval` 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 l'exigence qualité qui avait été formulée dans sa spécification initiale : « Le composant devra pouvoir facilement prendre en charge de nouveaux formats de données ». De manière identique, la séquence de sous-composants `AccessCtlDup` et `LoggingDup` est une duplication architecturale de la fonction d'authentification. En effet, si la séquence normale `AccessCtl` et `Logging` n'est pas utilisable du fait, par exemple, d'une absence de données en entrée *via* le port `DataManagement`, il sera toujours possible pour une liste restreinte de personnes (dont les autorisations sont stockées localement dans le sous-composant `AccessCtlDup`) d'accéder au bâtiment. Cette duplication architecturale fut 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.

3.1.2. *Evolution et liens architecture-exigences*

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 `DataAdminRetrieval` lors d'une évolution peut potentiellement manifester une perte du niveau de maintenabilité attendu sur les formats de données.

Notifier automatiquement, au moment même de la suppression, cette perte potentielle peut aider l'architecte à déterminer si il doit ou non maintenir sa décision.

Nous proposons donc, d'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 à la demande le respect des choix architecturaux documentés, 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 nouvelles exigences, 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.

3.2. *Une implantation sous la forme de contrats d'évolution*

Dans l'approche que nous proposons, 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 nous appelons le *contrat d'évolution* d'un logiciel. Dans un premier temps, nous détaillons la structure de ce contrat. Nous indiquons ensuite de quelle manière ce contrat va être exploité pour assister le processus d'évolution.

3.2.1. *Le contrat d'évolution*

Nous parlons de contrat car il documente 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. Nous associons au contrat d'évolution le vocabulaire ci-dessous :

NFP (propriété non fonctionnelle) : une 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) : une partie de l'architecture du logiciel qui cible une ou plusieurs NFP (par exemple le respect d'une 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 soucis de factorisation, sur la base d'autres AD ;

NFT (tactique non fonctionnelle) : un 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 syle en *pipeline* avec une NFP relative à la performance d'un service particulier offert par le logiciel) ;

NFS (stratégie non fonctionnelle) : l'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.

3.2.2. *L'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 dans 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). Nous avons 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 (Tibermacine *et al.*, 2005) :

– **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, de maintenir ou non les 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 redocumenter 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.

3.3. *Un exemple d'évolution avec assistance*

Nous allons maintenant montrer sur un exemple comment ces règles permettent d'assister le développeur lors de son activité d'évolution. Nous commençons par décrire le contrat d'évolution porté par le composant exemple avant son entrée dans un cycle d'évolution. Nous détaillons ensuite comment ce contrat d'évolution est exploité par l'algorithme précédent pour assister le développeur à chaque étape du cycle d'évolution.

3.3.1. *Le contrat d'évolution avant évolution*

L'assistance associée aux règles précédentes est illustrée par le scénario d'évolution de la figure 5. Nous disposons avant évolution dans ce système 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.

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, nous illustrons 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.

3.3.2. *Une évolution avec assistance*

Supposons que la personne réalisant l'évolution du logiciel, auquel est associée la NFS ci-avant, 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).

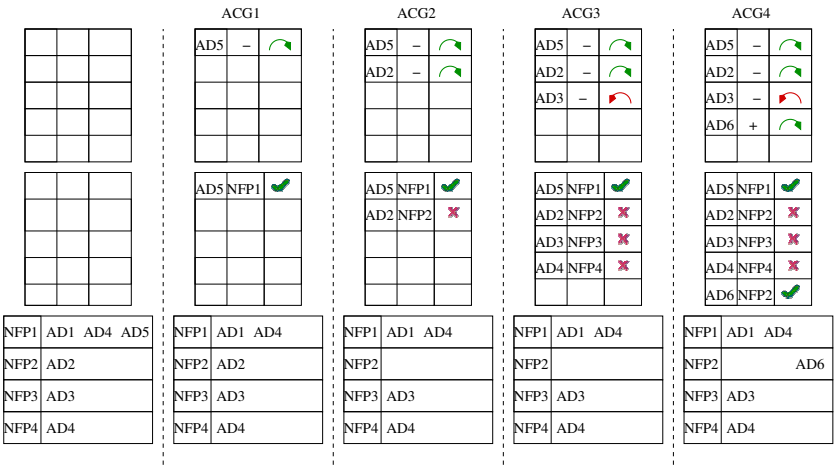

Figure 5. *Assistance à l'activité de l'évolution architecturale avec le contrat d'évolution (NFS)*

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 notifier que AD3, et par conséquent NFP3, vont être affectées. En plus, cette AD a une autre AD qui l'inclut dans sa description (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.

Nous avons montré à travers ce scénario d'évolution comment l'approche proposée assiste le développeur durant ses opérations d'évolution. Cette assistance aide à obtenir une architecture conforme aux exigences qualité souhaitées. 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 notre 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. Nous allons consacrer toute la section qui suit à présenter un langage baptisé ACL qui a été défini pour ce faire.

4. Le 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, nous commençerons par présenter le langage ACL. Ce langage présente une structure très particulière qu'il est important de bien comprendre. Nous commençerons donc par évoquer cette structure avant de présenter dans une deuxième sous-section la syntaxe et la sémantique du langage ACL. Dans une troisième sous-section, nous décrirons, pour conclure, la structure des documents XML utilisés pour stocker les NFT.

4.1. *La structure du langage ACL*

Nous allons, dans un premier temps, lister les différents types de décisions architecturales (AD) que nous souhaitons prendre en compte. Nous mettrons en particulier en évidence la grande diversité des AD que l'on doit pouvoir exprimer. Nous présenterons, ensuite, la structure langagière à deux niveaux que nous avons conçue ; une structure originale usant d'un langage de contrainte nommé CCL et de plusieurs métamodèles regroupés au sein de profils ; cette structure originale facilite la prise en compte de cette diversité. Nous décrirons ensuite, comment ces deux niveaux s'articulent pour permettre la description d'AD.

4.1.1. *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, qu'on ajoute 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 (van Ommering *et al.*, 2000).

La seconde difficulté est liée au fait que ce langage doit être utilisable à chaque étape du cycle de vie d'une application. Nous devons 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, telle que EJB (Sun-Microsystems, 2003), COM+/.net (Microsoft, 2005) ou CCM (OMG, 2002). Pour répondre à cette diversité, nous avons conçu un langage, nommé ACL, (*Architectural Constraint Language*).

4.1.2. *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 légèrement modifiée d'OCL, baptisée CCL (*Core Constraint Language*). Le second niveau prend la forme d'un ensemble de métamodèles au format MOF. Ces métamodèles représentent les abstractions structurelles à contraindre, rencontrées dans les principaux langages de modélisation utilisés à chaque étape du cycle de vie. Ces abstractions 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 métamodèle particulier représente ce que nous appellerons 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 (xAcme : Acme Extensions to xArch, 2001) 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 de notre profil dédié à CCM. Le profil xAcme est composé de CCL et du métamodèle xArch établi pour xAcme (voir plus loin). Le profil CCM est lui composé de CCL et de notre métamodèle 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 entrelacé 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 métamodèle concerné. Ces métamodèles 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.

4.1.3. *Articulation d'un couple (CCL, métamodèle) 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/postconditions 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 ArticleL_Objet inv:
self.taille >= 10
context a:ArticleL_Objet inv:
a.ecritPar->size() >= 1
```

Dans les deux cas, le contexte est une classe (`ArticleL_Objet`). Les deux contraintes sont écrites selon le point de vue d'une instance quelconque du contexte (ici un objet instance de la classe `ArticleL_Objet`). Mais la contrainte exprimée doit être vérifiée par toutes les instances du contexte. C'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 pour la désigner un identificateur *ad hoc* `a`. 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 métamodèle. Un métamodèle 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 métaclasse 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 phrase sont écartées. Si ce métamodèle 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 bridé sciemment dans son pouvoir d'expression car on ne tolère pas la description de certains types d'architecture. 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 métaclasse `composant`. Cette contrainte est exactement du type de celle que nous souhaitons pouvoir exprimer. Malheureusement sa portée est globale. Elle s'applique à tout composant et non à un composant particulier comme nous souhaitons 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 nous allons donner maintenant la description.

4.2. *Syntaxe et sémantique du langage ACL*

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

4.2.1. *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 présentait, en effet, plusieurs avantages. Il fait l'objet d'un standard dans le sillage d'UML, donc connu de tous. La plupart des formations en informatique conduisant au niveau cadre incluent d'ailleurs son apprentissage. Il était en effet important de ne pas ajouter un n^{ieme} 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 (Briand *et al.*, 2005). Enfin, il existe des compilateurs de ce langage dont les sources sont libres de droit.

CCL est (presque) sur le plan syntaxique et sémantique, un sous-ensemble du langage OCL. Les seules règles de production du langage OCL qui ne sont pas supportées par CCL sont celles relatives à l'expression des pré et postconditions. De plus,

pour limiter la portée d'une contrainte à un composant particulier, nous proposons de modifier légèrement la syntaxe et la sémantique de la partie contexte d'OCL (CCL). Sur le plan syntaxique, nous imposons que tout contexte introduise nécessairement un identificateur. Cet identificateur doit obligatoirement être le nom d'une ou plusieurs instances particulières de la métaclasse citée dans le contexte. Sur le plan sémantique, nous interprétons la contrainte avec le sens qu'elle aurait dans le contexte d'une métaclasse mais en limitant sa portée aux instances citées dans le contexte.

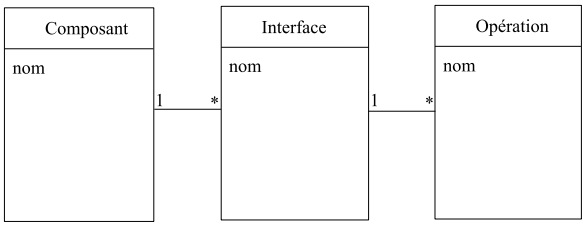

Figure 6. *Un métamodèle jouet*

Pour mieux comprendre la syntaxe et la sémantique de CCL, voici un petit exemple. On supposera que le métamodèle jouet (Figure 6) parcouru par cette contrainte est composé de 3 métaclasses (`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 à notre métamodèle. 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 ou nous réduisons la portée de cette contrainte à seulement deux instances de la métaclasses 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'arterfact architectural auquel il s'applique : le marqueur `@old`. Considérons l'exemple suivant s'appliquant à notre métamodèle jouet.

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

Cette contrainte précise que le composant `C` ne peut pas augmenter de plus d'une unité le nombre de ses interfaces d'une version à sa suivante. Notons qu'elle ne dit rien sur les modifications éventuellement subies par les anciennes interfaces de ce composant. Si nous voulions imposer, qu'au plus une seule de ces anciennes interfaces ne peut être altérée d'une version à l'autre, nous devrions comparer les structures avant et après évolution de toutes ces interfaces. Ce type de contrainte serait pénible à écrire avec les opérateurs OCL usuels et le seul marqueur `@old` . Nous avons donc introduit les opérateurs de collection qui suivent : `modified() :Collection(T)` qui retourne tous les éléments de la collection à laquelle il s'applique qui ont subi une modification entre l'actuelle et la précédente version (seulement pour ceux existant dans les deux versions), `added() :Collection(T)` qui fait de même pour les éléments qui n'existaient pas dans la précédente version et `deleted() :Collection(T)` qui 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()->size())=0) and
(self.interface->modified()->size())<2)
```

Maintenant que nous avons présenté le langage CCL, nous allons présenter les profils que nous avons définis. Un profil est composé du langage CCL et d'un métamodèle sur lequel seront écrites les contraintes. Un métamodèle 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 nous souhaitions supporter un nombre suffisant de langage pour couvrir l'intégralité du cycle de vie, nous avons à ce jour défini 5 profils (donc 5 métamodèles) : xArch, UML, CCM, EJB et Fractal (Bruneton *et al.*, 2004). Nous ne présenterons dans la suite que deux d'entre eux : xArch 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.

4.2.2. *Un premier exemple de profil : xArch 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 (Medvidovic *et al.*, 2000). 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 (Luckham *et al.*, 1995), Darwin (Magee *et al.*, 1996) ou Koala (van Ommering *et al.*, 2000) par exemple, n'explicitent pas la notion de connecteur. D'autres permettent la description 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 (Shaw *et al.*, 1995), Wright (Allen, 1997) ou encore Acme (Garlan *et al.*, 2000), nous pouvons définir des connecteurs composites, alors que cela est 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, pour sa part, a pour objectif la description architecturale de systèmes événementiels. SADL (Moriconi *et al.*, 1995) se focalise sur le raffinement des architectures. C2SADEL modélise des architectures dans le style C2 (Medvidovic, 1999), etc. La classe des langages ADL est donc assez inhomogène.

Nous avions alors deux solutions pour couvrir l'ensemble des ADL : proposer, soit un profil propre à chaque ADL, soit un unique profil pour tous les ADL. La première solution a le mérite d'offrir un métamodèle dédié à chaque ADL de la littérature. Ces métamodèles 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 cette ADL cible. Le problème est qu'il nous aurait fallu plus d'une dizaine de profils pour couvrir les ADL les plus connus à ce jour. De plus les ADL que nous ne conssaissions pas se retrouveraient sans profil et pour chaque nouvel ADL apparaissant il nous aurait fallu créer un nouveau profil. La seconde solution consiste à proposer un seul profil (donc un seul métamodèle), plus générique, pour tous les ADL. Ce métamodèle 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 nous avons retenue. Nous avons donc créé un unique profil pour tous les ADL, dont le métamodèle décrit les concepts structuraux du langage xArch (ISR, 2002). 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. Nous avons donc décidé de l'utiliser comme référence pour élaborer notre métamodèle.

La figure 7 représente le métamodèle MOF du profil xArch. 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`).

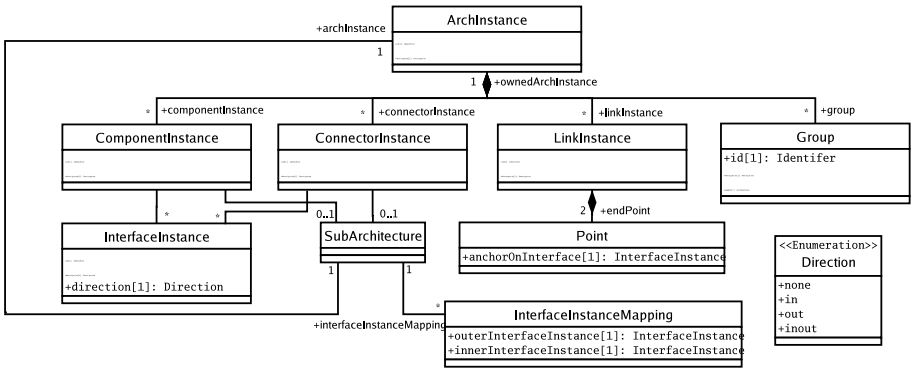

Figure 7. *Le métamodèle du profil xArch*

4.2.3. *Un second exemple de profil pour 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 pour l'exécution de ces applications, en termes de services transactionnels, de sécurité, de répartition, etc. Les modèles abstraits définis 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.

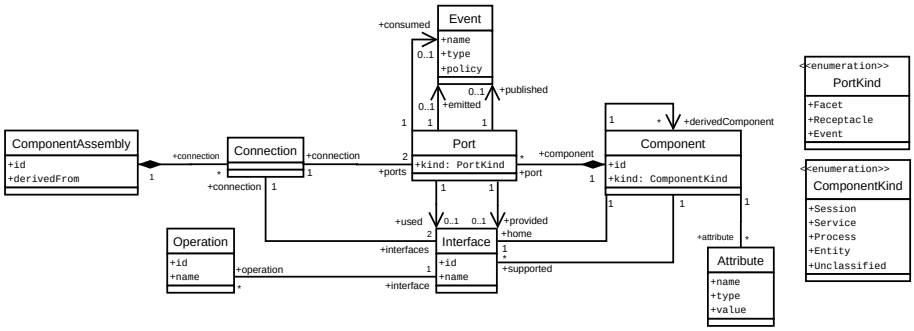

Figure 8. *Le métamodèle du profil CORBA*

La figure 8 représente le métamodèle 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. Il peut être de différents types : session, entité, service, process ou d'un autre type. Par contre, CCM est un modèle de composants plat. Un composant ne peut être conçu sur la base d'autres composants.

4.3. *Description des NFT et des NFS*

Nous venons de présenter le langage ACL qui permet de documenter des décisions architecturales. Il nous reste à proposer un moyen pour documenter les liens unissant ces décisions (que nous appelons des AD) aux exigences qualité (que nous appelons des NFP). Dans notre approche un tel lien est appellé tactique non fonctionnelle (NFT). L'ensemble des NFT d'un composant constitue une sratégie non fonctionnelle (NFS). Nous allons dans un premier temps détailler la structure que nous avons adoptée pour documenter les NFT. Nous avons fait le choix de stocker ces NFT dans des documents XML. Nous donnerons dans un deuxième temps, un exemple de ce type de document.

4.3.1. *La structure des NFT*

Les définitions introduites dans la section 3 sont illustrées par la figure 9. 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 tout ou partie de l'exigence 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 exigences 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 une des exigences qualité formulées dans le document de spécification d'un composant logiciel. Une NFP doit être une exigence atomique. Une NFP est dite atomique si elle n'est associée qu'à un et un seul attribut qualité dont la portée est unique. Notre modèle accepte actuellement comme attribut qualité, les caractéristiques et les sous-caractéristiques du modèle de qualité ISO/IEC 9126 (ISO, 2001) (par exemple, la maintenabilité, la portabilité, etc.). Un attribut qualité est porté par un artefact architectural externe. Un artefact externe représente un concept architectural publique d'un composant ; c'est-à-dire un artefact visible par les autres composants. Dans notre 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 cette exigence. Voici deux exemples de NFP qui pourraient 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.

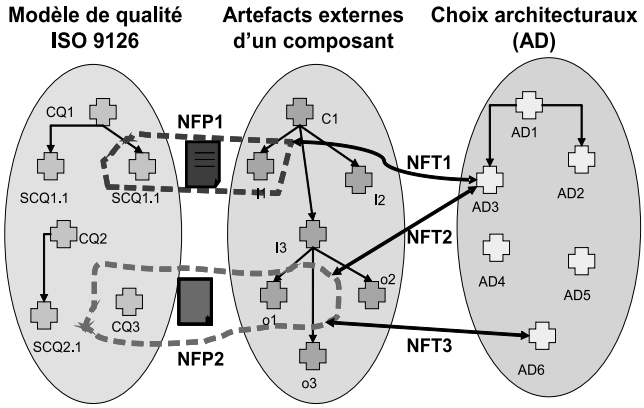

Figure 9. *Expression des NFS*

En l'état nous ne faisons aucune hypothèse sur le format d'écriture des exigences dans la documentation du composant. Elles peuvent être formulées 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 (Frolund *et al.*, 1998) 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. Nous sommes ici tributaires des avancés 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 possible, la seule contrainte que nous imposons, pour des raisons de stockage et d'affichage, est que le format d'expression des exigences 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.

4.3.2. *Le stockage des NFS*

Très concrètement, une NFS qui détaille l'ensemble des NFT prend la forme dans notre approche d'un document XML respectueux d'un schéma XML reprenant la

structure que nous venons de décrire. Le listing ci-dessous représente l'exemple d'une NFS.

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

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 qualié Portabilité, ii) l'artefact architectural externe auquel est associé cet attribut et qui représente le composant nommé MACS et iii) la description textuelle de l'exigence 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).

5. Une plate-forme logicielle pour les contrats d'évolution

Une plate-forme a été développée pour offrir les outils nécessaires au suivi d'un microprocessus d'évolution respectueux de la démarche évoquée dans la section 3. Nous présenterons d'abord les fonctionnalités offertes par ces outils. Nous insisterons ensuite sur le défi posé par l'évaluation de contrats écrits dans un langage polymorphe car offrant différents profils. Cette difficulté a pu être surmontée en introduisant un langage intermédiaire servant de base à ces évaluations. Nous décrirons le contenu et des exemples d'utilisation de ce langage.

5.1. *Les outils*

Nous avons développé des outils qui permettent l'édition, la compilation et l'évaluation des contrats d'évolution. Ces outils sont les briques de base d'un outil d'assistance en cours de réalisation qui assistera le développeur durant une opération d'évolution selon l'algorithme décrit à la section 3.

– *Editeur de NFS :* cet outil permet une édition assistée de contrats. Il impose de choisir le profil d'écriture souhaité. Le choix d'un profil particulier se résume pour l'outil à charger le fichier au format XMI du métamodèle associé au profil sélectionné. Dans sa version actuelle, l'éditeur est à même, en parcourant le métamodèle, d'offrir des mécanismes de complétion automatiques des instructions de navigation saisies ;

– *Compilateur de NFS :* cet outil vérifie la correction syntaxique d'un contrat dans le cadre d'un profil particulier. Cette vérification se fait en trois passes : la première vérifie que le code est un code CCL valide, la seconde que les parcours font bien référence à des parcours valides dans le métamodèle du profil, la troisième que les entités citées restantes sont bien des artefacts de la description architecturale du composant sur lequel porte le contrat. La description architecturale d'un composant est le fichier XML produit par le modèle de composant ciblé par le profil (par exemple le descripteur Fractal généré par le modèle Fractal). Si la compilation est concluante, l'outil construit une archive composée de la description de l'architecture du composant et de sa NFS. La NFS est stockée sous la forme d'un fichier XML. Ce compilateur est une version modifiée du compilateur OCL (OCL Compiler (Dresden, 2002)) ;

– *Evaluateur de NFS :* cet outil vérifie le respect ou non d'un contrat par la version d'un composant qui lui est soumis. Pour ce faire, il récupère le fichier XML contenant la NFS du composant concerné et les descripteurs du composant avant et après évolution. Pour l'instant, cet outil n'est capable d'évaluer que les contrats écrits dans les profils xArch et Fractal.

5.2. *La problématique de l'évaluation d'un contrat*

Le langage ACL n'est pas un langage mais une collection de langages. Chaque profil ACL définit en fait un langage. La difficulté est de fournir autant d'évaluateurs différents qu'il y a de profils (actuellement 5). Nous avons trouvé plus judicieux, de par les ressemblances que présentaient les métamodèles de chacun des profils, de définir : un métamodèle façade auquel on associe un unique évaluateur et autant de traducteurs d'un profil vers ce langage intermédiaire. Ce métamodèle façade forme avec le langage CCL ce que nous appelons 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 au schéma XML de ArchMM. Ce modèle est une instance d'ArchMM générée par transformation XSL 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 métamodèle façade suffisamment riche et abstrait pour fournir un espace de traduction sans perte depuis chacun de nos 5 profils. Nous avons donc bâti ArchMM sur une étude comparative de deux types de métamodèles. Le premier concerne les langages de description d'architecture et le second, les technologies de composants. Une étude sur les ADL Acme, Koala (van Ommering *et al.*, 2000), xADL (Dashofy *et al.*, 2005), et d'autres, a été menée[1]. Pour la deuxième catégorie de métamodèles nous avons comparé les technologies de composants : EJB (*Enterprise JavaBeans*) de Sun Microsystems (Sun-Microsystems, 2003) et CCM (*CORBA Component Model*) de l'OMG (OMG, 2002). Nous avons jugé intéressant d'étendre notre étude sur le modèle de composants hiérarchique Fractal du consortium ObjectWeb et sur la partie du métamodèle UML concernée par les composants. Une synthèse des différents éléments architecturaux supportés par ces modèles est présentée sur la figure 10.

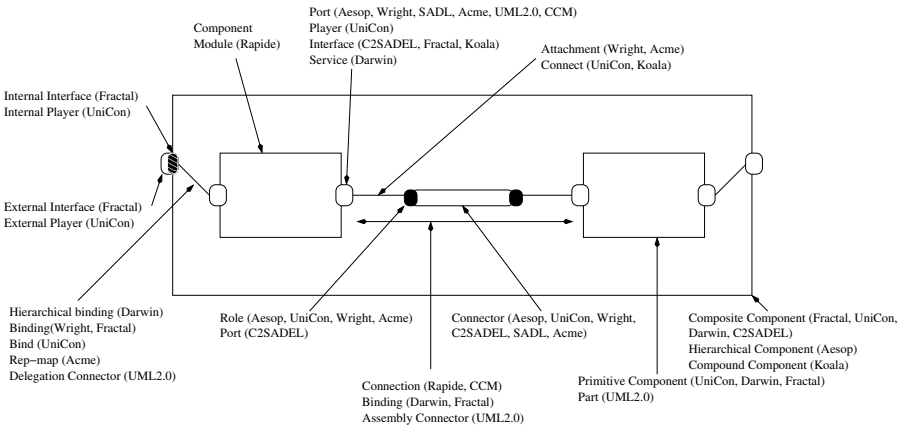

Figure 10. *Représentation graphique des abstractions architecturales dans les ADL, UML 2 et les technologies de composants*

Il faut noter également les points qui suivent. 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-avant. 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).

1. Nous nous sommes basés également sur un autre état de l'art fait il y a quelques années par Medvidovic (Medvidovic *et al.*, 2000).

5.3. *Description du profil standard ArchMM*

Suite à l'étude précédente, nous avons élaboré un profil standard constitué de CCL et d'un métamodèle nommé *ArchMM*. Ce métamodèle unifie et abstrait les concepts architecturaux représentés par tous les autres métamodèles de composant.

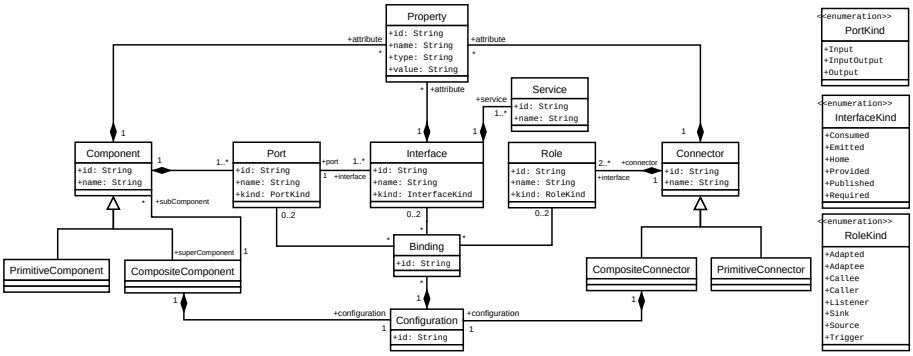

Figure 11. *ArchMM : le métamodèle façade*

La figure 11 présente le métamodèle ArchMM dans un format MOF (OMG, 2003). Dans ce métamodèle, 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 métamodèle met en avant les concepts suivants :

– 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 et 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 ADLs 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 métamodèle, une interface ou un rôle peut participer dans plusieurs attachements ;

– une description hiérarchique associe à un composant ou à un connecteur composites une structure interne. Cette structure interne est décrite par une configuration de sous-composants et connecteurs. Nous supposons 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 métamodèles, comme les composants, les interfaces requises ou fournies. Cependant, certains des concepts présents dans ArchMM n'existent pas dans d'autres métamodèles. Ces concepts, comme les ports, les `binding`, 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.

5.4. *Quelques exemples d'AD écrites dans le profil ArchMM*

Dans cette section, nous illustrons quelques contraintes architecturales définies avec ce profil pour montrer la puissance d'expression non seulement de CCL mais plus généralement du profil ArchMM. La première contrainte décrit une configuration de composants organisée selon le pattern architectural *pipeline* (Shaw *et al.*, 1996). Un pipeline est un style architectural décrivant les quatre contraintes architecturales suivantes :

– un composant (filtre) ne peut posséder plus d'une interface requise et interface fournie ;

– un connecteur (pipe) unique doit relier deux composants ;

– tous les connecteurs sont orientés dans le même sens ;

– l'ensemble des connecteurs ne doit pas former un circuit.

Dans le contrat d'évolution d'un composant (`nomComposant`) dont la structure interne est organisée comme un pipeline, la contrainte suivante doit être définie :

```
context nomComposant: CompositeComponent inv:
(self.configuration.binding.interface
->select(i|i;kind="Required").port.component =
(self.subComponent))
and
(self.configuration.binding.interface
->select(i|i;kind="Provided").port.component =
(self.subComponent))
```

La première partie de cette contrainte signifie que tout sous-composant doit avoir un et un seul lien *via* son interface requise. La seconde fait de même pour les interfaces fournies. Ces deux contraintes garantissent ensemble la structure en pipeline de la configuration interne de `nomComposant`.

Le second exemple illustre la contrainte suivante :

Aucun connecteur ne doit traverser la structure interne d'un composant.

Cette contrainte peut être décrite, sur le métamodèle ArchMM, de la manière suivante :

```
context nomComposant: CompositeComponent inv:
self.configuration.binding
->forAll(b|self.subComponent.port.interface
->includes(b.interface))
```

La contrainte ci-avant stipule le fait que, *pour tous les liens qui constituent la confi guration du composant composite, chaque interface d'un port d'un des liens doit appartenir à l'un des sous-composants du composite*.

Le troisième 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 comme suit :

```
context nomComposant:CompositeComponent inv:
((self.port.interface
->select(i:Interface|i.kind = 'Provided')->added()->size()<2) and
(self.port.interface->modified()->size()=0) and
(self.port.interface->deleted()->size()=0))
```

6. Conclusion

Dans cet article, nous avons présenté une méthode et un outil pour améliorer l'activité d'évolution à froid dans le monde des composants. Cette méthode propose de documenter formellement les liens unissant les attributs qualité d'une application aux choix architecturaux qui visent à leur obtention. Dans la mesure où ce sont les attributs qualité qui déterminent pour une part importante l'architecture d'une application, cette documentation facilite l'étape de compréhension en fournissant des informations essentielles aux personnes en charge de l'évolution. Les contraintes architecturales dont on use pour documenter ces choix architecturaux ne sont pas qu'un autre formalisme pour les annotations classiques de modèles ou de codes ou pour décrire des patrons ou des styles. En effet, le fait de pouvoir d'une part, les évaluer et d'autre part, les associer à des spécifications qualités autorise de nouveaux usages ; la mise en place d'outils d'assistance usant de cette documentation devient possible.

Un algorithme automatisable contribuant à la vérification *a priori* de la non-régression des attributs qualité d'une application a ainsi été proposé. Cet algorithme vient compléter les tests de non-régression et limiter les risques qu'ils se montrent non passants. Mais il ne les remplace pas. Car une technique de détection au plus tôt, si elle est moins coûteuse, introduit en contrepartie un niveau d'incertitude. La perte de choix architecturaux est un indicateur probable d'altération des propriétés non fonctionnelles qui leur sont associées, sans pour autant que l'on puisse quantifier cette perte ni même affirmer qu'une perte sera constatée. En effet, l'abandon de deux choix architecturaux indépendants peut très bien conduire à une nouvelle architecture qui pourrait malgré tout conserver et même améliorer certaines propriétés fonctionnelles supposées se dégrader. L'incertitude est le prix à payer pour une détection au plus tôt. Pour pallier cette faiblesse, nous travaillons sur la quantification *via* l'usage de métriques de l'influence d'un choix architectural sur les propriétés non fonctionnelles qui lui sont associées.

La génération de cette documentation a un coût. Il est donc important de trouver un niveau de granularité pour les choix architecturaux influençant significativement les propriétés d'interface. Dans cette approche, il est donc important de trouver un juste milieu et de développer une pratique similaire à celle que l'on doit mettre en place traditionnellement pour déterminer les commentaires à placer dans un code : ni trop, ni trop peu et toujours à valeur ajoutée manifeste. Une validation industrielle est en cours depuis septembre 2005 pour évaluer le retour sur investissement de cette méthode et 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 par de nombreux intervenants.

D'autres prolongements de ce travail sont possibles. Le langage ACL, en séparant la partie langage de prédicats, des abstractions architecturales à contraindre, peut s'utiliser dans tous types de paradigme de développement. Si, pour des raisons historiques, les profils actuels sont dédiés uniquement au paradigme orienté composant, rien n'empêche de porter l'approche sur d'autres paradigmes (orienté objets, fonctionnel, impératif, etc.). Il suffit pour cela de proposer les métamodèles *ad hoc*. Rien n'empêche également, d'étendre à loisir la portée d'un métamodèle existant pour lui adjoindre des abstractions complémentaires, ou même de créer plusieurs métamodèles pour un même langage. Nous réfléchissons, par exemple, à l'introduction d'abstractions comportementales venant compléter les abstractions structurelles existantes pour permettre l'écriture de règles limitant les évolutions possibles de la dynamique d'une application (par exemple contraignant le nombre des états et transitions d'un système séquentiel). Enfin, les techniques mises au point pour une évolution à froid, restent applicables pour contrôler les évolutions d'une application disposant au cours de son exécution, de moyens pour muter, par elle-même, ou *via* des interfaces de reconfiguration.

7. Bibliographie

Allen R., A Formal Approach to Software Architecture, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May, 1997.

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

Bennett K., « Software evolution : past, present and future », *Information and Software Technology*, vol. 38, n° 11, 1996, p. 671-732.

Briand L. C., Labiche Y., Di Penta M., Yan-Bondoc H. D., « An Experimental Investigation of Formality in UML-Based Development », *IEEE Transactions on Software Engineering*, vol. 31, n° 10, p. 833-849, October, 2005.

Bruneton E., Thierry C., Leclercq M., Quéma V., Jean-Bernard S., « An Open Component Model and its Support in Java », *Proceedings of the International Symposium on Component-based Software Engineering. Held in conjunction with ICSE'04*, Edinburgh, Scotland, may, 2004.

Chapin N., Hale J. E., Khan K. M., Ramil J. F., Tan W. G., « Types of Software Evolution and Software Maintenance », *Journal of Software Maintenance and Evolution : Research and Practice*, vol. 13, p. 3-30, 2001.

Dashofy E. M., van der Hoek A., Taylor R. N., « A Comprehensive Approach for the Development of Modular Software Architecture Description Languages », *ACM Transactions On Software Engineering and Methodology*, vol. 14, n° 2, p. 199-245, 2005.

Dresden T. U., « OCL Compiler Web Site », http ://dresden-ocl.sourceforge.net/, 2002.

Erlikh L., « Leveraging Legacy System Dollars for E-Business », *IEEE IT Professional*, 2000.

Frankl P. G., Rothermel G., Sayre K., Vokolos F. I., « An empirical Comparison of Two Safe Regression Test Selection Techniques », *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'03)*, Roma, Italy, p. 195-204, 2003.

Frolund S., Koistinen J., QML : A Language for Quality of Service Specification, Technical Report of Hewlett-Packard Laboratories HPL-9810, February, 1998.

Garlan D., Monroe R. T., Wile D., « Acme : Architectural Description of Component-Based Systems », *in* G. T. Leavens, M. Sitaraman (eds), *Foundations of Component-Based Systems*, Cambridge University Press, p. 47-68, 2000.

ISO, « Software Engineering - Product quality - Part 1 : Quality model », , International Organization for Standardization web site. ISO/IEC 9126-1. http ://www.iso.org, 2001.

ISR, « xArch web site », Institute for Software Research Web Site : http ://www.isr.uci.edu/architecture/xarch/, 2002.

Kemerer C. F., Slaughter S., « An empirical Approach to Studying Software Evolution », *IEEE Transactions on Software Engineering*, vol. 25, n° 4, p. 493-509, 1999.

Lehman M., Belady L., *Program Evolution : Process of Software Change*, London : Academic Press, 1985.

Lehman M. M., « Laws of Software Evolution Revisited », *Lecture Notes in Computer Science*, vol. 1149, p. 108-124, 1997.

Lientz B. P., Swanson E. B., « Problems in Application Software Maintenance », *Communiactions of the ACM*, vol. 24, n° 11, p. 763-769, 1981.

Luckham D. C., Kenney J. L., Augustin L. M., Vera J., Bryan D., Mann W., « Specification and Analysis of System Architecture Using Rapide », *IEEE Transactions on Software Engineering*, vol. 21, n° 4, p. 336-355, 1995.

Magee J., Kramer J., « Dynamic Structure in Software Architectures », *Proceedings of the fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96)*, San Francisco, USA, p. 3-14, 1996.

McKee J., « Maintenance as Function of Design », *Proceedings of AFIPS National Computer Conference*, Reston, Virginia, USA, p. 187-193, 1984.

Medvidovic N., Architecture-Based Specification-Time Software Evolution, PhD thesis, University of California, Irvine, 1999.

Medvidovic N., Taylor N. R., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n° 1, p. 70-93, 2000.

Microsoft, « COM : Component Object Model Technologies », , http ://www.microsoft.com/com/, 2005.

Moriconi M., Qian X., Riemenschneider R. A., « Correct Architecture Refinement », *IEEE Transactions on Software Engineering*, vol. 21, n° 4, p. 356-372, April, 1995.

Occello A., Dery-Pinna A.-M., « An adaptation-safe model for component platforms », *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, Nice, France, July, 2004.

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.

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.

Rajlich V., « Modeling software evolution by evolving interoperation graphs », *Annals of Software Engineering*, vol. 9, p. 235-248, 1999.

Ramil J., Lehman M., « Defining and applying metrics in the context of continuing software evolution », *Proceedings of the 7th International Software Metrics Symposium (METRICS)*, p. 199-209, April, 2001.

Robles G., Amor J., Gonzalez-Barahona J., Herraiz I., « Evolution and growth in large libre software projects », *Proceedings of the 8th International Workshop on Principles of Software Evolution*, p. 165-174, September, 2005.

Robson D., Bennett K., Cornelius B., Munro M., « Approaches to Program Comprehension », *Journal of Systems and Software*, vol. 41, p. 79-84, 1991.

Seacord R. C., Plakosh D., Lewis G. A., *Modernizing Legacy Systems : Software Technologies, Engineering Processes, and Business Practices*, SEI Series in Software Engineering, Pearson Education, 2003.

Shaw M., DeLine R., Klein D. V., Ross T. L., Young D. M., Zelesnik G., « Abstractions for Software Architecture and Tools to Support Them », *IEEE Transactions on Software Engineering*, vol. 21, n° 4, p. 314-335, 1995.

Shaw M., Garlan D., *Software Architecture : Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

Sun-Microsystems, « Enterprise JavaBeans(TM) Specification, version 2.1 », , http ://java.sun.com/products/ejb, November, 2003.

Tibermacine C., Fleurquin R., Sadou S., « NFRs-Aware Architectural Evolution of Component-based Software », *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, ACM Press, Long Beach, California, USA, p. 388-391, November, 2005.

van Ommering R., van der Linden F., Kramer J., Magee J., « The Koala Component Model for Consumer Electronics Software », *IEEE Computer*, vol. 33, n° 3, p. 78-85, March, 2000.

xAcme : Acme Extensions to xArch, School of Computer Science Web Site : http ://www-2.cs.cmu.edu/ acme/pub/xAcme/, 2001.